Universidade de Aveiro
2022

Tiago Saraiva Fernandes

Deep Learning para a Classificação de Ruídos Transitórios e Sinais nos Detetores LIGO

Deep Learning for the Classification of Transient Noises and Signals in the LIGO Detectors

Universidade de Aveiro
2022

Tiago Saraiva Fernandes

Deep Learning para a Classificação de Ruídos Transitórios e Sinais nos Detetores LIGO

Deep Learning for the Classification of Transient Noises and Signals in the LIGO Detectors

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Física, realizada sob a orientação científica do Doutor António de Aguiar e Pestana de Morais, Investigador Doutorado de Nível 1 do Departamento de Física da Universidade de Aveiro, e co-orientação do Doutor Felipe Ferreira de Freitas, Investigador Doutorado de Nível 1 do Departamento de Física da Universidade de Aveiro.

o júri / the jury

presidente / president	**Professora Doutora Florinda Mendes da Costa** Professora Associada com Agregação da Universidade de Aveiro
arguente / examiner	**Professor Doutor Alejandro Torres Forné** Professor Auxiliar da Universidade de Valência
orientador / supervisor	**Doutor António de Aguiar e Pestana de Morais** Investigador Doutorado de Nível 1 da Universidade de Aveiro

agradecimentos / acknowledgements

Em primeiro lugar, quero agradecer ao meu orientador, Doutor António Morais, e ao meu co-orientador, Doutor Felipe Freitas, por esta oportunidade.

Quero também prestar um profundo agradecimento ao Professor António Onofre por ter aceitado acompanhar o meu trabalho e pelo inprescindível apoio que prestou. Agradeço também ao Samuel e à Teresa, pelas discussões e partilhas que certamente contribuiram para enriquecer este trabalho.

Finalmente, agradeço à minha família, aos meus amigos e a todos os que me apoiaram ao longo deste percurso.

Um enorme obrigado à Margarida, por me ter motivado para continuar quando tudo parecia perdido, e por muito mais.

Palavras-chave

deep learning, aLIGO, ruído transitório, ondas gravitacionais, classificação de imagens

Resumo

Neste trabalho, dados dos detetores aLIGO recolhidos nos dois primeiros períodos de observação de LIGO e Virgo (O1 e O2), na forma de espectrogramas, foram classificados usando modelos de Deep Learning baseados em redes neuronais convolucionais. Além de serem usados modelos treinados do zero, também se testaram modelos pré-treinados, e os resultados foram comparados.

Para isso, começou por se fazer uma breve introdução às ondas gravitacionais e sua deteção nos detetores de LIGO. Foram também introduzidos os fundamentos relacionados com algoritmos de Deep Learning e das boas práticas para o treino de modelos para a classificação de imagens.

Verificou-se que usar os diferentes canais de cor das imagens para apresentar informação com diferentes janelas temporais melhora os resultados dos modelos e que, além disso, arquiteturas pequenas são capazes de separar eficazmente as 22 classes presentes no dataset Gravity Spy. Adicionalmente, a técnica de *transfer learning* permite acelerar a fase de treino e obter classificadores com um desempenho competitivo.

Os melhores modelos obtiveram um F1-score médio (macro) de 96.84% para o modelo pré-treinado e de 97.18% para o modelo base treinado do zero. Estes resultados estão em linha com os melhores resultados encontrados na literatura para o mesmo dataset. Adicionalmente, os modelos foram testados em sinais reais de ondas gravitacionais de Coalescências Binárias Compactas detetadas por LIGO, obtendo sensibilidades de, respetivamente, 25% e 75%, apesar de terem sido treinados com um número reduzido de sinais provenientes de simulações de ondas gravitacionais.

Keywords

deep learning, aLIGO, transient noise, gravitational waves, image classification

Abstract

In this work, data from the aLIGO detectores collected during the first two aLIGO and AdV observing runs (O1 and O2), in the form of spectrograms, were classified using Deep Learning models based on Convolutional Neural Networks. As well as training models from scratch, pre-trained models were also employed, and their performance compared.

Initially, a brief theoretical introduction on gravitational wave detection was performed, focusing on the LIGO detectors. In addition, the foundations of Deep Learning and current best practices for the training of image classification models were also presented.

The computational experiments showed that encoding information from different time windows in the different colour channels enhanced the performance of the models and that small architectures were capable of separating the 22 classes present in the Gravity Spy dataset. Moreover, transfer learning was able to accelerate the training process and achieve classifiers with competitive performance.

The best models obtained a macro-averaged F1 score of 96.84% (fine-tuned model) and 97.18% (baseline trained from scratch), which are in line with the best results in the literature for the same dataset. In addition, these models were evaluated on real gravitational wave signals from Compact Binary Coalescences from the first two aLIGO and AdV observing runs, and they achieved recalls of 75% and 25%, respectively, while only having been trained with a small number of signals from gravitational wave simulations.

Contents

List of Figures

List of Tables

List of Acronyms

AdV Advanced Virgo. 8, 25, 41, 43

aLIGO Advanced Laser Interferometer Gravitational-Wave Observatory. iii, 1, 5–8, 25, 41, 43

BBH Binary Black Hole. iii, 1, 2, 7, 8

CBC Compact Binary Coalescence. 1, 2, 6, 7, 41, 42

CNN Convolutional Neural Network. 1, 9, 17, 24, 35, 36, 40, 41, 43, 44

CPU Central Processing Unit. 11

DL Deep Learning. 1, 9–11, 18, 22

GPU Graphical Processing Unit. 11, 16, 24, 27

GW Gravitational Wave. iv, 1–8, 11, 40–43

ILSVRC ImageNet Large Scale Visual Recognition Challenge. 10, 11, 17, 18

LIGO Laser Interferometer Gravitational-Wave Observatory. 1, 5, 6, 8

LVK LIGO-Virgo-KAGRA. 8

MF Matched-Filtering. 6

ML Machine Learning. iii, 1, 9, 10, 14, 15, 22

MLP MultiLayer Perceptron. 12–14, 17

NN Neural Network. 9, 11, 13–15, 17, 19, 21, 25

ReLU Rectified Linear Unit. 19, 44, 45

TLU Threshold Logic Unit. 11, 12

Chapter 1

Introduction

On September 14, 2015, the Advanced Laser Interferometer Gravitational-Wave Observatory (aLIGO) [1] made the first direct observation of a Gravitational Wave (GW), which was caused by the violent merger of a pair of black holes more than 1.3 billion years ago [2]. The detection of gravitational waves initiated a novel era of astrophysics, as it provides a new window for observing the Universe and an unique way to study fundamental physics - for instance, allowing to test general relativity in new regimes [3] and presenting new ways to measure the Hubble constant [4]. Since its very first detection, aLIGO, most recently in collaboration with Virgo [5], has already detected 90 candidate signals from Compact Binary Coalescences (CBCs), most of which resulting from Binary Black Hole (BBH) mergers [6].

Measuring a gravitational wave requires detectors with enormous sensitivities, as the displacements produced by the GWs that reach Earth are smaller than the radius of a proton [2]. For this reason, aLIGO relies on interferometers with 4 km arms and cutting-edge designed and fabricated subsystems to amplify the gravitational signals and isolate the sensitive components from non-gravitational wave disturbances such as seismic noise. Nevertheless, gravitational wave detection is still affected by the existence of noise fluctuations of non-astrophysical origin, the so-called glitches [7], which contaminate the GW data and negatively affect the detection pipelines, as they can trigger false positives or overlap with GW signals.

Therefore, there is a need for strategies that reduce the rate of glitches in the aLIGO data, either by performing denoising, that is, subtracting the glitches from the data, or by attenuating or eliminating the sources of the glitches, when that is possible. Both strategies have a need for the study of the different glitches and their categorization into classes, as it allows to enhance the denoising mechanism and to search for correlations between each glitch type and the numerous auxiliary detectors at the Laser Interferometer Gravitational-Wave Observatory (LIGO).

The problem of glitch classification is characterized by the need to learn patterns from structured labelled data, either in raw time-series form or as spectrograms (time-frequency-energy images). Therefore, it is a good candidate for supervised Machine Learning (ML) algorithms, i.e., algorithms that learn from labelled examples instead of being explicitly programmed [8]. Of particular interest is Deep Learning (DL), a specific type of machine learning that has the advantage of not needing human feature engineering, and has achieved breakthroughs in tasks related to image, video, speech and audio processing, setting the state of the art and even achieving super-human performance on some of those tasks [9]. Having the previous reasons in mind, this work focuses on the application of supervised deep learning methods to the task of glitch classification in the aLIGO detectors.

This document is divided in five chapters, this being the first. In chapter 2, gravitational waves are introduced, as well as the main detection techniques, and some issues of the current GW detection paradigm are presented. Chapter 3 presents the theoretical foundations of Deep Learning and Convolutional Neural Networks (CNNs), as well as some modern performance-enhancing techniques that are used throughout this work. Chapter 4 contains the core of this work, introducing the methods and results of the application of deep learning methods to glitch classification, and includes a brief study of the application of glitch classifiers to the task of gravitational wave detection. Finally, the main conclusions are summarized in chapter 5.

Chapter 2

Gravitational wave detection

This chapter gives an overview of the detection of Gravitational Waves (GWs). In section 2.1, the concept of gravitational waves, the main sources of astrophysical GWs, and the effect of GWs on matter is presented. Then, section 2.2 introduces the first generation of GW detectors, based on resonant bar detectors. Finally, section 2.3 focuses on the modern GW detectors based on interferometry. In this section, both the basic detection principle and the necessary enhancements are examined. Moreover, the unwanted glitches and the different detection pipelines are presented. A brief account of the first direct GW detection and the prospects for GW detection are also included in this section.

2.1 Gravitational waves

The concept of gravitational waves was introduced in 1905 by Poincaré [10], who proposed that gravity was not instantaneous but instead transmitted at the speed of light by a wave, appropriately called a gravitational wave. With the publication of Albert Einstein's Theory of General Relativity in 1915 [11], the paradigm of gravitation was revolutionized: gravity was not a force but a manifestation of the curvature of the space-time. Shortly after, he also conjectured that the existence of gravitational waves, similar to the already known electromagnetic waves, was a possibility. However, for a long time GWs were deemed more as a mathematical artifact than a real phenomenon.

It was not until 1974 that the first proof of the existence of gravitational waves was found, after the discovery of a binary pulsar system, which general relativity had predicted to radiate gravitational waves. The orbits of the stars were closely monitored for several years, and it was found that they closely followed the predictions of general relativity, resulting in the first indirect detection of gravitational waves [12]. Since then, more observations of binary pulsars have corroborated the existence of GWs. Despite these advances, the first direct observation of a gravitational wave was only achieved in 2015, as detailed in section 2.3.5, in one of humanity's greatest scientific achievements.

Today, it is widely agreed that gravitational waves are real. They are ripples in the fabric of space-time, caused by accelerating masses that lose energy emitted in the form of undulations of the space-time itself, which propagate in all directions through the Universe at the speed of light, bringing information about their genesis.

2.1.1 Sources of gravitational waves

According to General Relativity, every object that accelerates originates GWs. However, the masses and accelerations of objects on Earth are far too small to produce detectable GWs. Instead, the waves detected so far are astrophysical GWs, originated in cataclysmic events outside of our Solar System.

Astrophysical gravitational waves can have many different origins, from the Big Bang itself to a single spinning massive object, such as neutron star. The former waves are thought to be mixed in the so-called stochastic signal, which is a random mixture of many small GWs, while the later are due to the imperfections of the spherical shape of the object, originating a continuous wave, i.e., a wave with constant frequency and amplitude. There are also waves from impulsive sources that emit bursts of intense gravitational waves. [13]

Nevertheless, the most studied class of GWs corresponds to the Compact Binary Coalescences (CBCs). This type of wave is produced by orbiting pairs of massive and dense objects such as neutron stars and black holes. So far, all the GW events detected belong to this type, which can be further divided in three categories, each associated to a unique wave pattern: Binary Neutron Star, Binary Black Hole (BBH), and Neutron Star-Black Hole Binary. In these binary systems, the two objects revolve around each others for millions of years, slowly losing energy as they emit gravitational waves. They gradually become closer and thus emit stronger GWs, until they finally collide, merging together and releasing huge amounts of energy. [14]

2.1.2 Effect of gravitational waves on matter

In order to better understand the nature of GWs and the strategies for their detection, it is useful to consider how a GW affects a circular ring of particles, initially at rest, when the motion of the GW is perpendicular to the direction to the plane of the particles. Consider Fig. 2.1, with the particles initially arranged in a circle (and not attached to anything or each other) as shown in the left diagram, and a gravitational wave which propagates downwards in a direction perpendicular to the paper. Initially, the particles are compressed in the vertical direction and stretched horizontally, starting to form an ellipsis until the deformation reaches its maximum, at which point the process is reversed, the particles reach the initial position and continue to be stretched and squashed until the direction in which it initially compressed is at a maximum. This process continues until the wave has passed through the particles. [15]

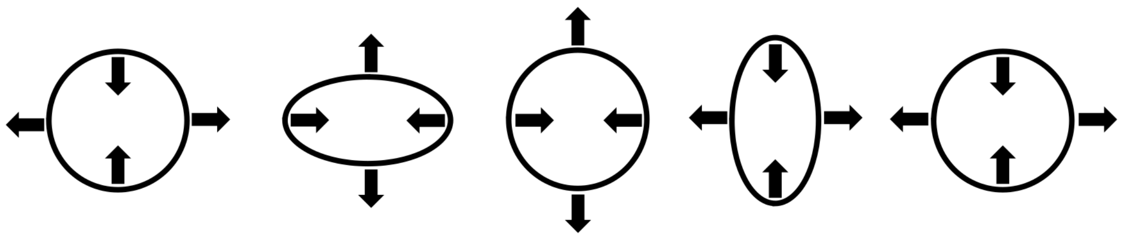

Figure 2.1: Scheme of the effect of a gravitational wave, going in a downward direction (i.e. entering the paper) on a ring of particles. The diagram on the left shows the initial state and the other diagrams show subsequent instants in time. The amount of the displacements is highly exaggerated for visualization purposes. Figure inspired by [15].

Having this behaviour in mind, it could be naively suggested that it is very simple to detect GWs, as one only needs to compare distances between perpendicularly placed pairs of particles in free motion as a wave passes through. This is not the case mainly for two reasons. First, the squashes and stretches shown in Fig. 2.1 are actually much smaller, in the order of zeptometers (10^{-21} m). Moreover, even if one could use an instrument like a ruler to measure such small displacements, it would not stretch the same way as the free particles, due to the elastic properties of the ruler. For these reasons, more ingenious strategies are needed for the detection of gravitational waves.

2.2 Primitive gravitational wave detectors

The first instruments developed to detect GWs were the so-called Weber bars, which are large cylinders of metal proposed by Joseph Weber that would, in theory, show resonant vibrations produced by the transit of a GW [16]. The first Weber bars consisted of a big aluminum cylinder, weighting three tonnes, which was hanging from a system designed to isolate environmental vibrations. Each instrument was equipped with a belt of piezoelectric crystals, which would convert the mechanical vibrations into electrical pulses. Two such detectors were built and located 950 km away, allowing to eliminate local signals by discarding signals which were not recorded in both detectors within an appropriate time interval[1].

Since 1969, Weber made multiple publications claiming the detection of GW signals; however, theoretical physicists soon found out that some of his measurements did not agree with established theories and, more importantly, none of the efforts to replicate his detectors, even with enhanced sensitivities achieved through cooling of the devices, was able find a gravitational wave [15]. Thus, this class of detectors was mostly abandoned in detriment of new detection strategies, which is greatly

[1]Modern gravitational wave detectors still take advantage of this technique to increase the significance of detections.

illustrated by Peter Kafka's statement in the 1975 International School of Cosmology and Gravitation [17]: "only a combination of extremely high quality and extremely low temperature will bring resonance detectors near the range where astronomical work is possible. (Another way which seems worth exploring, is laser interferometry with long 'free-mass antennas'.)".

As reported in section 2.3, the mentioned laser interferometry based methods turned out to revolutionize the field of gravitational wave detection. Nevertheless, methods based on cryogenic resonant bars are still studied, and experiments such as AURIGA [18] are still trying to detect GWs. At the time of the writing of this report, no detection of a gravitational wave has been made using a resonant bar.

2.3 Interferometric gravitational wave detectors

Since the 1960s, there have been several proposed schemes to detect gravitational waves using interferometers, which are all based in the classic Michelson interferometers.

2.3.1 Basic detection principle

Consider again the example at the end of section 2.1.2, where it was tried to measure the displacement between free particles as they were traversed by gravitational waves. Instead of trying to measure the distances, the time it takes by light to travel between them can be measured accurately, as the speed of light (in vacuum) is invariant and thus unaltered by a gravitational wave.

In principle, such measurement can be achieved with a very simple device, the Michelson interferometer, as schematized in Fig. 2.2. This simple setup comprises a light source, typically a laser, which is equally split by a beam-splitter (BS) into two equally long perpendicular arms with a mirror at the end. When the light is reflected back and recombines at the BS, interference patterns known as fringes are observed in the photodetector. When a GW passes through the interferometer at a certain direction, one of the mirrors slightly reduces its distance to the BS, while the other increases its distance. The difference in arm length originates a phase shift between the beams of the two arms, which is detected in the form of a variation of the intensity reaching the photodiode.

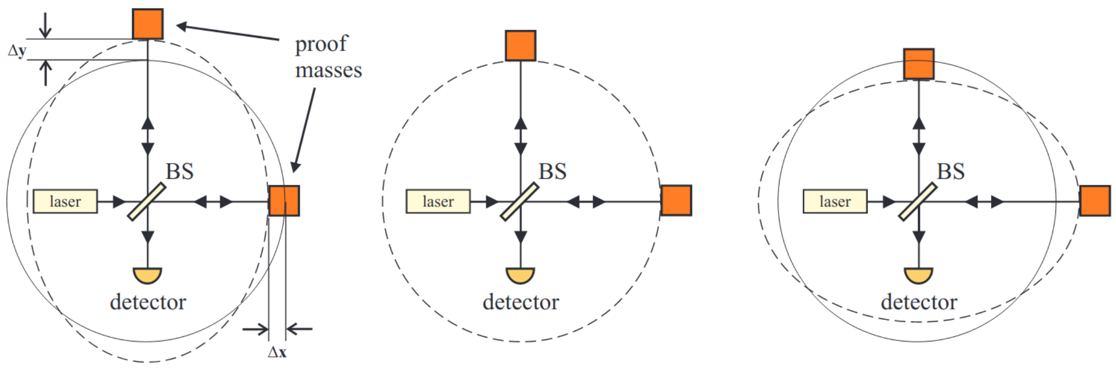

Figure 2.2: Configuration of a Michelson interferometer to measure gravitational waves. The image shows three sequential time instants as a wave is passing. The distance between each proof mass and the splitter changes, and this is measured at the detector. Figure reproduced from [19].

In practice, the detection is not so simple, as the GW signals that reach the Earth are very faint, making it difficult to extract them from the background noise. It turns out that the length of the interferometric arms is a limiting factor of its sensitivity. For instance, for a gravitational wave at 100 Hz, the optimal length of the arms of a Michelson interferometer would be 750 km [20]. As shown in section 2.3.2, modern interferometers overcome this issue by combining their huge arm length with ingenious modifications to enhance the signals of interest.

2.3.2 Large scale gravitational wave interferometers

The LIGO project was born from a collaboration between competing groups at the United States, which were working on small interferometric detectors for gravitational wave detection. After overcoming organizational difficulties, LIGO managed to obtain funding for the construction of two similar Michelson based interferometers with 4 km arms, one in Hanford (Washington, USA) and the other in Livingston (Louisiana, USA). From the start, the goal was to produce an evolutionary apparatus, which could be evolved by improving the interferometer components through different stages. The first stage, "initial LIGO" was designed to test the concept of a large scale detector, and the detection of GWs would only be likely in a later stage. The initial LIGO started operating in 2002. [15]

Simultaneously, two similar projects appeared in Europe. In Germany, the study of the noise and performance of a 30 m-arm detector culminated in the construction of a 600 m interferometer, GEO 600 [21], which started operating in 2002. Since then, GEO 600, due to its close collaboration with LIGO, has been an important development and test laboratory for technologies that eventually are implemented in the larger interferometers. In Italy, the Virgo collaboration managed to build a 3 km interferometer [22], which started operation in 2000. In 2007, Virgo and LIGO started a formal collaboration that included the exchange of data and joint analysis. The large detectors made joint observations from 2002 to 2010, but failed to detect gravitational wave signals. [15].

In late 2010, LIGO was disassembled for the construction of the second-stage detector, "advanced LIGO" (aLIGO). This major upgrade, which included the replacement of all the interferometer components with the ultimate goal of enhancing the initial LIGO sensitivity by a factor of 10, ended in March 2015. Among the notable changes are the use of much larger mirrors, the improvement of the passive seismic isolation system with a quadruple pendulum and the usage of glass fibers instead of metal wires, and the introduction of active seismic cancellation mechanisms. [1]

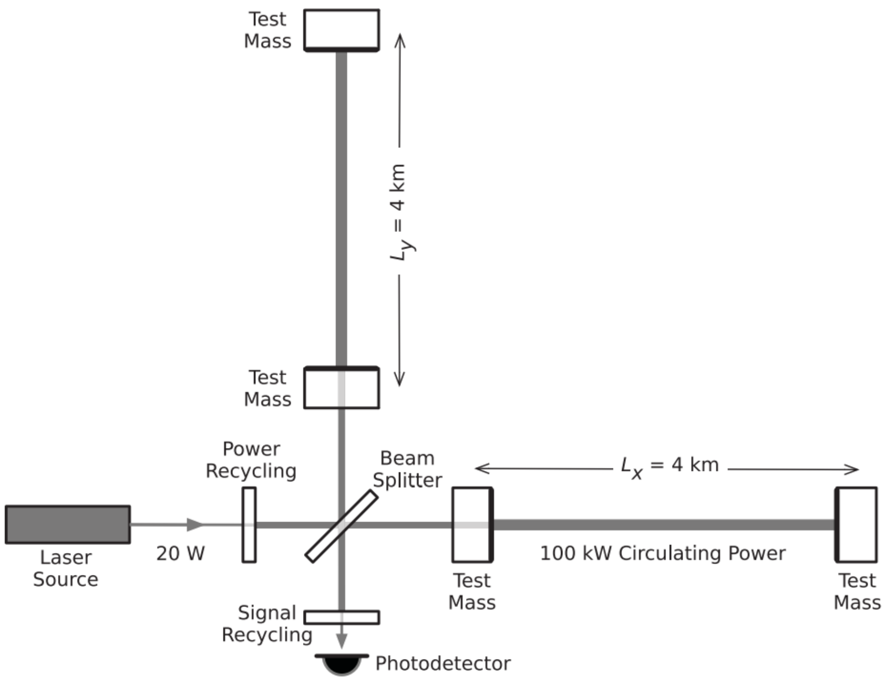

Figure 2.3: Simplified diagram of an aLIGO detector. Figure adapted from [2].

A simplified diagram of an aLIGO detector is shown in Fig. 2.3. This detector is based on the Michelson interferometer shown in Fig. 2.2, but with the necessary enhancements to achieve the required sensitivity to measure GWs. The laser source is a 1064 nm Nd:YAG laser stabilized in amplitude, frequency and beam geometry. Each arm contains a resonant Fabry-Perot cavity, formed by the two test mass mirrors, which multiplies the effect of a GW in the light phase by a factor of 300 and increases the power circulating in each arm. Moreover, a partially transmissive power recycling mirror provides additional buildup of the beam power circulating in the interferometer. A signal recycling mirror at the output is used to maintain a broad detector frequency response. [1, 2]

All components apart from the laser are kept in a ultra-high vacuum system. As mentioned before, the test masses are suspended at the final stage of a quadruple-pendulum system supported by an active seismic isolation platform, providing more than 10 orders of magnitude of isolation from seismic noise. Additional precautions are taken regarding the thermal noise of the test masses and the alignment of the optical components, in order to minimize noise sources. [1, 2]

The output port of the Michelson is held close to a dark fringe, resulting in a small amount of light leaving the output port and most of the light being kept circulating. The output light signal is measured by a homodyne detector and calibrated with less than 10% uncertainty in amplitude before it is sent to the detection pipelines. [2]

2.3.3 Glitches

The aLIGO detectors are designed to minimize the noise sources, enhancing the sensitivity of the instruments. Nevertheless, seismic noise dominates at low frequencies, while thermal noise from the coating of the test masses is relevant at intermediate frequencies. At higher frequencies, the detectors are fundamentally limited by the quantum noise. [1]

In the daily operation of the detectors, they are also affected by the existence of short, non-Gaussian, transient noise fluctuations that happen due to instrumental or environmental reasons, the so-called glitches [7], which contaminate the strain data and reduce the quantity and quality of the detections. Glitches vary widely in duration, frequency range, and morphology; moreover, the commissioning of the detectors between each observing run is expected to originate new types of glitches [7]. The most common glitch type is named "blip", and it resembles the merger of high mass binaries [23]. In the last observing run, these glitches happened at a rate of 4 per hour at the Livingston detector, and 2 per hour at Hanford, and they were responsible for a significant amount of the discarded data [23].

Overall, glitches occur quite frequently: despite the efforts to reduce the occurrence of glitches, even in the last aLIGO-Virgo's observing run the glitch rate was close to $1\,\mathrm{min}^{-1}$ [6]. Moreover, the origin of many of the glitch types, including the "blips", is still unknown [23]. Glitches can either raise false alarms when they occur simultaneously in multiple detectors or overlap with GW signals, reducing the effectiveness of both the modelled and unmodelled detection pipelines. Although possible, it is computationally expensive to artificially remove glitches from the signals, and this problem will increase as more GWs are detected in future observing runs [24]. The best way to reduce the amount of glitches is to identify their causes and fix them, which requires a proper classification of different glitches into well-defined classes. Having this information, it is easier to try to correlate each glitch type with the auxiliary sensors and unveil their cause.

2.3.4 Detection pipelines

In the data measured at LIGO, the gravitational signals are buried inside noise (see e.g. Fig. 2.4), so there is a need for methods that allow to search for the GW signals, preferably with low latency and high confidence. Currently, the array of detectors employs two distinct types of search methods: modelled and unmodelled searches.

Modelled searches are used to detect well-modeled GW sources, such as CBCs, using the technique of Matched-Filtering (MF), where waveform templates obtained from numerical relativity are correlated with the detectors' data, in order to extract signals from the noise. Even though MF is the optimal method only in the case of known waveforms embedded in stationary Gaussian noise, it is successfully used for the detection of GWs originated in CBCs [25].

However, other GW sources cannot be well modelled by general relativity, demanding the use of unmodelled searches. These are generic searches that make minimal assumptions about the signal morphology, and rely on identifying coincident excess power on the strain data. Due to their robustness, these algorithms are also used for CBC searches, complementing the MF methods. They allow to detect events with features such as misaligned spins, high mass ratios, or eccentric orbits, that could escape from the MF pipeline [25]. In fact, the signal GW150914 (see section 2.3.5) was first detected by a generic algorithm, within three minutes of data acquisition [2].

After the identification of a GW candidate by a search pipeline, bayesian inference algorithms are used to estimate the properties of the event, such as the sky localization, distance, and the masses and spins (in the case of CBCs). These algorithms can take hours to converge, although probability sky maps with comparable accuracy can be obtained within tens of seconds. [25]

Despite its success, matched-filtering is a costly method, as the execution time scales linearly with the number of waveform templates. This limits the number of waveforms in the template bank, as a bigger bank will slow the detection pipeline. Moreover, unmodelled searches have the disadvantage of not using all the available knowledge about the morphology of the signals expected from a CBC. An ideal CBC detection pipeline would remain low-latency, while containing the information about many examples over the full range in every parameter dimension. Nevertheless, this is impossible in the current paradigm, which gives rise to the need for new detection pipeline methods, perhaps taking advantage of Machine Learning tools.

2.3.5 GW150914: the first direct gravitational wave detection

The first aLIGO detection happened four days before the start of the inaugural official observing run, after a long night of last-minute tests [15]. On September 14, 2015, at 4:50 a.m. local time, the two detectors observed the coincident signal GW150914, shown in Fig. 2.4, within seven milliseconds of each other. In the time-frequency representation, the signals show the characteristic chirp morphology, where the frequency increases over time.

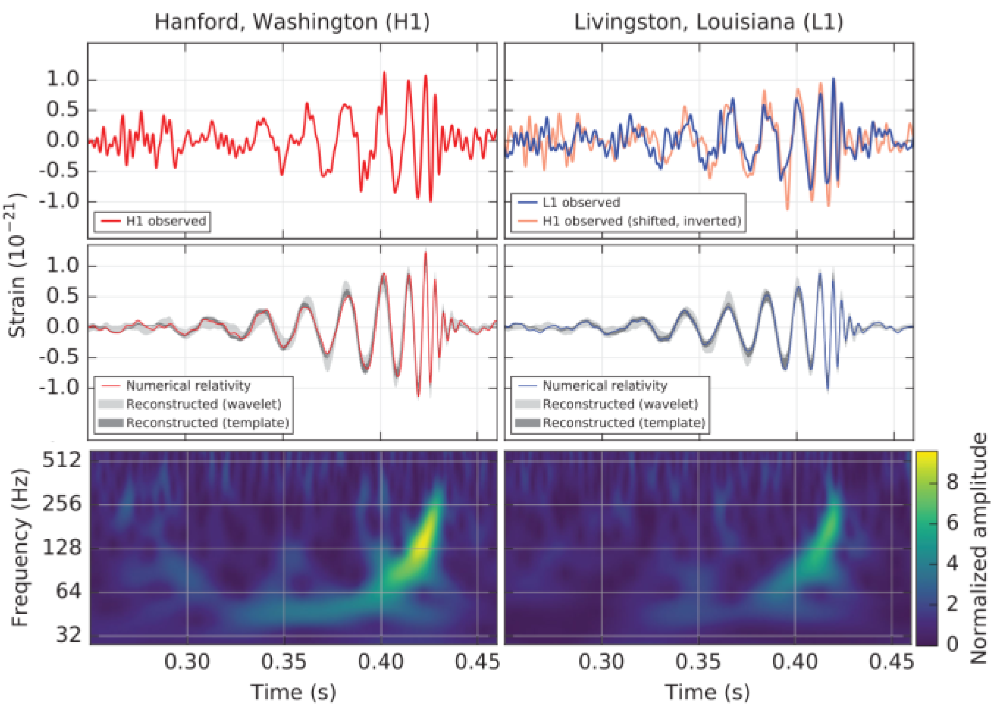

Figure 2.4: Gravitational wave event GW150914 observed by the Hanford and Livingston detectors. The top row shows the strain signals while the middle row shows the signal predicted by general relativity for a BBH system with the parameters recovered from GW150914 and the 90% credible regions for two independent waveform reconstructions. The bottom row shows a time-frequency representation of the strain data. Figure adapted from [2].

The signal was found to have been produced by the coalescence of two black holes, starting with the last phase of the orbital inspiral, followed by the merger of the two black holes, which corresponds to the maximum in the strain amplitude, and the subsequent ringdown of the final black hole. The estimated masses of the two black holes are $36\,M_\odot$ (solar masses) and $29\,M_\odot$, which resulted in the formation of a $62\,M_\odot$ black hole, meaning that $3\,M_\odot$ were released as energy in the form of GWs. This energetic event occurred at a distance of about 410 Mpc (approximately 1.3 billion light years) from Earth.

2.3.6 Prospects for gravitational wave observation

So far, the LIGO-Virgo-KAGRA (LVK) Collaboration[2] has conducted three observing runs. In the first observing run (O1), which took place between September 2015 and January 2016, the aLIGO detectors achieved a BBH range (computed for a $30\,\mathrm{M}_\odot + 30\,\mathrm{M}_\odot$ black hole system) of 740 Mpc [26]. O1 managed to detect three GW signals, all from BBH mergers [26].

After a commissioning phase to increase the detectors' sensitivity, the second observing run (O2) took place from the end of November 2016 to the end of August 2017. Here, the aLIGO detectors operated at a BBH range of 910 Mpc. Moreover, they were joined in the last month of O2 by the Advanced Virgo (AdV) detector, with a range of 270 Mpc [26]. During this run, eight GW signals were detected, including the first detection of a binary neutron star merger [26]. Furthermore, the addition of AdV allowed to enhance the accuracy of the localization of some of the events [26].

The third observing run (O3) started on April 2019 and ended prematurely due to the COVID-19 pandemic on March 2020. Here, aLIGO achieved BBH sensitivities between 990 and 1200 Mpc, while AdV operated at 500 Mpc [26]. O3 was very prolific regarding the detection of GW signals, with a total of 79 events, including the first confident observations of binary neutron star-black hole binaries [6].

After an intense period of upgrades and commissioning, the next observing run (O4) is scheduled to begin on March 2023 and last a full year [27]. The ranges of the aLIGO and AdV detectors will be significantly higher than O3 (60% and 100% increases, respectively) [27]. The KAGRA detector [28], located in Japan, is set to join the three detectors at the beginning of the run, albeit with a much lower sensitivity [27]. A fifth observing run (O5) is also planned to begin in 2026. Moreover, there are plans to join a new observatory, LIGO-India, to the array of networks. This observatory will have an aLIGO detector configured identically to the other two at the United States [26].

The increases in sensitivity and in the number of detectors will dramatically increment the rate of GW detections. However, this will accentuate the problems with the current detection pipelines. Since matched-filtering algorithms are run independently for each detector, the computational cost of the searches will increase. Moreover, unmodelled searches can be affected, as coincident glitches at two or more detectors will be more likely with the increase in the number of detectors. Therefore, there is a necessity for a detection pipeline that scales well with the number of detectors and is robust against the noise glitches as well as the reduction of the glitch rate.

[2]LVK Collaboration includes the LIGO Scientific Collaboration, as well as the Virgo Collabotation (which has been carrying joint analysis and co-authoring observational result papers since 2010) and the KAGRA Collaboration (which started in 2021 to co-author observational results from the second half of O3 onward).

Chapter 3

Deep Learning

This chapter contains the theoretical foundations of the Deep Learning (DL) methods used in this work. First, a brief introduction to Machine Learning (ML) is presented in section 3.1 and then, section 3.2 reports what is Deep Learning, how it became possible, and some of its applications. Afterwards, sections 3.3 to 3.6 focus on the theory of artificial Neural Networks (NNs), which are at the core of Deep Learning. In section 3.3, a brief history of the field is presented, along with the main theory behind NNs, while section 3.4 focuses on the task of training such networks. Moreover, Convolutional Neural Networks (CNNs), which are the cornerstone of many DL applications in the visual domain, are introduced in section 3.5, as well as the revolutionary concept of skip connections. Then, section 3.6 reports some algorithmic advances that allowed to overcome the Vanishing Gradient Problem and to effectively train deep networks. Following that, section 3.7 explains the metrics used to compare models in this work, and section 3.8 presents some important model hyperparameters and strategies to optimize them. Finally, this chapter ends in section 3.9, with a presentation of the `fastai` DL library and some of its useful methods, including the powerful technique of transfer learning.

3.1 Introduction to Machine Learning

A traditional computer program follows a set of rules created by a human in order to turn input data into an appropriate output. The Machine Learning (ML) paradigm turns this around: the computer is fed many examples and autonomously learns the rules that allow it to complete the desired task. Thus, a ML algorithm is one that is trained with data, and learns how to solve a task without being explicitly programmed to do so. More formally, in Mitchell's definition [29]: "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 two most common tasks are classification, in which the algorithm specifies which categories the input belongs to, and regression, where the program predicts a numerical value when given some input. Nevertheless, ML can tackle much more tasks, from anomaly detection to data synthesis. In the case of classification, a simple performance metric is accuracy, which is the proportion of examples that the algorithm classified correctly. For regression problems, mean squared error, i.e., the average squared difference between the predicted and real values, is a common choice. [8]

According to the experience that the algorithms get during the training phase, they can be categorized as unsupervised or supervised algorithms. In both cases, the algorithms experience an entire dataset, $\boldsymbol{X}$, which is a collection of m data points $\boldsymbol{x}^{(i)} = (x_0, x_1, ..., x_n)$, also called examples, each of which has n features. Unsupervised algorithms try to learn useful patterns from the structure of the dataset, by implicitly or explicitly estimating the probability distribution of the dataset. On the other hand, supervised learning algorithms experience a dataset where each example $\boldsymbol{x}^{(i)}$ has additionally an associated label $y^{(i)}$, which is the target for the prediction of the point. The latter algorithm category will be the focus of this work.

The main challenge of machine learning is that it does not suffice that the algorithm works well in the dataset used for training. Instead, the algorithm must generalize, that is, perform well on previously unseen data. As the algorithm is iteratively modified during training in order to minimize some measure of the training error, the error measured in the samples used for training is an over-confident measure of the generalization ability of the algorithm. For this reason, the training phase is performed using a training dataset and, after training, the algorithm is evaluated in a different dataset containing data not previously seen by the model, the so-called test dataset. For an accurate estimation, it is important that the two datasets are sampled from the same distribution and, furthermore, that the test set is not used to guide the choices made in the training phase. [8]

Consequently, a good ML algorithm must make the training error small while also keeping the gap between training and test error as small as possible. A large training error is usually associated with underfitting, which occurs when the model does not possess sufficient capacity to find the underlying relationships between the data. For example, underfitting happens when a linear function is fitted to data that follow a quadratic distribution, as seen in Fig. 3.1 (left). On the other hand, the gap between the training error and the testing error will be large if the model is overfitted to the training dataset. This means that instead of learning general patterns, the model is too optimized for the training set and learned patterns that are specific from the training data but not relevant to new data. Overfitting happens, for instance, when a m-degree polynomial is used to fit a dataset with $m-1$ points, as shown in Fig. 3.1 (right). To summarize, a good model needs to be between underfitting and overfitting, that is, having enough capacity to learn meaning patterns from the data, but being simple enough to avoid learning the noise in the training dataset. This is the case of Fig. 3.1 (middle).

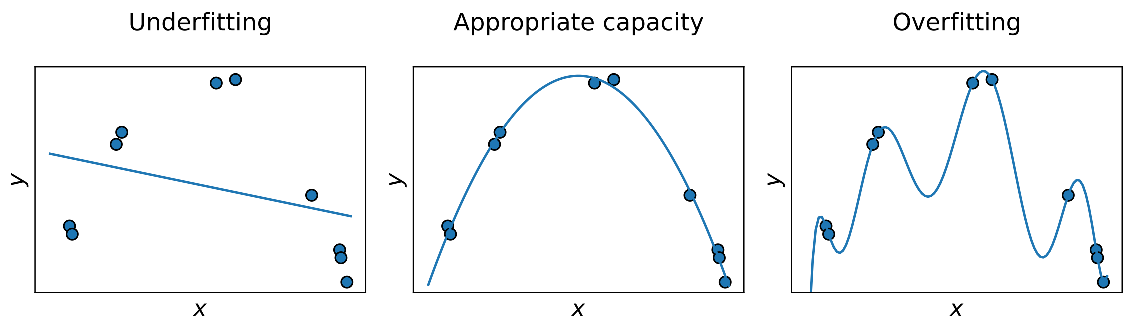

Figure 3.1: Three models fitted to a training set containing 10 points that follow a quadratic distribution with some random noise. (Left) A linear function suffers underfitting because it cannot capture the curvature of the data. (Middle) A quadratic function fits the data well and will be able to generalize to new points. (Right) Despite fitting the training points, the 9^{th} degree polynomial will not be able to generalize because it has captured the noise of the training set. Figure inspired by [8].

3.2 The Deep Learning revolution

Deep Learning (DL) is a subfield of Machine Learning which relies on neural networks[1] with many intermediate layers, each of which builds on the representations learned by the previous layer in order to automatically develop increasingly complex and useful representations of the data [30]. While shallow learning methods, i.e., methods with very few layers, rely on carefully handcrafted features to produce useful results, deep learning requires no feature engineering, as it can autonomously find what are the useful features, considerably simplifying machine learning workflows.

For instance, consider the case of an image that comes in the form of raw pixels. The features learned in the first layer would typically represent the existence of edges at certain locations and orientations, and the second layer would detect particular arrangements of the edges of the previous layer. Subsequent layers would assemble these patterns in larger combinations that are relevant for the chosen task, and the final layer would use these to make the predictions. The same result could not be achieved by stacking multiple shallow models, as they would be trained in succession, learning representations that would be good for each individual model but not the best for the overall goal [30]. On the contrary, the power of deep learning comes from the ability to train the multiple representational layers jointly [30].

In the last decade, DL methods have been very successful in a wide range of tasks. The first widely recognized breakthrough result was achieved in the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [31] using AlexNet [32], a deep convolutional network with eight layers. This network significantly increased the top-5 accuracy of the image classification challenge, which had

[1]In this work, the term “neural network” is used to refer to artificial neural networks unless stated otherwise.

been dominated by classical computer vision[2] algorithms, from 74.3% to 83.6%. This astounding result revived the interest in deep neural networks, which have since become the dominant approach for image recognition and detection tasks [9].

Despite the fact that by the 1990s the core ideas behind deep learning were already well understood, it took until the 2010s for deep learning to take off. This can be explained by two main factors: the improvement of hardware and the abundance of large datasets and benchmarks. Since deep learning is fundamentally guided by experimental findings, only when appropriate data and hardware became available could new ideas be tried, leading to improved techniques that resulted in advancements in the field. [30]

In 20 years, commercial Central Processing Units (CPUs) became about 5000 times faster, which allows to run on current laptops small deep learning models that would be impossible to run even in the most powerful computers 25 years ago. Yet, the most important improvement was brought from the massification of Graphical Processing Units (GPUs), developed to power the graphics of video games. In 2007, NVIDIA launched CUDA [33], which creates a programming interface for its line of GPUs, and soon some researches started to write CUDA implementations of neural networks, taking advantage of the possibility of paralellizing matrix multiplications, which are the main operation of neural networks. This increase of the computational power allowed to experiment with much more complex algorithms, as modern GPUs can easily achieve computational power on the order of $\mathcal{O}(10^{12})$ floating point operations per second. [30]

With the exponential progress in storage hardware and, most notably, the popularization of the Internet, it is very easy to collect and distribute large datasets for machine learning. For instance, Wikipedia articles are a key source of data for natural language processing models. In terms of computer vision, the most important dataset has been ImageNet [34], which contains more than 1.2 million images, each hand-labelled with one of 1000 categories. From 2010 to 2017, a yearly competition using the ImageNet dataset, the aforementioned ILSVRC [31], pushed competing teams against each other, resulting in rapid advances in DL for computer vision.

Since 2012, deep neural networks have dominated ILSVRC, achieving a 96.4% top-5 accuracy by 2015. At the same time, deep learning also emerged as a powerful tool in other areas, achieving breakthroughs in tasks such as speech transcription [35], machine translation [36], speech synthesis [37], image generation [38] and protein structure prediction [39].

Deep learning has also gained traction in the field of physics, particularly in particle physics and gravitational wave science. In particle physics, deep learning was shown to beat previous methods for both event selection and jet tagging at the Large Hadron Collider [40]. In the GW domain, deep networks are very promising for glitch classification, signal denoising, gravitational-wave signal searches and parameter estimation. A review of the use of machine learning for GW-related tasks can be found in [41].

3.3 Simple Neural Networks

After introducing deep learning and its great potential, it is important to understand its cornerstone, artificial neural networks. The field of Neural Networks (NNs) started with the investigations of McCulloch and Pitts, who in 1943 created a simple model of a biological neuron [42]. The model accepted one or more binary inputs and produced one binary output by activating the output if more than a certain number of inputs were active and keeping the output inactivated otherwise. This very simple model was able to compute logical propositions such as "and" or "or".

In the late 1950s, Rosenblatt [43] further developed this model by introducing real-valued weights, which allowed to attribute a different importance to each input. The resulting model, depicted in Fig. 3.2, became known as the Threshold Logic Unit (TLU). In order to produce the output, first the neuron computes the weighted sum of the inputs, $z = \Sigma_{i=1}^{n} w_i x_i = \boldsymbol{x}^\intercal \boldsymbol{w}$, and then applies a step function to the result, yielding

$$h_w(\boldsymbol{x}) = \text{step}(z) = \text{step}(\boldsymbol{x}^\intercal \boldsymbol{w}). \tag{3.1}$$

The most common step function used was the Heaviside step function, which returns 0 if the argument

[2]Computer vision is a sub-field of artificial intelligence that tries to derive meaningful representations from visual data, tackling the tasks of image classification, image segmentation, and object detection.

is lower than 0 and 1 otherwise:

$$H(z) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases} . \tag{3.2}$$

A single TLU can be used for binary classification, as it computes a linear combination of the inputs and, if the result exceeds a threshold, outputs 1, which corresponds to the positive class; otherwise, it returns 0, the negative class. The behaviour of the neuron depends on the input weights, which must be tuned for the task, that is, they need to be trained.

Instead of relying on a single neuron, algorithms become more powerful when using multiple neurons combined in a network like Perceptron [43], which is a neural network composed of a layer of TLUs, each of which connected to all the inputs. In this architecture, each input is fed to passthrough neurons that act like the identity function, leaving it unchanged. Additionally, a bias neuron, which is a simple neuron that always outputs 1, is added to the inputs of all TLUs, allowing to, in essence, change the threshold of the TLUs. A Perceptron with $N_i = 2$ input neurons and $N_o = 3$ outputs is shown in Fig. 3.3. Since the inputs travel through the network in only one direction, from the input layer to the output layer, the Perceptron is an example of a feedforward neural network. Moreover, as the neurons of one layer are connected to all the neurons of the next layer, the Perceptron is a fully connected network.

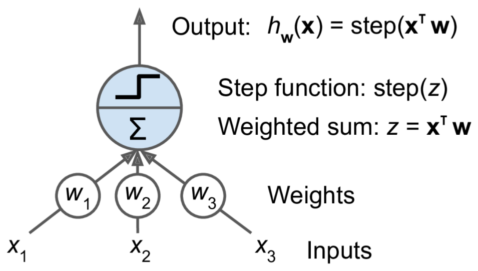
Output: $h_w(\mathbf{x})$ = step($\mathbf{x}^T$ $\mathbf{w}$)
Step function: step(z)
Weighted sum: z = $\mathbf{x}^T$ $\mathbf{w}$
w_1
w_2
w_3
Weights
x_1
x_2
x_3
Inputs

Figure 3.2: The threshold logic unit, a simple neuron model which is the building block of the Perceptron architecture. Figure reproduced from [44].

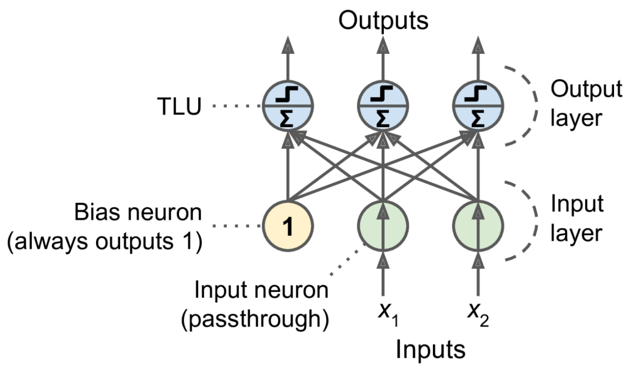

Figure 3.3: Architecture of a Perceptron with two inputs and three outputs. Figure reproduced from [44].

One important aspect of the neural network is that it is possible to efficiently compute its outputs for several instances at once, using the following equation [44]:

$$h_{\boldsymbol{W},\boldsymbol{b}} = \phi(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}), \tag{3.3}$$

where $\boldsymbol{X}$ is the matrix of data previously introduced. It is an m by n matrix, where each row is associated with a different example and each column corresponds to a distinct feature. $\boldsymbol{W}$ is the weight matrix containing all the connection weights except the ones from the bias neuron. It has one row per input neuron and one column per output neuron, resulting in a n by N_o shape[3]. $\boldsymbol{b}$ is the bias vector, which contains all the connection weights involving the bias neuron, thus having N_o elements.[4] Together, the weights and biases form what are called the model's parameters, that is, the values that control the model's behaviour. Finally, ϕ is an activation function, which is the Heaviside function in the case of the Perceptron.

It was shown in [45] that Perceptrons are incapable of solving some trivial but important problems, like replicating the “exclusive or” (XOR) logical function, but the authors also showed that these limitations could be eliminated by stacking multiple TLU layers, which results in a architecture called the MultiLayer Perceptron (MLP). An MLP contains one passthrough input layer, followed by one or more intermediate layers of TLUs called hidden layers, and one final TLU layer, the output layer.

[3]The number of input neurons, N_i, is set to match the number of features, n, so $n = N_i$.
[4]Note that $\boldsymbol{b}$ is summed to the $\boldsymbol{X}\boldsymbol{W}$ matrix using broadcasting, i.e., adding the vector to each matrix row.

Every layer except the output layer contains one bias neuron, connected to all neurons of the next layer. Fig. 3.4 shows a MLP based on the Perceptron of Fig. 3.3, but with the addition of a hidden layer containing four neurons, which introduces additional non-linearity to the model.

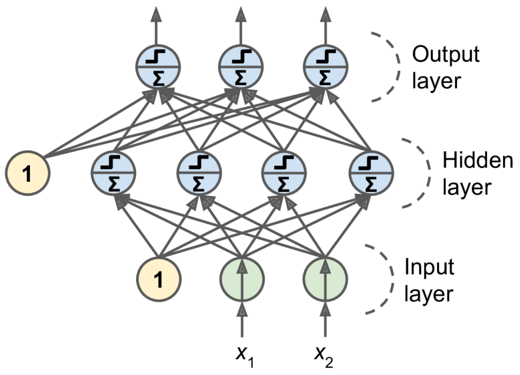

Figure 3.4: Architecture of a Multilayer Perceptron with two inputs, a hidden layer with four units, and three output neurons. Figure reproduced from [44].

In this network, the outputs of the l^{th} layer can be calculated similarly to equation (3.3):

$$\boldsymbol{h}^{l} = \phi(\boldsymbol{h}^{l-1}\boldsymbol{W}^{l} + \boldsymbol{b}^{l}), \tag{3.4}$$

where superscripts denote the layer number, following the same conventions as equation (3.3). Note that $\boldsymbol{h}^0$, the output of the input layer, is not calculated using this equation, as it corresponds to the input data matrix, $\boldsymbol{X}$. Using this equation, the output of a feedforward network can easily be calculated by sequentially computing the output of each layer from the input layer to the output layer.

The use of a non-linear activation function is of utmost importance for neural networks because it allows the network to apply a range of non-linear transformations to the data. Since the rest of the network only applies linear transformations, if there was not a non-linear activation function, the network would only be able to apply linear transformations even when using a large number of hidden layers[5]. Thus, the additional layers would not be useful, as the resulting composite function would be linear, and thus, could be achieved using a single layer of neurons.

Therefore, it is the non-linear activation function that gives the representational power to NNs. In fact, the universal approximation theorem [46] states that a feedforward network with at least one hidden layer and a squashing activation function (one whose outputs are in a closed interval) can, in theory, approximate a desired function with arbitrarily high precision.

The simple network architecture of an MLP can be used for multi-label classification problems, i.e., when each example belongs to one or more classes, by replacing the activation function of the last layer with the sigmoid function, which returns a real value between 0 and 1. Applying this change, the output of each neuron is a continuous value between 0 and 1, which can be interpreted as a probability. Thus, the network would need one output neuron per class, and, for each sample, it would return the probability of the sample belonging to each class.

The aforementioned sigmoid function is defined by

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{3.5}$$

This function, similarly to the Heaviside step function, squashes the output between 0 and 1. A noteworthy difference is that, unlike the step function, it is differentiable across the whole domain, which is useful for the training procedures described in section 3.4. A comparison between the Heaviside and sigmoid functions is depicted in Fig. 3.5

The sigmoid activation function can also be used in the particular case of binary classification by having just one neuron which outputs the probability of belonging to one of the classes. However,

[5] Chaining two linear functions yields another linear function. For two linear functions $f(x) = ax+b$ and $g(x) = cx+d$, the composite function is also a linear function: $f(g(x)) = (ac)x + (ad + b)$. It can be proven by induction that the same will happen for n chained linear functions.

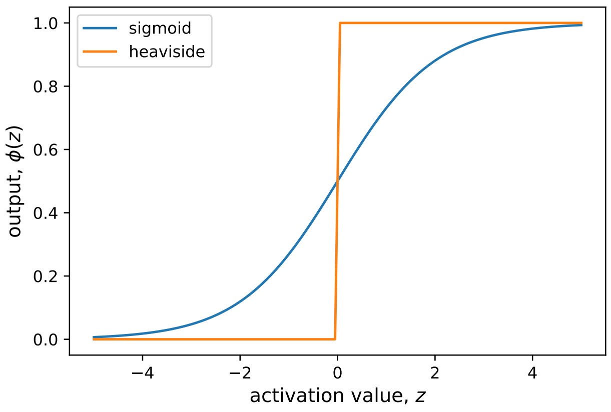

Figure 3.5: The Heaviside and sigmoid step functions. The sigmoid resembles a smoothed version of the Heaviside.

---

for multi-class classification tasks, where there are more than two classes and each sample belongs to one and only one class, another activation function, the softmax, is used in the last layer. Then, the classifier's prediction is the class with the highest estimated probability, which corresponds to the one with the highest score. The softmax function, which is a generalization of the sigmoid for more than two classes, takes a vector of scores (the activations of the output neurons), $\boldsymbol{z}$, computes the exponential of every score and normalizes them in order to estimate the probability of each class. Thus, the probability that the instance belongs to class k is

$$\hat{p}_k = \frac{\exp(\boldsymbol{z}_k)}{\Sigma_{j=1}^{K} \exp(\boldsymbol{z}_j)}, \tag{3.6}$$

where K is the total number of classes. Due to the normalization, the sum of all $\hat{p}_k$ is 1.

3.4 Training Neural Networks

It was previously stated that neural networks, per universal approximation theorem, can approximate any desired function, so the only issue is how to tweak them in practice to find good network parameters. In ML, this task of gradually adjusting the parameters so that performance gets better and better is known as training the model. This is not a trivial task, and it held back the field of NNs for many years.

The process of training a ML algorithm is depicted in Fig. 3.6. At the beginning, one starts with a chosen architecture, which is the term for the backbone of a model (for instance, a MLP with a specified number of neurons and layers), and initializes the parameters. During training, the model, composed of the architecture and parameters, predicts the labels for the input data. Then, the parameters of the model (in the MLP case, they correspond to the weight and bias of each neuron connection) are updated according to the loss function, which is a mathematical function that evaluates the mismatch between the labels the model predicts and the true labels. This loop is then repeated as long as wanted.

Note that after a model is trained, it is very easy to use it, as it behaves like a traditional computer program, as shown in Fig. 3.7. Thus, the sheer amount of computation is spent during training, and processing new data after the algorithm is fully trained can be done very quickly.

3.4.1 Loss function

As mentioned above, the loss function is important for the training loop. This loss function, $\mathcal{L}$, returns a real number, which is small when the performance of the algorithm is good. Furthermore, to enable the use of the gradient algorithms introduced in section 3.4.2, it must be differentiable with

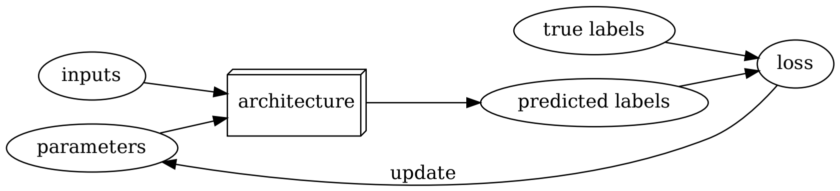

Figure 3.6: Training loop of a supervised ML algorithm. Figure inspired by [47].

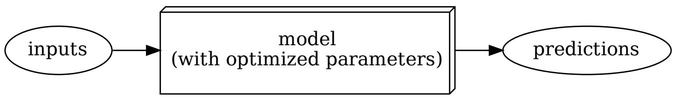

Figure 3.7: Using a trained ML model for inference is very straightforward. Figure inspired by [47].

respect to the model parameters. Thus, a simple loss function such as the number of incorrectly classified samples cannot be used as it is not smooth, since making small changes to a parameter is not likely to change the classification of any sample and, consequently, the value of the loss.

The problem of the lack of differentiability can be addressed by using instead the mean squared error, which is defined as

$$\mathcal{L}(\boldsymbol{x}) = \frac{1}{n}\sum_{x} \|\hat{y} - y\|^2, \tag{3.7}$$

where $\hat{y}$ is the model prediction for the given example, y the true label, and the sum is computed over n examples. Note that the function depends on the model parameters, since they control the predictions of the model. This function has the property of being non-negative, and becoming closer to 0 as the predictions become more similar to the target labels. Thus, the optimization goal is to minimize this loss function.

3.4.2 Gradient descent

But how does one update the parameters to increase the model's performance? The training of modern ML algorithms is based on gradient-based optimization, specifically using the gradient descent algorithms and similar variants. These are generic optimization algorithms, which tweak parameters iteratively in order to minimize a cost function.

At the beginning, all the network's trainable parameters are randomly initialized. Then, for each data sample, the model is used to predict its label. At this point, the algorithm needs to decide in which direction to update each weight. A simple approach would be, for each weight, to slightly increase it, and then decrease it, while keeping the other parameters frozen, in order to find what direction lowers the loss function. This would be highly inefficient, as two forward passes through the network would be needed for each single parameter.

Gradient descent, on the other hand, allows to minimize the loss function $\mathcal{L}(\boldsymbol{\theta})$, parameterized by the model's parameters $\boldsymbol{\theta}$, without having to compute the function for the whole domain, by updating the parameters in the opposite direction of the gradient of the loss function with respect to the parameters. It takes advantage of the differentiability of the functions used in the model[6], which when chained also form a differentiable function. Thus, the function which maps the model parameters to the loss of the model given an input is differentiable, which allows to compute the gradient of the loss with respect to each parameter.

[6]The neural networks described in section 3.3 are not differentiable because of the use of the step function, which has no derivative at $z = 0$ and null derivative in the rest of the domain. Modern NNs use differentiable activation functions

Since the loss function may not be smooth, the gradient can change considerably between close points, so the magnitude of the gradient is multiplied by a small positive constant, the so-called learning rate, one of the most important model hyperparameters.[7]. The gradient of a function at a certain point gives the direction of steepest ascent of a function around that point, so to minimize the loss function one needs to move against the direction of the gradient. The following equation describes how to update a parameter using gradient descent [8]:

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}), \tag{3.8}$$

where $\boldsymbol{\theta}$ is the parameter vector, α is the learning rate, and $\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$ is the gradient of the loss function at point $\boldsymbol{\theta}$. Note that the gradient is a vector which contains all the partial derivatives, with element i of the gradient corresponding to the partial derivative of $\mathcal{L}(\boldsymbol{\theta})$ with respect to θ_i.

The gradient descent algorithm is guaranteed to converge if an adequate learning rate is chosen, but it may find a local minimum instead of the desired global minimum [8]. Nevertheless, gradient descent has been used in practice with great success, and the convergence to local minima appears not to be a common problem, as the algorithm often finds a very low value of the loss function fast enough to be useful [8].

Normally, this iterative process is repeated for a pre-defined number of forward passes through all the training data, known as epochs, but it can be stopped earlier depending on the behaviour of the loss function or a performance metric. Note that the number of epochs, `n_epochs`, is an important hyperparameter. The whole process of gradient descent is summarized in Fig. 3.8.

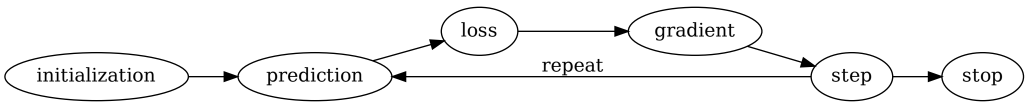

Figure 3.8: Overview of the gradient descent algorithm for the optimization of parameters. Figure inspired by [47].

Depending of the number of data points used for each iterative step, gradient descent can be classified in three types: batch gradient descent, stochastic gradient descent and mini-batch (stochastic) gradient descent. In batch gradient descent, the whole dataset is used for each parameter update. This gives the most accurate gradients at each step, as the whole dataset is used for the calculation, at the cost of being too slow for large training sets. Stochastic gradient descent, on the other hand, uses only one sample per iterative step to compute the gradients. This is much faster, but since only one sample is used to estimate the loss function, the behaviour is much more uncertain, and the algorithm may not converge. Mini-batch gradient descent is at a halfway point between the two previous algorithms, as it computes the gradients on small random subsets of the training data. This allows to increase the accuracy of the estimation while not running out of memory and taking advantage of the GPUs' parallelization capabilities [44]. Therefore, it is the most widely used in practice. The number of samples per mini-batch is another very important hyperparameter, known as the batch size, `bs`.

Mathematically, the gradient estimation using mini-batch gradient descent can be expressed as [8]:

$$\textbf{grad} = \frac{1}{\texttt{bs}} \sum_{i=1}^{\texttt{bs}} \nabla_{\boldsymbol{\theta}} \mathcal{L}(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}), \tag{3.9}$$

where f represents the transformation that maps the inputs to the outputs using the parameters $\boldsymbol{\theta}$, and $\boldsymbol{x}^{(i)}$ and $\boldsymbol{y}^{(i)}$ represent, respectively, the i^{th} example of the dataset and the associated target. Note that one gradient is calculated for each example, and the final gradient estimation is a simple average of those gradients.

in order to allow the use of gradient based optimizations.

[7]An hyperparameter is a parameter which does not affect the model's behaviour, but changes the learning process itself. Hyperparameters are discussed in section 3.8.

Gradient descent can be further accelerated with the introduction of momentum [48], which is analogous to velocity in a physics setting. If one imagines a ball rolling down a hill with the shape of the loss curve, it will not always follow the direction of the gradient, due to the velocity acquired so far (resulting from past acceleration). If the ball has enough momentum, it will not get stuck at local minima, and is more likely to reach the base of the hill, i.e., the global minimum. In practice, using momentum means including a moving average of the parameters' values, and including them in the calculation of the next parameters.

3.4.3 Backpropagation

In the explanation of the gradient descent algorithm, it was implicit that there is a way to calculate the gradient of the loss of a given example with respect to each parameter. However, only in the 1980s an effective algorithm to train MLPs was popularized, after a paper [49] used backpropagation to successfully train neural networks. This algorithm manages to compute the gradient of the loss function using only one backward pass through the network by calculating the gradient for each layer based on the immediately deeper layer, using the chain rule of calculus.

Backpropagation is a very efficient method, as it only requires a backward pass through the network in addition to the forward pass, and they have approximately the same complexity. It works on two assumptions: that the loss function can be written as a average over loss functions for individual examples, and that the loss function can be written as an function of the model's outputs. Fortunately, both hold for NNs and the loss functions used in practice. [50]

3.5 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a subset of feedforward deep networks, inspired by the study of the brain's visual cortex, and have been used with success for image-related tasks since the 1980s [51,52]. They have re-gained popularity in the last decade and have been the cornerstone of deep learning applications in the vision domain, as all the editions of ILSVRC from 2012 to 2017 were won by different CNN architectures. The distinctive feature of such networks is the employment of convolution operations, instead of regular matrix multiplications. CNNs have at least one convolutional layer, where neurons are optimized to detect specific image features. Typically, they have more than one such layer, with the first layers serving as detectors of small-scale low-level features such as edges, and the deeper convolutional layers detecting larger high-level features.

In a convolutional layer, the neurons share weights within a kernel filter, so rather than learning a separate set of parameters for every different input location, the model only learns a single set of parameters, the filter weights, which allows to detect similar features in different locations of the image. In the convolution operation, the kernel is slid across the input, multiplying each element of the kernel by an element of the input and then adding together the products. The sharing of the weights greatly reduces the number of parameters to be learned, thus reducing the memory requirements of the models.

Moreover, unlike traditional fully-connected networks, CNNs often have sparse interactions, because the kernel size is much lower than the size of the input. This means that each neuron is only connected to a small rectangle[8] of the previous layer of neurons (or the image, in the case of the first layer). This sparsity allows the model to compute the output with much less operations than a classic neural network. [8]

Typically, each layer has many different filters, in order to detect multiple features. In a regular 2D convolution, each filter is slid over three dimensions (width, length and depth/number of channels), producing one two-dimensional (width $\times$ length) distinct feature map for each filter. These feature maps are then stacked over the depth dimension, as shown in Fig. 3.9. Thus, by controlling the number of filters, it is possible to control the depth at each layer. Similarly, changing the filter size and stride (the step size of the moving window) allows to alter the width and the length.

Generally, at the end of a convolutional layer, in addition to a non-linear activation function, there is a pooling layer that acts as a downsampling operator, allowing to reduce the size of the output

[8]Unlike fully-connected networks, where neurons in the same layer are organized in a long line, in CNNs the neurons are better represented in a rectangle, to better match each neuron with the corresponding input.

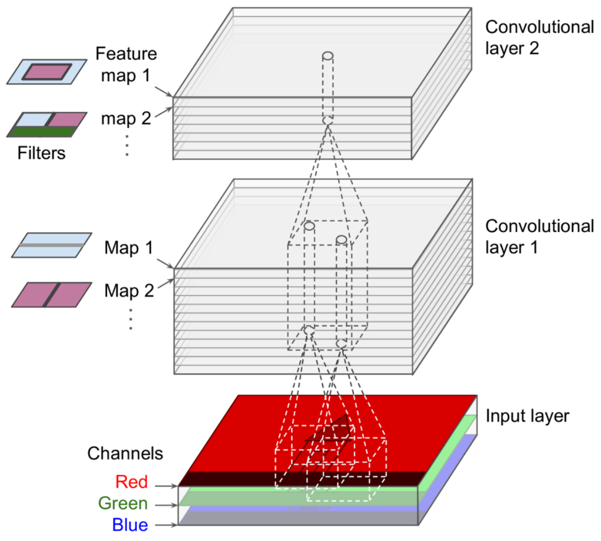

Figure 3.9: Two convolutional layers applied to a three-channel (RGB) image input. Each filter detects a different feature and produces a two-dimensional feature map. The multiple feature maps are then stacked. Figure reproduced from [44].

---

while preserving the depth. For example, the popular max pooling operation returns the maximum output value within a moving rectangular window for each feature map. Pooling helps to make the representations approximately invariant to small translations of the input, since small shifts in the input positions will not change the pooled outputs, allowing the convolutional layers that follow to still find the features even if they are not in the exact expected positions. [8]

3.5.1 Residual Neural Networks

One of the most important algorithmic advances in DL for computer vision was arguably the introduction of residual connections, also known as skip connections, by He et al. in 2015 [53], which allowed to successfully train the first truly deep convolutional neural networks, with tens to hundreds of layers. Their model, which was called a Residual Neural Network (ResNet), won the 2015 edition of ILSVRC. Since then, skip connections have been ubiquitous on modern convolutional networks, like Inception v4 [54], EfficientNet [55] and the novel ConvNeXt [56].

ResNets were introduced to solve the problem of training deep CNNs, as it was observed that the training error of the networks would saturate for a certain number of layers and even decrease as more layers were increased (see e.g. [57]). This is counter-intuitive, since the result should be at least the same if the additional layers would simply be identity mappings. Residual connections solve this problem by skipping over layers, that is, creating a direct path between non-adjacent layers, as seen in Fig. 3.10. This way, the network only needs to learn the residuals, i.e., the slight differences between doing nothing and passing through the skipped convolutional layers [47].

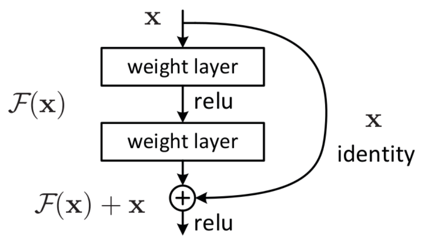

Figure 3.10: Simple ResNet block with a residual connection. Figure reproduced from [53].

It has been reported that using skip connections helps to smooth the loss function, which makes training easier [58]. Moreover, the authors of [59] argue that the benefit of residual connections is providing many short paths through the network, which contribute the most to the gradients and are easier to train than the longer paths which use less skip connections.

3.6 The Vanishing Gradients Problem

In section 3.2, better hardware and more data were deemed essential for the rise of deep learning. However, these improvements by themselves were not enough to train deep neural networks, as researchers faced the problem of unstable gradients, which would either fade away or become bigger and bigger as the number of layers increased, resulting in either null or enormous parameter updates and thus hindering the training of the NNs. This issue, known as the vanishing/exploding gradients problem, was alleviated with the improvement of algorithms, mainly through the choice of better loss functions, activation functions, weight-initialization schemes, and optimizers.

Mean squared error (introduced in section 3.4.1) was a popular loss function in the 1980s and 1990s, but the models suffered from saturation of neurons and slow learning [8]. This issue was addressed with the use of the cross-entropy loss function, which for the case of a single example and K neurons in the output layer is given by [44]

$$\mathcal{L}(\boldsymbol{\theta}) = -\sum_{k=1}^{K} y_k \log(\hat{p}_k), \tag{3.10}$$

where y_k is the target probability of class k, typically either equal to 0 or 1, and $\hat{p}_k$ is the probability that the example belongs to class k, computed for instance using equation (3.6). In contrast with the mean squared error, cross-entropy has the handy property of, when used in conjunction with sigmoid activations, resulting in bigger gradient updates when the mismatch between predictions and targets is higher, and slower learning when the probabilities are close to each other [50].

The sigmoid was the most used activation function in the hidden layers for many years, but it was shown by Glorot and Bengio [60] that this function was not appropriate for deep networks because neurons would easily saturate. As inputs become large in modulus, the function saturates either at 0 or 1, with an almost null derivative, which makes the gradient very small and hinders training. Gradually, it was replaced by an activation function which does not saturate for positive values, the Rectified Linear Unit (ReLU) function [61], defined as $\text{ReLU}(z) = \max(0, z)$. Despite not having derivative at $z = 0$ and null derivative for $z < 0$, it is very easy to compute and it has proven to work very well in practice [44].

The same paper [60] also raised the point that the unstable gradient problem could be alleviated if the signal travelling the network could flow properly in the forward and backward directions. For this to happen, the variance of the outputs of a layer would need to be equal to the variance of its inputs and the gradients should keep their variance after traversing a layer in the reverse direction. They showed that this was not addressed by the random parameter initialization techniques used at the time and, despite the impossibility to fulfill the two requirements simultaneously, Glorot and Bengio proposed a good compromise that works well in practice, known as Glorot initialization. Using this technique, the initial value of each weight from layer j is sampled from the following distribution:

$$W \sim U\left[-\sqrt{\frac{3}{\text{fan}_{\text{avg}}}}, \sqrt{\frac{3}{\text{fan}_{\text{avg}}}}\right], \tag{3.11}$$

where U denotes an uniform distribution, n_j is the number of neurons in layer j and fan_{avg} is the average between the number of inputs and the number of neurons of the layer. As Glorot initialization was proposed considering sigmoid functions, this method was subsequently adapted by He et al. in [62] to account for ReLU (and alike) activation functions by scaling the limits of the distribution by $\sqrt{2}$, which became known as He initialization.

Another very important improvement was the introduction of the batch normalization technique [63], which introduces an operation that zero-centers and normalizes the output of a layer, and then scales and shifts the result using two tunable parameters. The “batch” in its name comes from the fact that at training time, the mean and variance of the inputs is computed over the mini-batch[9]. It was found that, apart from avoiding unstable gradients during training[10], reducing the dependence on the activation functions and weight initialization, it increases the speed of training, as it allows the use of higher learning rates. Finally, it was found empirically that batch normalization acts as a regularizer, i.e., avoids overfitting and allows the model to generalize better [63].

Deep learning also benefited from the development of better optimizers, the most popular being Adam [64], which is inspired in the momentum approach. Adam uses an exponentially decaying average of past gradients as a direction and divides it by the decaying average of past squared gradients to yield an adaptive learning rate to each parameter [47].

3.7 Model evaluation

In section 3.1, it was stated that after a model has been trained, using the training set, its performance is evaluated on the test set. But how can two different models be compared? The simplest approach would be to directly compare the test loss of both models. It is true that the loss function is a measure of performance, but it mostly exists to allow for the automatic update of weights during training and usually is not very meaningful to humans, at least in classification problems. Thus, it is better to rely on metrics, which are functions that measure the quality of the model's predictions, but in a human-friendly way that does not usually possess the properties needed by a loss function.

To better introduce the most common metrics, it is useful to visualize the model results in a confusion matrix, which is a table that summarizes the number of times an example of each class was classified as each of the classes. The structure of the confusion matrix for a binary classification can be seen in Table 3.1. Table 3.2 shows an example confusion matrix filled with hypothetical values, which will be used to compare the different metrics. The confusion matrix can be extended to multi-class classification problems by having one column and one row for each class, and it easily allows to find the most confused classes (hence its name).

Table 3.1: Structure of the confusion matrix for a binary classification problem.

		Predicted	
		Positive	Negative
Actual	Positive	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

Table 3.2: Example of a confusion matrix for a binary classification problem.

		Predicted	
		Positive	Negative
Actual	Positive	3	5
	Negative	0	92

The most straight-forward metric for classification problems is accuracy, which is given by the number of correctly classified examples over the total number of examples. Using the confusion matrix nomenclature, accuracy is computed using

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}. \tag{3.12}$$

In multi-class problems, accuracy is computed by dividing the number of samples in the main diagonal of the confusion matrix by the total number of samples.

Accuracy is a useful metric when the dataset is balanced and all classes are equally important. However, in a case such as the one in Table 3.2, accuracy can return a over-optimistic value. In fact, despite misclassifying most of the samples from the negative class, the model would have a 95% accuracy.

Instead of accuracy, precision and recall are often used together for such problems. Precision is defined as the ratio of correct positive predictions to the actual number of positive examples, that is:

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}. \tag{3.13}$$

[9]Since this is not feasible at test time, the statistics are then estimated using a moving average.

[10]Note that the enhanced activation functions and weight initialization schemes mostly address the problem of unstable gradients at the beginning of training, but do not guarantee the problem will not appear later.

On the other hand, recall is the ratio between the number of correct positive predictions and the total number of positive examples:

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{3.14}$$

Despite being defined for the binary class, precision and recall can be used in multi-class problems by computing these two metrics for each class. To do so, the selected class is considered as the positive class and all the other classes are the negative class.

In the case shown in Table 3.2, the precision is 100% but the recall is a mere 37.5%, which is more indicative of the true performance of the model. However, relying on precision and recall introduces the problem of having two metrics to compare models, instead of a single number. This is solved by the so-called F1 score, which is the harmonic mean of precision and recall:

$$\text{F1 score} = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = \frac{\text{TP}}{\text{TP} + \frac{\text{FN}+\text{FP}}{2}}. \tag{3.15}$$

The choice of the harmonic mean gives more weight to low values and, consequently, a high F1 score can only be achieved if both precision and recall are high. For instance, in the presented example the F1 score is only 54.5% despite the perfect precision.

Similarly to precision and recall, the F1 score can also be computed for each class in multi-class problems. Nevertheless, once again the issue of having multiple metrics appears, as there will be one metric for each class. This is solved by computing the macro-averaged F1 score instead. This metric is calculated by plugging in the macro-averaged precision and macro-averaged recall, which are respectively the precision and recall averaged across all classes, in equation (3.15).

3.8 Hyperparameter optimization

Most machine learning algorithms have hyperparameters, which are not the specific parameters of the algorithm (the weights and biases in the case of NNs), but the settings that change the behaviour of the algorithm itself (e.g. the number of layers in a NN) or its training (for instance, the number of samples in each mini-batch during training).

3.8.1 The importance of hyperparameters

One of the most important hyperparameters when using gradient-based optimization methods is the learning rate, which is the small number, α, that is multiplied by the gradient in equation (3.8). The choice of a good learning rate is of utmost importance to allow the algorithm to converge in reasonable time. In fact, if the learning rate is too small, the gradient descent steps can become very small, increasing the time until training converges to a minimum. On the other hand, if the learning rate is too high, the gradient descent can overshoot the minimum, bouncing around it or even diverging away from it. [47]

As mentioned before, another important hyperparameter is the batch size. Larger batch sizes ensure more accurate gradient estimates and take advantage of modern hardware parallelization capabilities, while needing more memory. In contrast, lower batch sizes will yield longer training times, but often contribute to regularize the model, achieving lower generalization errors. Typical batch sizes try to balance the aforementioned advantages and disadvantages, using powers of 2 between 32 and 256[11]. [8]

The number of epochs is also important, as too few epochs will cost less time but not allow the model to learn enough from the data and too many epochs will take longer and run the risk of overfitting [8]. Considering the variety of different training enhancement algorithms that have been developed, there are many more hyperparameters related to training, such as the choice of the optimizer (like Adam) and the tunable parameters of the optimizer itself, the various learning rate scheduling options (i.e., algorithms that change the learning rate during training), and so on.

There are also hyperparameters related to the model architecture, such as the number of layers or the number of neurons in each layer. Nowadays, an already-proven fixed architecture is used in

[11]Powers of 2 were initially used because of the better runtimes due to the architecture of the available hardware, but nowadays are used mostly due to historical reasons. See for example this blog post.

most cases, so they are typically not optimized, albeit different models can be tried and the choice of architecture can be seen as another hyperparameter.

3.8.2 Validation dataset

The optimization of the hyperparameters cannot be done using only the training dataset, as sometimes a choice can lead to worse fits in the training set but lower generalization error. If it is instead evaluated on the test dataset, there is a fundamental problem if one follows the iterative approach of training the model in the training set, estimating the generalization error using the test set, and tweaking the hyperparameters accordingly: the more iterations are done, the sooner the model will be overfitted to the test data, yielding overly-confident estimations of the generalization error.

Thus, a new dataset is introduced: the validation set, which serves the purpose of evaluating the model during each training iteration. With this inclusion, the test set is used only once, at the end of training, to estimate the generalization error of the algorithm. Using the improved approach with three datasets, the estimation of the generalization error will be much more accurate. To avoid biasing this estimation, no exploratory data analysis or statistics should be computed over this dataset prior to training. In ML practice, a good rule of thumb is to randomly split all the available data into three subsets: the training set, the validation set, and the test set, for instance using a 70/15/15 split [65].

3.8.3 Hyperparameter optimization techniques

The value of a single model hyperparameter can make the difference between an average model and a state of the art result. In modern machine learning, hyperparameter tuning is still a engineering discipline, with some guidelines but without formal rules. As the hyperparameter space is often formed by discrete decisions, it is not differentiable, meaning that the powerful gradient descent techniques are not applicable.

The most common hyperparameter optimization technique is a combination of manual search and grid search, where a few hyperparameters as well as their possible values are chosen manually relying on the experimenter's expertise in the task and the algorithms. Then, each combination of values for all the chosen hyperparameters is tested and the results compared at the end. However, due to the so-called curse of dimensionality, this technique is not feasible when the number of hyperparameters to be tested increases, because the number of possible configurations grows exponentially.

As typical problems have much fewer relevant dimensions (unknown *a priori*) than the number of hyperparameters, grid search wastes many trials in the irrelevant dimensions. A good alternative is random search, in which the hyperparameters of each trial are chosen at random and independently from previous trials. Unlike grid search, this method has the same efficiency in the relevant parameter space as if it had been used only in the relevant dimensions, allowing to significantly reduce the number of necessary trials. In sum, random search has the simplicity of grid search but trades a small reduction of performance in low-dimensional hyperparameter spaces for a large increase in high-dimensional search spaces. [66]

One issue with both grid search and random search is that the trials are independent of each other, that is, the previous information is not used to enhance the search method. This can be addressed taking advantage of bayesian optimization algorithms, which use a simple surrogate model like a Gaussian kernel to model the metric to optimize in function of the hyperparameters, and iteratively choose the hyper-parameter set that performs best on the surrogate model, train a model with the chosen hyper-parameters and use the results to update the surrogate model. This bayesian approach has the advantage of using all the previous information to compute the next point to sample. Although it requires more time to decide each new set of hyperparameters, fewer iterations are needed to find good results, which is useful in cases where each new point is costly, like when training DL models. [67]

3.9 The `fastai` deep learning library

`fastai` [68] is a modern deep learning library, which is built on top of the very popular `PyTorch` library [69]. It combines `PyTorch`'s flexibility with high-level components that are easy to use, and allows to achieve state-of-the-art results by providing a high-level API that offers ready to use functions for domains such as vision, while using sensible default values.

For instance, the main training routine for models is a variant of the 1cycle policy [70], which consists of a hyperparameter scheduler with a cosine annealing that increases the learning rate up to a point and then decreases it, and does the inverse for the momentum. The default setting is to reach the maximum of the learning rate at 30% of the training. The behaviour of the learning rate and momentum hyperparameters is represented in Fig. 3.11.

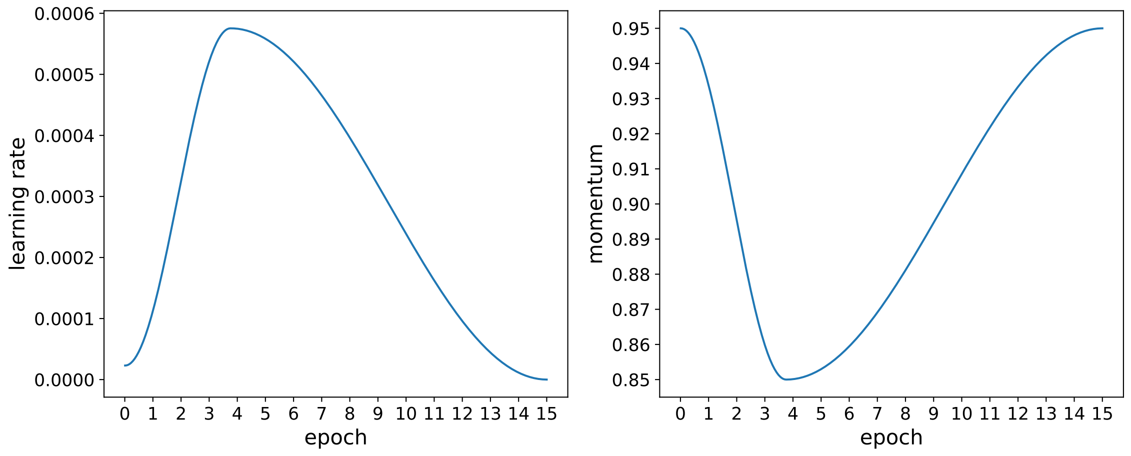

Figure 3.11: Evolution of the learning rate and momentum values when using `fastai`'s `fit_one_cycle` training routine, with a maximum learning rate of 5.75×10^{-4} and momentum between 0.85 and 0.95.

The use of a smaller learning rate at the beginning of training is not new, because it allows to warm-up the training and to make sure it does not diverge. Then, the use of higher learning rates during the middle of training acts like a regularizer, since if forces the model to stay in a smoother minimum of the loss function, thus avoiding overfitting. Finally, the descending learning rates in the last part of training allow to explore steeper local minimums inside the smoother part. Regarding the momentum, the point of the cycling momentum is mostly to make the training more robust to the large learning rates. This 1cycle was found to allow training at higher learning rates, without overfitting, and train neural networks an order of magnitude faster than with standard methods [71].

In `fastai`, the default optimizer is Adam, introduced in section 3.9, with a default maximum learning rate of 0.001, $\beta_2 = 0.999$ and β_1, which is the momentum, cycling between 0.95 and 0.85 when using the 1cycle policy. In addition, `fastai` introduces weight decay by default. It is a regularization technique, which penalizes the loss function by a value proportional to the sum of all squared weights. By benefiting lower weight values, the model avoids overly complex functions, resulting in better generalization [47].

Instead of using the default learning rate, the best practice is to choose an appropriate learning rate using the learning rate finder [70]. This tool is remarkably simple, as it consists of starting with a very small learning rate, passing one mini-batch through the model, computing the loss and repeating this process while gradually increasing the learning rate until the loss stops decreasing. Finally, the loss is plotted against the learning rate, as shown in Fig. 3.12, and there are a few methods to choose a good learning rate value based on this plot, using one of the following suggestion functions:

- `steep`, which chooses the learning rate where the slope of the loss is the steepest, i.e., where the model is learning faster;
- `minimum`, which returns a learning rate one-tenth of the minimum before the loss diverges;
- `valley`, which suggests the learning rate associated with the steepest slope through the longest valley in the plot;
- `slide`, which returns a learning rate following an internal slide rule.

Moreover, this library allows to easily take advantage of mixed precision training [72], a technique which reduces the memory requirements and speeds up training. In mixed precision training, the forward and back-propagation passes are computed in half precision (16-bit signed floats), in order

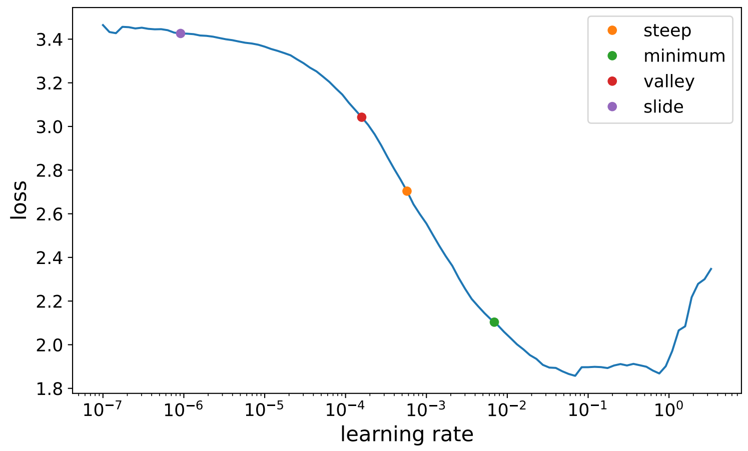

Figure 3.12: Result of a learning rate finder, with the points suggested by the four functions available in `fastai`.

---

to save memory and, in the case of NVIDIA's latest GPUs, reduce the training time. Then, the parameters are updated in single precision (32-bit signed floats), to correctly handle the common case where the gradient is several orders of magnitude below the respective parameter. If this step was instead performed using half-precision, the update would not change the parameter.

3.9.1 Transfer learning

Transfer learning [73] is a technique which allows to take advantage of knowledge learned in one domain, typically from a task that has many labelled examples, and to apply it to a different but related task which may have a limited number of examples. This technique is one of the cornerstones of the `fastai` library, as it allows to train more accurate models, more quickly, and using less data, as compared to training the same models from scratch [47]. Transfer learning for vision applications typically uses the weights of a model pretrained by experts on the aforementioned ImageNet dataset, and `fastai` is no exception.

The default `fastai`'s transfer learning approach is to, after assigning the pretrained weights, replace the final layer with a fully connected layer with the correct number of outputs, and initialize it with entirely random weights. As all the previous layers have meaningful weights, in the first epoch(s), known as a frozen epoch(s)[12], the pretrained layers are frozen, and only the final layer is trained using the first part of the 1cycle policy, with the learning rate monotonically increasing until it reaches the chosen learning rate. Then, the model is unfrozen and all layers are trained by the desired number of epochs, also using the 1cycle policy and with a maximum learning rate of half the one used in the previous step. However, as previously mentioned, the first layers of CNNs learn very simple features, which are probably useful for all tasks, while the final layers learn more domain-specific features. For this reason, it is not sensible to train all layers with the same learning rate, so `fastai` fine-tunes the models using discriminative learning rates, which simply means using lower learning rates for the early layers and higher learning rates for the later layers, which more likely need to be updated. By default, the learning rate of the first layer is 100 times smaller than the one of the last pretrained layer.

[12]There can be more than one frozen epoch, but this is `fastai`'s default value.

Chapter 4

Classification of LIGO data using Deep Learning

In this chapter, deep learning is applied to the classification of aLIGO data from the first two aLIGO and AdV observing runs (O1 and O2). In section 4.1, the dataset used is presented and, in section 4.2, a baseline model consisting of a residual NN is trained on this dataset. In sections 4.3 and 4.4, experiments are performed in order to find better models, respectively, by optimizing the hyperparameters of the baseline model and trying a transfer learning approach. Finally, the best models are evaluated on the test set in section 4.5 and used to classify real gravitational wave signals in section 4.6.

The architectures and pre-trained weights used in this section were retrieved from the `timm` library [74]. The code used to train the models and analyze the results can be accessed on GitHub and the raw results on Weights & Biases.

4.1 Dataset

The Gravity Spy dataset, introduced in [75], and accessible here, is a collection of spectrograms of (mostly) glitches identified during the first and second aLIGO and AdV observing runs. All the samples were hand-labelled by volunteers participating in the Gravity Spy project [76], which combines machine learning with citizen science in the task of categorizing all glitches recorded by the aLIGO (and Virgo) detectors. Samples were only included in the Gravity Spy dataset after the volunteers' classifications achieved a high level of agreement with experts and the project's machine learning classifier. This dataset presents a multi-class classification problem, where the goal is to assign the only correct glitch class to each noise sample.

Each glitch sample contains four spectrograms centered at the same instant in time but with different durations: 0.5, 1.0, 2.0 and 4.0 seconds, all represented as 140×170 pixels grey-scale images, as seen in Fig. 4.1. It can be seen that the glitch classes vary widely in duration, frequency range and shape. Nevertheless, some classes can be easily mistaken for each other due to similar morphology, like the `Blip` and `Tomte` classes, or if they are not visualized using the appropriate time window: for instance, a `Repeating_Blips` sample may be identified as a `Blip` if the window is too small.

In this version of the dataset, there are 8583 glitch samples distributed unevenly over 22 different classes, as seen in Table 4.1. The majority of examples are `Blips`, with almost 1900 samples, while there are five minority classes with less than 190 examples (10% of the number of `Blips`), the less represented being the `Paired_Doves`, with only 27 examples. The `Chirp` class, with only 66 examples, does not actually represent glitches but instead hardware injections, which are simulated signals created to resemble real gravitational waves and used for the testing and calibration of the detectors [75]. The dataset is already split, using stratified sampling to ensure similar distributions in each subset, in training (70%), validation (15%) and test (15%) sets, to facilitate the comparison of different methods.

4.2 Baseline classifier

In this work, the ResNet architecture [53], introduced in section 3.5.1, is used as a baseline for the classification task. In the original implementation, ResNets end with a fully connected layer with 1000 units, but in the case of the Gravity Spy dataset, there are 22 classes, so the last layer was replaced by a fully connected layer with 22 units, each corresponding to one of the classes.

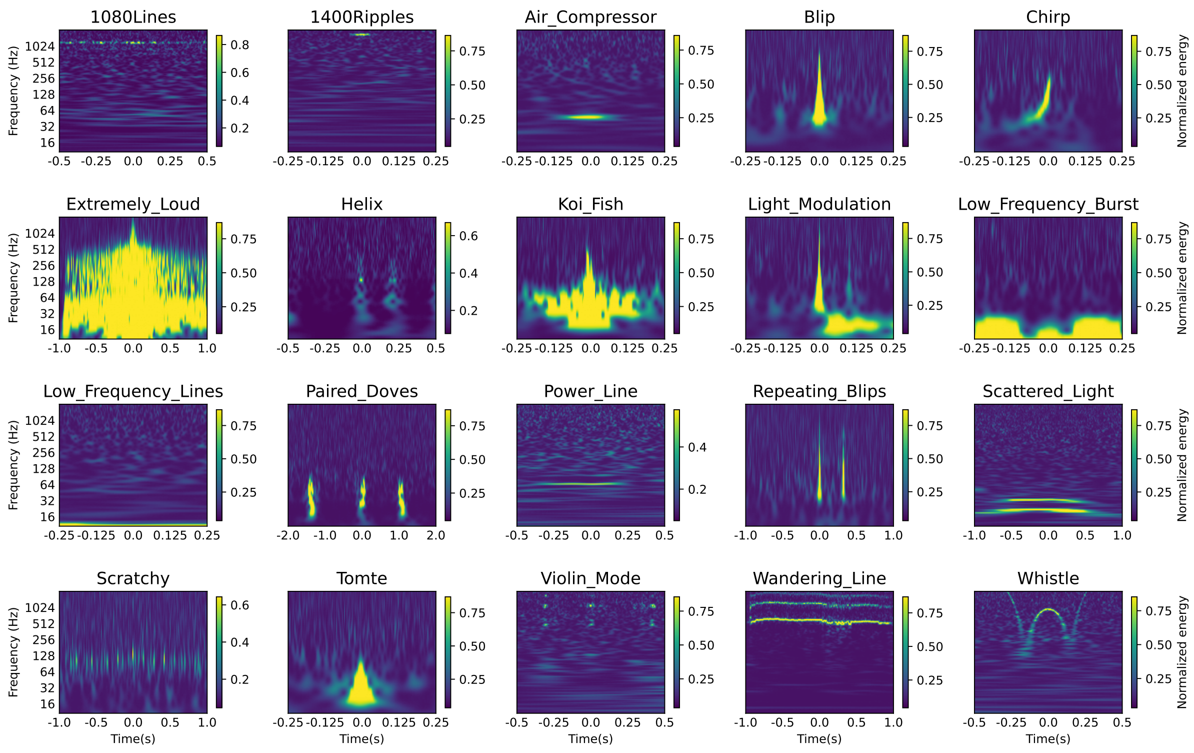

Figure 4.1: Examples of the different glitch classes. The classes `None_of_the_Above` and `No_Glitch` are not represented. The time-scale is not equal for all images, as some classes require longer periods to be identified. Despite being grey-scale images, they are shown with a colour map for better visualization.

Table 4.1: Different glitch classes in the Gravity Spy dataset, with the number of samples of each class indicated in the Total samples column. The dataset is clearly imbalanced, and classes with less than 10% of the samples of the most represented class are signaled in red.

No.	Class	Total samples	No.	Class	Total samples
0	1080 Lines	328	11	No Glitch	181
1	1400 Ripples	232	12	None of the Above	88
2	Air Compressor	58	13	Paired Doves	27
3	Blip	1869	14	Power Line	453
4	Chirp	66	15	Repeating Blips	285
5	Extremely Loud	454	16	Scattered Light	459
6	Helix	279	17	Scratchy	354
7	Koi Fish	830	18	Tomte	116
8	Light Modulation	573	19	Violin Mode	472
9	Low_Frequency Burst	657	20	Wandering Line	44
10	Low Frequency Lines	453	21	Whistle	305

For a quick search of the baseline classifier trained from scratch, the ResNet18 and ResNet34 architectures[1], with and adapted number of inputs for the first convolutional layer, and using random (He) weight initialization were chosen, training with fastai's `fit_one_cycle` routine over 10 epochs with the steepest point from the learning rate finder as the maximum learning rate, and the other hyperparameters set to fastai's defaults. The loss function chosen was the cross-entropy loss, which is tailored for multi-class classification.

Several approaches were tested regarding the amount and format of the information provided to

[1]The Resnet18 and Resnet34 architectures are briefly presented in appendix A.

the model. In the simplest approach, the model had only access to one of the four spectrograms for each example, always with the same duration. This corresponds to views single1 (0.5 s), single2 (1.0 s), single3 (2.0 s) and single4 (4.0s). Thus, these single-view models accepted batches of grey-scale (1-channel) 140 × 170 (height × width) pixels images. The second approach was a merged view model inspired in [77], which consists of placing the four single view 140 × 170 images next to each other, forming a 280 × 340 pixels image. Finally, an approach based on encoding information related to a different time duration in each image channel, as introduced in [78], was also implemented. Encoded views with all combinations of 2 to 4 time durations were compared.

Initially, it was evaluated whether mixed precision training, introduced in section 3.9, was useful for the GPU used on this work, NVIDIA's RTX A4000 [79]. Thus, a quick grid hyperparameter sweep was performed using the two architectures and only an example view of each combination of channel dimension and image size, both with and without mixed precision training. The models were trained for 10 epochs using the `steep` learning rate function. In total, 20 different models were trained, and the results in terms of the use of mixed precision are shown in Fig. 4.2. Mixed precision reduced the training time median by 29%, while making a difference of less than 0.2 percentage points in the median macro-averaged F1 score achieved in the validation set. Hence, mixed precision training was used for the rest of the models, as it saves time at virtually no cost in performance.

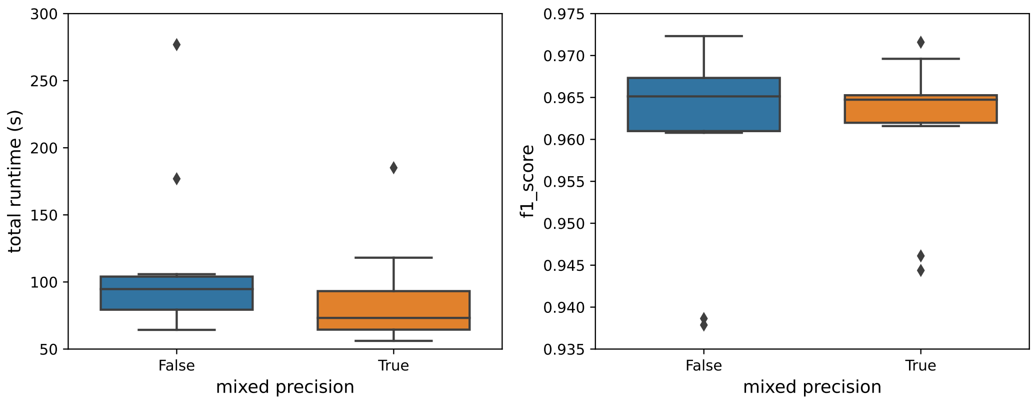

Figure 4.2: Box plots of the results for the grid search to study the effect of mixed precision training in terms of: (left) total training time, and (right) the F1 score in the validation set. "False" indicates normal double precision training was used, while "True" labels the usage of mixed precision. Mixed precision allows to decrease the training time while maintaining the performance of the models.

Moreover, it was also found that most models which use mixed precision took between 6.4 and 9.3 seconds per epoch during training. As models that train faster are easier to optimize and can also result in faster inference times (if the lower training time is a result of a simpler architecture and not a reduced number of epochs), a new metric was created to penalize models that take longer to train:

$$\text{combined\_f1\_time} = \text{f1\_score} - \text{total\_runtime}/30000. \tag{4.1}$$

The last value of the equation was chosen ad hoc so that an increase of 0.2 percentage points in the F1 score will only be accepted if it adds less than 60 seconds to the training time. Thus, the metric allows to search for better models without increasing too much the training time.

Subsequently, a broader grid search was run over two hyperparameters, the two model architecture and all the possible views, in order to select the best views and choose a baseline model. In this search, the models were trained for 15 epochs instead of the previous 10, to better allow them to converge and produce more useful comparisons. The results in terms of each view are shown in Fig. 4.3.

It was found that the single views were outperformed by almost all the other views. In fact, the merged view and many encoded views yielded better F1 scores than the single views, with enhancements of up to 1.6 percentage points when compared to view 3 (single3). The highest average F1 scores were obtained by encoded134, which was even better than the merged view and encoded1234 while using 25% less information. The views encoded124, encoded34, encoded14 and encoded234 also achieved competitive F1 scores, within 1 percentage point of the best view. Note that due to the

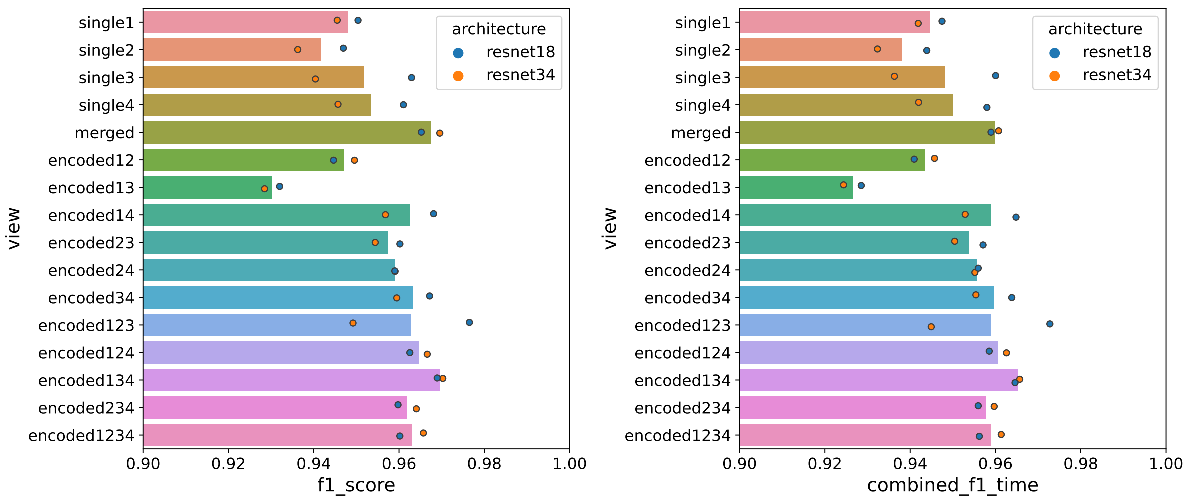

Figure 4.3: Comparison of the different views regarding F1 score and combined_f1_time, computed over the validation set. The horizontal bars show the average of the result of the two architectures. In general, views with more information, simultaneously including views with shorter and longer time windows, achieve better performance.

reduced number of trials for each view and the randomness inherent to the training process, comparisons should be made with caution when the differences are in the order of 1 percentage point or less. Also, there are many views, including all single views, which show lower results for the ResNet34 architecture than for ResNet18, which does not agree with the higher capacity of ResNet34, but could be due to the non-optimized training routine used.

Taking the total training time in consideration using the combined_f1_time metric, however, can help to filter the wanted views. For instance, the merged view takes about 6 more seconds per epoch (a 70% increase) than the encoded view encoded134, while achieving slightly worse performance regarding the F1 score. Thus, encoded134 is preferable to the merged view, as expressed by a wider gap in combined_f1_time. The view encoded134 was chosen for all subsequent models because, in addition to being the highest performer, it uses 3 channels, enabling the use of pre-trained models for transfer learning. Encoded124 could also be tried, but it has the same structure and contains approximately the same information as the chosen view, so it would be unlikely to yield significantly better results.

In addition, the results of the grid search were also evaluated in terms of the chosen architecture, as depicted in Fig. 4.4. It is observed that, on average, ResNet18 achieves better results both on the F1 score and the training time, resulting in a median combined_f1_time 0.6 percentage points higher than ResNet34. Thus, the ResNet18 was chosen for the baseline model. Nevertheless, the ResNet34 was still kept for subsequent searches for optimizations, due to its higher capacity and the potential to perform better with a different set of hyperparameters.

After deciding the baseline configuration, five models were trained for 15 epochs using the same configuration as before, and selecting the ResNet18 architecture and the encoded134 view. The training and validation curves are plotted in Fig. 4.5. It is possible to observe that both curves converge to a low value in all of the runs, with final training losses and validation losses lower than, respectively, 0.002 and 0.06 for all runs. Although there are some accentuated increases in the first half of training, especially in the validation curves, which contrast with the wanted behaviour of monotonically decrease for both curves, they are more or less expected as the 1cycle policy is being used, and the models seem to always have found a good minimum. Moreover, the F1 scores of the five models are shown in Fig. 4.6. From the five trained models, which obtained F1 scores between 95.9% and 97.4%, the best one, run 3, was chosen. Note that the best model can be an outlier, as there is variability of 1.5 percentage points between the best and worst runs and its performance is more than 1 percentage point higher than the median.

The confusion matrix of the model's predictions for the validation set is shown in Fig. 4.7. In

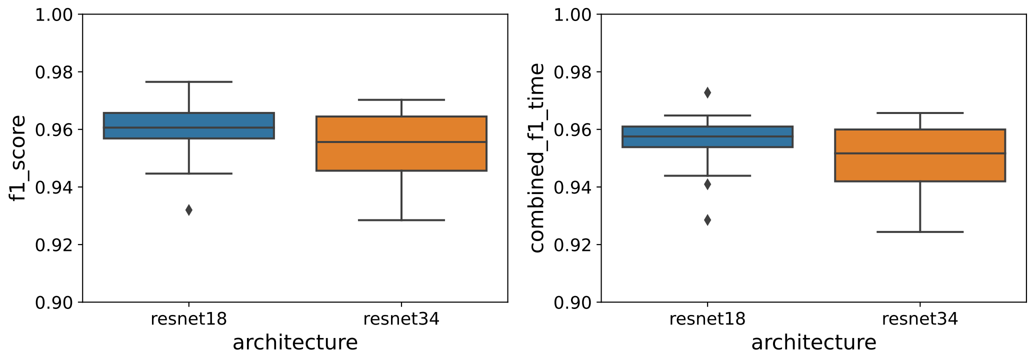

Figure 4.4: Comparison of the architectures regarding F1 score and combined_f1_time, computed over the validation set. ResNet18 achieved comparable performance while taking less time to train, resulting in higher values of F1 scores and combined_f1_time.

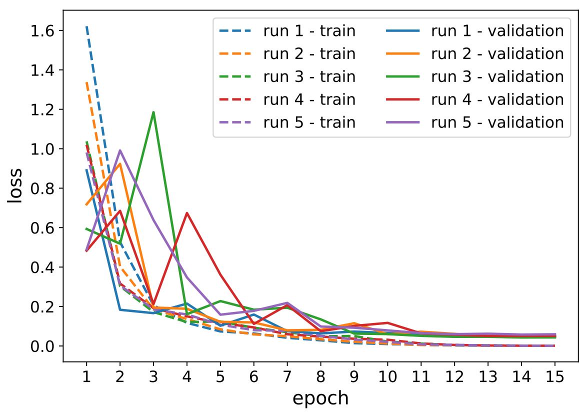

Figure 4.5: Training and validation curves of the five runs with the chosen baseline configuration. Despite some steep increases in the earlier epochs, especially in the validation loss, the training and validation losses converge to low values in all runs.

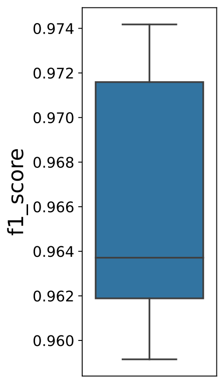

Figure 4.6: F1 score distribution of the five models trained with the baseline configuration. The best model outperforms the worst by 1.5 percentage points, with a F1 score of 97.4%.

total, there are only 13 instances out of the matrix diagonal, which correspond to the only validation examples incorrectly classified out of all 1288. Moreover, the model shows perfect precision and recall for 9 of the 22 classes (that is, all their examples were correctly identified an these classes were not attributed to any mislabeled example), and these values are simultaneously above 95% for 18 out of the 22 classes. The `Chirp` class is not included in the previous group, as one `Chirp` which was identified as a `Tomte` reduces the recall of the `Chirp` class to 90%.

The 12 validation examples where the model shows the greatest losses (all but 1 of the misclassified examples) are depicted in Fig. 4.8. In some cases, like the `Repeating_Blips` example which was identified as a single `Blip` (row 3, column 3), or the `Whistle` example that was labeled as `Blip` (row 2, column 1), the model's error is obvious. However, in other cases it is not so clear. For instance, the `None_of_the_Above` that was classified as a `Koi_Fish` (row 1, column 4) actually looks like a somewhat sideways `Koi_Fish`. Regarding the case of the misclassified `Chirp` (row 1, column 1), the high-energy at lower frequencies after the peak time makes the spectrogram almost symmetric, which is not typical in `Chirps` and could explain the confusion with the symmetrical and wide-base `Tomte` class. Additionally, the samples of two cases where the model mistook `Paired_Doves` and `Low_Frequency_Burst`, once in each direction (row 1, column 2 and row 3, column 1) actually look very similar despite being labelled with different classes. After looking at the samples of these classes

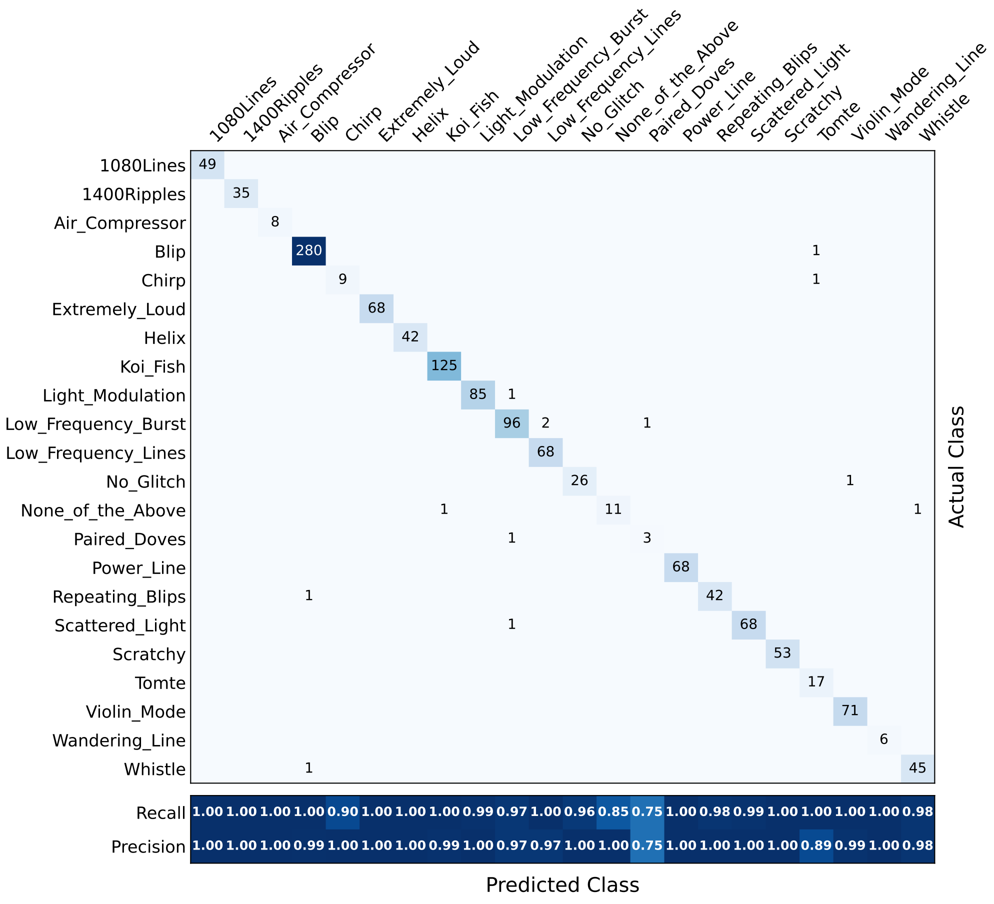

Figure 4.7: Confusion matrix of the predictions of the baseline model over the validation set. Note that only 13 examples are misclassified. The model shows precision and recall simulatenously above 95% for 18 of the 22 classes.

in the training and validation sets and finding, it is clear that this is an error in the labelling, and that the example labelled with `Low_Frequency_Burst` is indeed from the class `Paired_Doves`. If this is taken in account and the instance label is corrected, the recall and precision of the `Paired_Doves` increase to 100% and 80%, respectively, and the macro-averaged F1 score increases to 98.1%.

Taking into account the high F1 score that the model achieves, especially with the corrected label, and looking at its number of mistakes in the validation dataset, it might appear that there is not much that can be done to improve it. However, note that this model instance was chosen exactly because it was the best performing in the validation dataset, and it is not guaranteed to generalize well to the test set. In fact, from Fig. 4.6, it seems that other models trained with the same configuration can achieve significantly lower performances. Thus, it is justified to search for stabler configurations using different hyperparameters which have not yet been optimized. In addition, given that 4 of the 12 errors (the mislabelled sample is excluded) involve the five less represented classes, methods of increasing the importance of these classes can be tried.

4.3 Improving the baseline classifier

Initially, an attempt was made to tune the learning rate choice, the number of epochs and the batch size with the goal of further optimizing the model's performance. Hence, a bayesian sweep was performed over 50 configurations of the following three hyperparameters:

- number of epochs, between 8 and 25;
- batch size, sampling from the set $[32, 64, 128, 256]$;

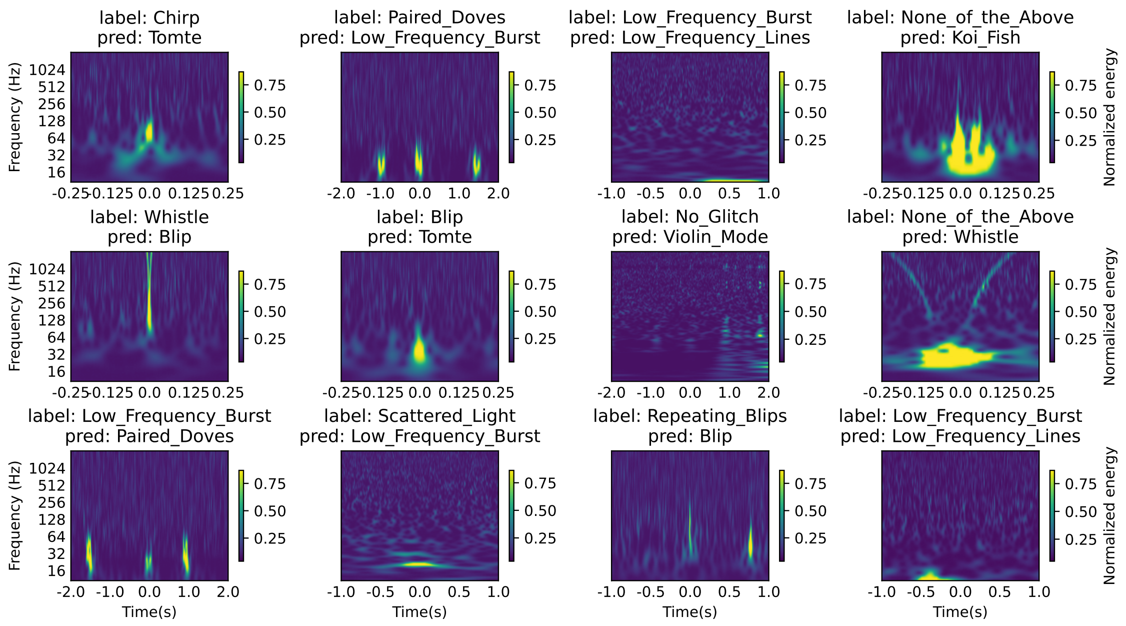

Figure 4.8: Misclassified examples of the validation set, with their true class and the model prediction. The examples are represented using a single view that was in one of the encoded view channels presented to the model and allows for a human to identify the glitch class.

- learning rate finder function, sampling from the four available methods in `fastai` (`steep`, `valley`, `slide`, `minimum`).

The configurations of the ten models with the best F1 score are compared with the baseline in Table 4.2. It was found that no model obtained a F1 score higher than the baseline, although some models obtained a slightly better combined score than the baseline by mantaining a comparable F1 score while training for less epochs. There is a prevalence of models with the `valley` suggestion function, while no model uses the `steep` function chosen for the baseline. In terms of the batch size, 32 seems to be a good size, especially in combination with `valley`. Most models were trained for 12 to 15 epochs, although there are two models that were trained for only 9 epochs and three for 20 or more epochs.

Table 4.2: Configuration of the baseline model and the ten best models found in the bayesian sweep over the number of epochs, batch size and learning rate function. Metrics are computed on the validation set. The last column shows the names of the seven configurations that were saved based on the ten models.

name	epochs	lr_function	batch_size	combined_f1_time	f1_score	configuration
baseline run 3	15	steep	64	0.9771	0.9807	baseline
jumping-sweep-48	9	valley	32	0.9776	0.9802	best1
polar-sweep-24	14	valley	32	0.9754	0.9788	—
efficient-sweep-23	15	valley	32	0.9734	0.9776	best2
valiant-sweep-33	25	minimum	128	0.9724	0.9775	best3
earnest-sweep-35	15	minimum	32	0.9719	0.9755	best4
breezy-sweep-30	12	minimum	64	0.9723	0.9749	best5
dauntless-sweep-14	13	valley	32	0.9705	0.9744	—
northern-sweep-19	9	valley	32	0.9711	0.9734	—
polished-sweep-12	20	valley	256	0.9679	0.9728	best6
summer-sweep-1	24	slide	256	0.9649	0.9715	best7

Following, additional instances of each configuration were trained in order to produce more meaningful comparisons with the baseline. Since some models are similar in terms of configuration, using close number of epochs and the `valley` function with equal batch size, not all configurations were chosen. In Table 4.2, the last column shows the name attributed to model configuration if it was saved. In total, seven different configurations were chosen, and five new models were trained using each of the configurations.

The box plots of the distributions of the F1 scores for the newly trained models using the seven configurations, in addition to the baseline (recomputed with the label correction), are shown in Fig. 4.9 for the validation set. It was found that only the `best1` configuration achieved an F1 score higher than the baseline maximum in one of the runs, but the value distribution suggests that this configuration is not better than the baseline. Configurations `best6` and `best7`, which use a 256 batch size, on the other hand, achieved medians higher than the baseline median, although their maximum value is lower than the baseline maximum. On the other hand, the configurations with the `minimum` suggestion function (`best3-5`) are clearly worse than the baseline, while `best2`, which uses the `valley` function is better than them but also appears to perform worse than the baseline.

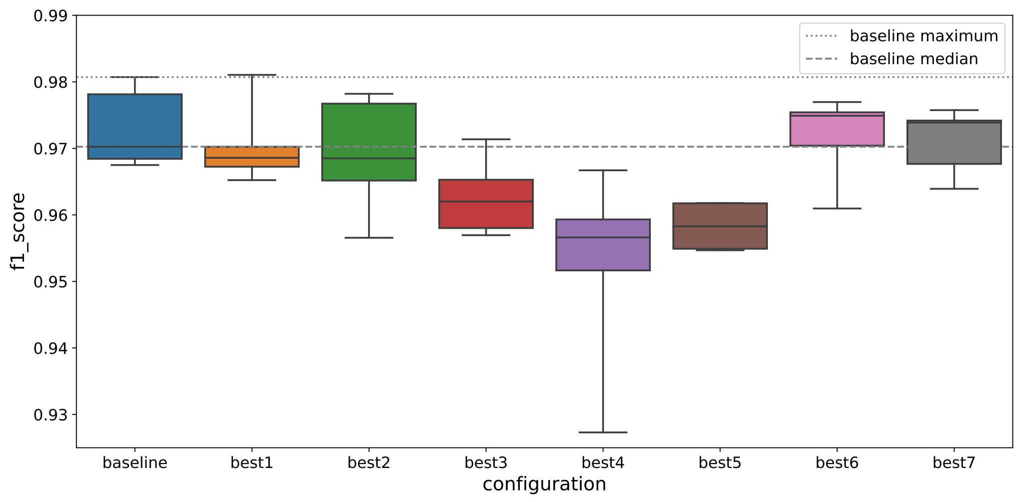

Figure 4.9: Box plots of the F1 scores on the validation set fot the five runs for the baseline and each of the seven best configurations from the first bayesian sweep. None of the configurations appear to be significantly better than the baseline.

Afterwards, a weighted loss function was tried, to penalize more the model for mistakes in the less represented classes, by introducing a class-dependent weight, w_i in the cross-entropy loss of equation (3.10):

$$\mathcal{L}(\boldsymbol{\theta}) = -\sum_{k=1}^{K} w_k \; y_k \log(\hat{p}_k). \tag{4.2}$$

If w_k is equal to 1 for all classes, the previous equation is the plain cross-entropy loss. In turn, the choice of higher w_k values for the classes with less samples increases the loss function's dependence on these classes. Thus, such weighted loss function can be interpreted as an artificial oversampling of the dataset.

One possible way to calculate the weights of each class is to use the inverse number of samples weighting scheme, where the weight of class k is given by

$$w_k = \frac{1}{N_k}, \tag{4.3}$$

with N_k being the total number of samples of class k. In order to create weights closer to unity, they are then normalized by dividing each class weight by the sum of the class weights and multiplying by the number of classes. The weights must be calculated using the total number of samples in the training and validation sets but not on the test set, to prevent any possible information leak.

Another interesting weighting scheme is the effective number of samples, proposed in [80], which obtains weight associated with each class using

$$w_k = \frac{1-\beta}{1-\beta^{N_k}}, \tag{4.4}$$

where β is a hyperparameter which allows to control the re-weighting degree from no re-weighting ($\beta = 0$) to inverse class frequency ($\beta \to 1$). The authors suggest trying β values like 0.9, 0.99, 0.999 and 0.9999.

The weights obtained in the Gravity Spy dataset using the two previous methods are shown in Fig. 4.10. As expected, the inverse strategy produces the most aggressive weighting, with the highest difference between the maximum and minimum weights. The two highest β values produce weights very close to the inverse method, so they were not tested. The lowest β was also not used, as it seems to introduce almost no re-weighting, with a difference of only 0.1 between the extreme weights. On the other hand, $\beta = 0.99$ seems to introduce a more balanced re-weighting, so it was tried in addition to the inverse weighting.

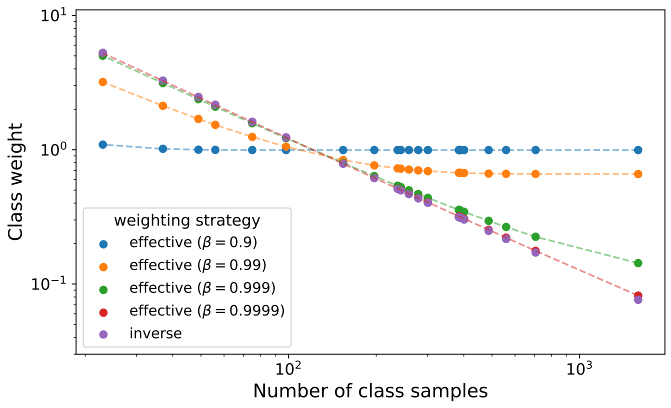

Figure 4.10: Class weights for the Gravity Spy classes using the inverse weighting scheme and the effective number of samples with different values of β. The inverse method produces the most aggressive cost-sensitivity while the effective number of samples with the highest β introduces almost no change. The same method with $\beta = 0.99$ seems to achieve a good balance.

The re-weighting strategies were compared using the baseline and `best6` configurations, in addition to a configuration equal to the baseline but with the ResNet34 architecture instead of the ResNet18. Each configuration/strategy combination was run for 10 times, and the results are shown in Fig. 4.11.

Comparing only the models without re-weighting, the increased number of runs seems to confirm that the baseline configuration is better than `best6`, although the latter achieved the highest overall F1 score. Once again, the ResNet34 architecture also seems to perform worse than the ResNet18. When the inverse weighting strategy is introduced, the performance of the baseline decreases, while the `best6` performance is not much changed and the ResNet34 F1 scores even slightly increase. Finally, the effective re-weighting strategy seems to result in similar performance for the baseline in comparison to the models without re-weighting, while the `best6` models slightly deteriorate and the ResNet34 models very slightly increase performance.

About half of the misclassification errors of the two models trained with re-weighting were found in the same exact validation samples as the unweighted baseline model, some of which not involving the minority classes. This raises the question whether the problem might be that the model is relying too much on the easier samples and has more difficulty in harder samples like the ones it constantly misclassifies on the validation set.

The focal loss function, introduced in [81] for the task of object detection, addresses this exact issue by adding to the cross-entropy loss a modulating factor which down-weights easy examples, that is, examples where the model is more confident; it has also been reported to help in imbalanced

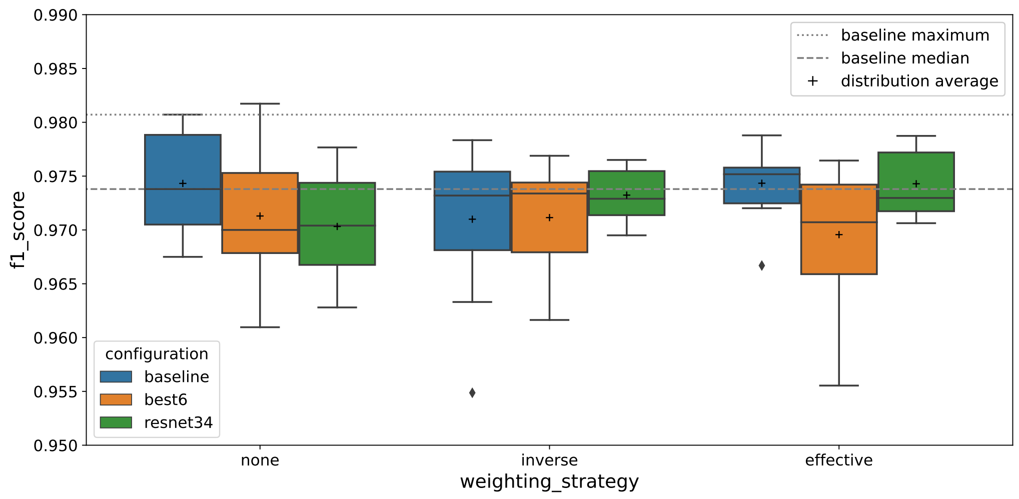

Figure 4.11: F1 scores on the validation set for three different weighting strategies using three different configurations. The re-weighting strategies do not appear to increase the performance of the configurations with the ResNet18 architecture (baseline and `best6`) but slightly increase the performance for the ResNet34.

---

classification tasks (see e.g. [82]). In practice, focal loss is combined with the re-weighted cross-entropy:

$$\mathcal{L}(\boldsymbol{\theta}) = -\sum_{k=1}^{K} w_k \ (1 - \hat{p}_k)^{\gamma} \ y_k \log(\hat{p}_k), \tag{4.5}$$

where γ is the focusing parameter. When $\gamma = 0$, the equation reduces to equation (4.2), and as γ is increased, the effect of the modulating factor does too, practically unaffecting the loss for misclassified examples with small $\hat{p}_k$ and pushing to 0 the importance of examples classified with higher confidence, that is, high $\hat{p}_k$). The authors of [81] have found that $\gamma = 2.0$ is a good default, so this value was kept constant in this work.

The baseline configuration was trained, using the focal loss, five times for each of the previous weighting strategies (including the absence of re-weighting), and the results are shown in Fig. 4.12. Focal loss does not seem to work with the inverse re-weighting strategy, and yields similar performance without re-weighting and using the effective number of samples strategy. In any case, the results with focal loss are worse than the results with cross-entropy in all cases, which suggests that the inclusion of focal loss is not positive.

To sum up, even after extensive efforts to optimize hyperparameters such as the number of epochs, the batch size and the learning rate finder function, as well as attempts to address the misclassification of harder examples using class-dependent weights and a different loss function, the baseline configuration with vanilla cross-entropy loss appears to be the best choice.

4.4 Transfer learning classifier

After being unable to improve the baseline model in the last section, an approach using transfer learning was attempted. Although the models used in transfer learning are pre-trained on the ImageNet dataset, the features from the first layers may still be relevant for the Gravity Spy dataset, possibly allowing to train models faster and achieve better results.

Initially, pre-trained models using the ResNet18 and ResNet34 architectures were trained with `fastai`'s `fine_tune` routine using the same configuration as the baseline. The results obtained over 10 runs with each architecture are compared to the models trained from scratch in Fig. 4.13. The use of transfer learning degrades the performance of the models with the ResNet18 architecture, but appears to improve the performance of the ResNet34 models, reaching F1 scores close to the ResNet18 trained from scratch. One possible explanation is that the increased number of layers allows to combine

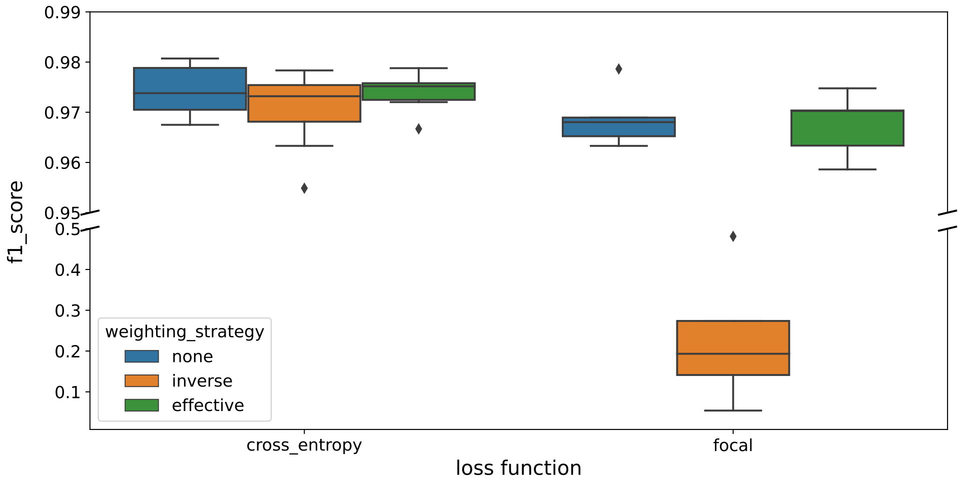

Figure 4.12: Comparison between the F1 scores on the validation set for the cross-entropy loss and focal loss functions, using the baseline configuration with three different re-weighting strategies. The use of focal loss does not improve the performance of the models.

general features learned on ImageNet in the initial layers with specific features from this dataset in the later layers, and that this is harder with only 18 layers. Moreover, it should also be considered that the models are trained with a configuration that was found to be the best for the ResNet18 trained from scratch, and not necessarily for the other cases.

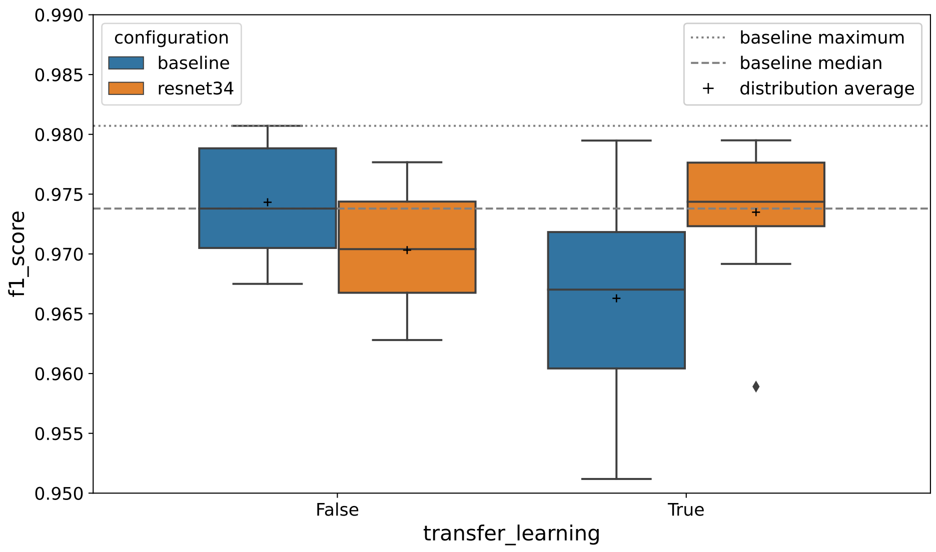

Figure 4.13: Comparison between the F1 scores on the validation set for the models trained from scratch and the ones using transfer learning. Transfer learning appears to improve the performance of the ResNet34, but does not for the ResNet18 architecture.

As the previous result suggest that ResNet18 architecture might not to be the best for transfer learning, other architectures, both from the ResNet family and other families, were tried in the next step. First, the ResNet26, a variant which has a size between the two previous architectures, was considered. Furthermore, bigger ResNet networks which might better make use of the pre-trained features, namely ResNet50 and ResNet101, were also analyzed. In terms of other architecture families, the novel Vision Transformers such as ViT [83] and Swin [84] were discarded because, as they are not CNNs, they are out of the scope of this work. On the other hand, the ConvNeXt family [56] is a strong contender, as it is fully based on CNNs whose design is inspired on the successful Vision Transformers, achieving competitive performances (see e.g. [85]).

In order to avoid selecting models that are too big, and thus, too slow to train and prone to overfitting, the number of parameters of selected models from the ResNet and ConvNeXt families was compared in Fig. 4.14. ResNet18 is the smallest model overall, with about 11 million parameters, while ResNet101 has more than 42 million parameters and ConvNeXt_Small contains almost 50 million parameters. The smallest model from the ConvNeXt family is the ConvNeXt_Nano, which has almost the same size as the ResNet26. It was decided to use models up to three times bigger than ResNet18, which includes ResNets 18 through 50 and the "nano" and "tiny" versions of ConvNeXt.

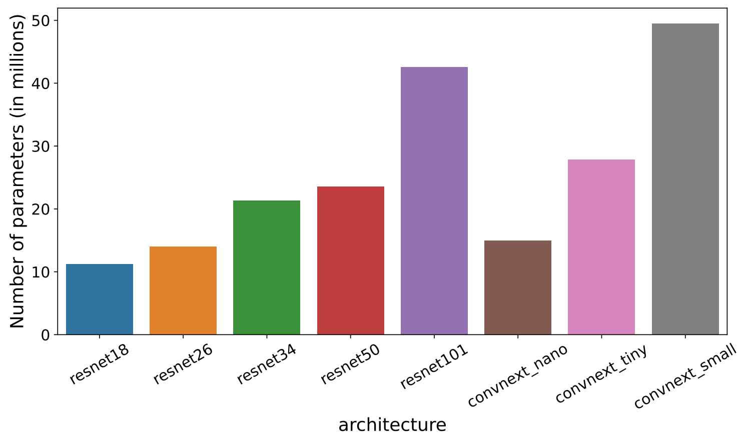

Figure 4.14: Comparison of the size of different models of the ResNet and ConvNeXt families.

Having selecting the six architectures, a grid sweep was performed in order to compare them, training each model for 4, 9 and 14 epochs (in addition to the initial frozen epoch) and using two different learning rate functions, `steep` and `minimum`, while keeping the other hyperparameters at the default values. In total, each architecture was trained six times, with the results shown in Fig. 4.15. It can be observed that ResNet18 produced the best single run, but it appears to be worse than all the other architectures when averages and medians are considered. The performance of the ResNet family seems to peak with the ResNet34 architecture, but it is surpassed by the ConvNeXt family. ConvNeXt_Nano is close to ConvNeXt_Tiny in terms of F1 score, but is preferable when the training time is considered. For these reasons, ConvNeXt_Nano[2] was chosen as the architecture for subsequent optimizations.

In addition to evaluating the different architectures, the models were evaluated in terms of the number of epochs. It was found that the models trained for only 4 epochs outperformed the others even in terms of F1 score without time considerations, which accentuates the power of transfer learning for quickly training powerful models. For this reason, the following optimizations explored a parameter space with a reduced number of epochs.

Then, a bayesian sweep was run to optimize the hyperparmeters of the transfer learning model with the ConvNeXt_nano architecture. Fifty configurations were evaluated, given the following parameter space:

- number of frozen_epochs, with a uniform integer distribution between 1 and 5
- number of unfrozen epochs, using a uniform integer distribution between 1 and 10;
- batch size, sampling from the set $[32, 64, 128, 256]$;
- learning rate finder function, sampling from the four available methods in `fastai`;
- loss function: either cross-entropy or focal loss;
- re-weighting strategy, which could be none, inverse of class frequency, or effective number of samples.

The configurations of the overall top 7 models in terms of F1 score in addition to the three best models trained for 6 epochs or less are reported in Table 4.3, as well as the baseline configuration.

[2]The ConvNeXt_Nano architecture is briefly described on appendix A.

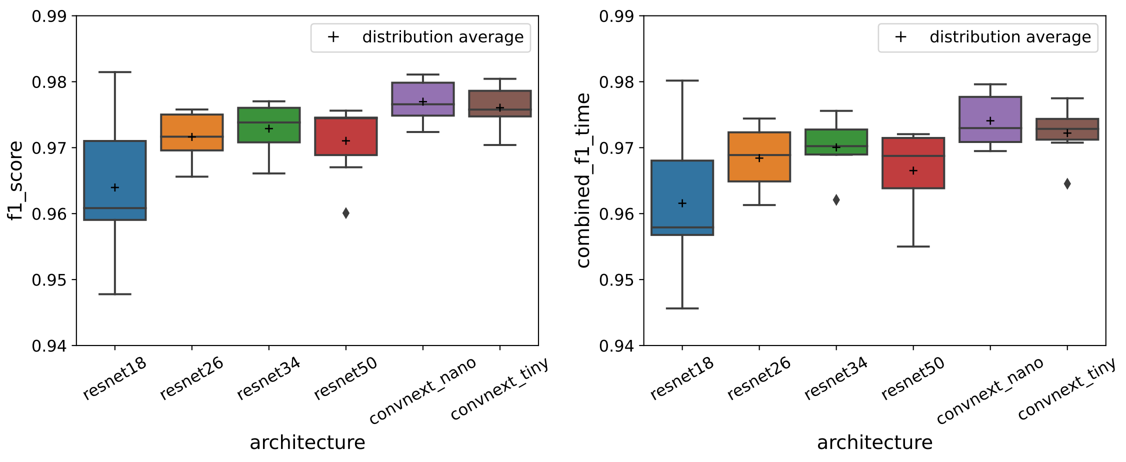

Figure 4.15: Comparison of the architectures regarding F1 score and combined_f1_time, computed over the validation set. ConvNeXt_Nano is the best model in terms of both metrics.

Four of these models achieved higher performance than the baseline, both in terms of F1 score and combined_f1_time, with an increase of almost 0.3 percentage points in F1 score. Moreover, they take generally less epochs to train, achieving good results with 7 to 10 epochs in total. In terms of batch size, 64 appears to be a good default value, as it appears in most of the top runs. The `minimum` learning rate finding function appears to be the best choice, closely followed by `steep`. Unlike the results obtained in the models trained from scratch, the use of the focal loss function and loss re-weighting strategies is helpful, as they are present in most of the best models. As to the faster models, they all use the cross-entropy loss with a re-weighting strategy, and achieve F1 scores higher than 97.5% with minimal training.

Table 4.3: Configuration of the seven best overall and the three best fast pre-trained models found in the bayesian sweep, in addition to the baseline (trained from scratch). The top four models slightly beat the baseline model in both metrics.

bs[3]	epochs[4]	lr_function	loss	re-weighting	combined_f1_time	f1_score	configuration
64	0 + 15	steep	cross_entropy	none	0.9771	0.9807	baseline
64	3 + 7	minimum	focal_loss	effective	0.9804	0.9836	tl_best1
64	2 + 6	minimum	focal_loss	none	0.9804	0.9830	tl_best2
32	1 + 6	steep	focal_loss	none	0.9787	0.9818	tl_best3
128	2 + 7	minimum	focal_loss	effective	0.9785	0.9816	tl_best4
64	2 + 8	minimum	focal_loss	effective	0.9766	0.9800	—
64	2 + 8	minimum	cross_entropy	inverse	0.9760	0.9792	tl_best5
256	1 + 9	steep	focal_loss	effective	0.9755	0.9789	tl_best6
64	1 + 5	minimum	cross_entropy	inverse	0.9749	0.9768	tl_fast1
64	1 + 4	steep	cross_entropy	inverse	0.9750	0.9765	tl_fast2
64	1 + 4	steep	cross_entropy	effective	0.9746	0.9762	tl_fast3

The nine chosen configurations were more thoroughly evaluated by running them five times each, and the results are shown in Fig. 4.16. The best overall score was obtained by the `tl_best6` configuration, which beat the baseline's best run. The `tl_best5` and `tl_fast1` configurations achieved F1 scores higher than the baseline median in all five runs, and both also produced one run with F1 score very close to the baseline maximum. Moreover, `tl_fast1` has the highest median among all models. This configuration outperforms the baseline while being trained for only 6 epochs (about one third of the baseline model), which is a good indicator of the power of transfer learning.

[3] "bs" denotes the batch size.
[4] The two values in each entry of the "epochs" column are, respectively, the number of frozen and unfrozen epochs.

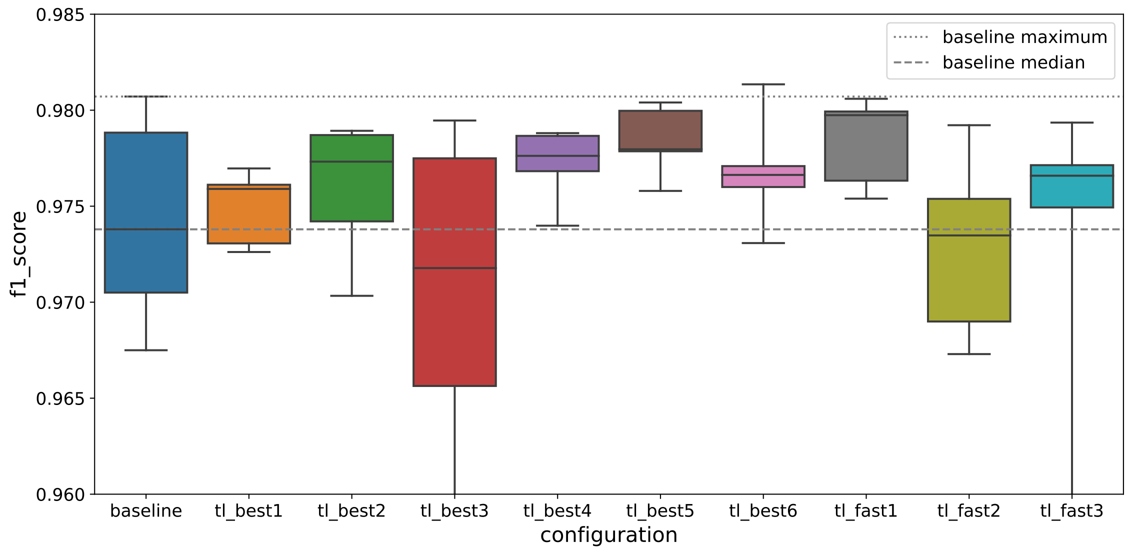

Figure 4.16: F1 scores on the validation set of the five runs for the baseline and the nine configurations selected from the second bayesian sweep. `tl_best6` obtained the best F1 score in a single run, but `tl_best5` and `tl_fast1` appear to be the more consistent configurations, always performing above the baseline median.

It is true that the baseline and transfer learning models use a different architecture and that the improvement could be partially due to the ConvNeXt architecture. To resolve this question, a bayesian sweep with parameters similar to the previous bayesian sweep, but training the ConvNeXt models from scratch, was performed. This architecture proved very difficult to train from scratch, and even when training for about 25 epochs the highest F1 score was lower than 95%, which is worse than the first quick models using encoded views shown in Fig. 4.3.

One final investigation was performed to assess whether the use of image augmentation techniques would improve the models, as reported in [78]. Image augmentation allows to artificially increase the training data, as each sample experiences random transformations that yield different but still recognizable images. Following the procedure of [78], each image suffers a random horizontal and vertical shift and is slightly zoomed in or out at each epoch. The maximum shift value for each dimension was defined as shift_fraction × dim_length and the zoom ratio as $(1 - \text{zoom\_range}, 1 + \text{zoom\_range})$. Thus, shift_fraction and zoom_range are two additional hyperparameters which can be optimized.

Two bayesian sweeps were performed over 30 runs each, using the `tl_best5` and `tl_fast1` configurations, in order to optimize the image augmentation hyperparameters for each configuration, using a distribution of values from 0 to 0.10 for both of them. The shift and zoom values from the three runs with highest F1 score for each configuration are shown in Table 4.4, and the best four achieved higher F1 scores than the baseline's best run.

Each of the previous configurations was repeated ten times, and the results are plotted in Fig. 4.17. The very high F1 scores obtained in the bayesian sweep could not be replicated later, so they were likely very lucky outliers. Aside from `tl_best5_aug2`, no other set of augmentation hyperparameters produced results comparable to the versions without data augmentation. Moreover, `tl_best5_aug2` has only minimal zoom and shifts in comparison to the other configurations, which yielded considerably poorer results than the versions without augmentation. Although data augmentation is frequently used in many applications with low sized an imbalanced datasets, results here showed that it does not improve the performance of the models on the validation set or, at least, that only a very small amount of zooming and shifting could be useful. To reinforce this hypothesis, another hyperparameter sweep similar to the one at the beginning of this section was performed, while also varying the zoom and shift, and it failed to produce F1 scores higher than the version without augmentation.

Finally, the best model of `tl_best5` was chosen as the transfer learning classifier. In comparison

Table 4.4: Selected configurations with data augmentation, based on the top performing configurations in the previous sweep, shown in Fig. 4.16. The `tl_fast1` models with almost no zoom and a shift fraction of 8.5% achieved the highest F1 scores so far.

base configuration	zoom_range	shift_fraction	f1_score	configuration
tl_fast1	0.000	0.085	0.9843	tl_fast1_aug1
tl_fast1	0.015	0.085	0.9842	tl_fast1_aug2
tl_fast1	0.035	0.005	0.9826	tl_fast1_aug3
tl_best5	0.085	0.070	0.9822	tl_best5_aug1
tl_best5	0.015	0.005	0.9800	tl_best5_aug2
tl_best5	0.070	0.050	0.9786	tl_best5_aug3

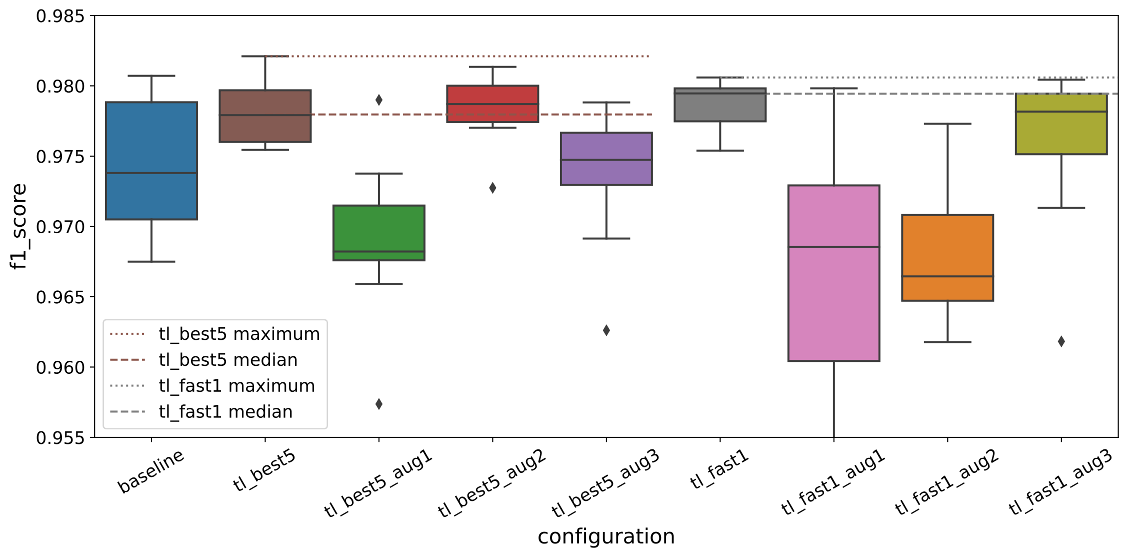

Figure 4.17: F1 scores on the validation set of the ten runs for each configuration of Table 4.4 and the respective base configurations without augmentation. Data augmentation does not appear to help improve the performance of the models on the validation set.

with the baseline model, it only increases the F1 score from 98.07% to 98.21%, but the transfer learning configuration appears to result in stabler performances across multiple training trials, with its worst performance higher than the baseline median. In fact, the baseline configuration achieves a F1 score of $(97.4 \pm 0.5)\%$, less than `tl_best5`'s $(97.8 \pm 0.2)\%$ and `tl_fast1`'s $(97.9 \pm 0.2)\%$. Note that the previous F1 scores are reported in the form $(\mu \pm \sigma)\%$, where μ is the distribution average and σ its standard deviation.

4.5 Evaluation of the selected models

Finally, the baseline and `tl_best5` model's performance was evaluated on the test dataset, which had been kept apart until that point. It was found that the F1 score of the baseline slightly dropped to 97.18%, while `tl_best5`, the best model on the validation set, performed worse than the baseline on the test set, with a F1 score of 96.84%. Nevertheless, the F1 scores, obtained on data which was never seen by the models nor used to perform choices regarding hyperparameter optimization, are still very high for both models.

Despite not being the best on the validation set, the baseline model turned out to show better generalization than the best optimized model on the validation set, which could be due to having overfitted the validation dataset, as it was used to validate hundreds of models through this work. In hindsight, it is possible that using k-fold cross validation [86], at least for the repeated runs using the same configurations, would have been a better approach to estimate the performance of the models.

The confusion matrix of the baseline model, using the predictions on the test set, is shown in Fig. 4.18. The model shows both precision and recall of at least 95% for 19 out of the 22 classes, as it only appears to have difficulties in the `Air_Compressor`, `None_of_the_Above` and `Paired_Doves` classes, all of which have very few examples on the training set. Moreover, the model shows perfect precision and recall on the `Chirp` class, which might be useful for the task of detecting GW signals, as explored on the next section. Looking at the off-diagonal entries of the matrix, the only mistakes which occurred more than once were confusing two `Repeating_Blips` samples for `Blips` and identifying three `None_of_the_Above` examples as `Light_Modulation`. In these cases, it was found that the correct label was the the class where the model had the second highest score, which is a good indicator, related to the so-called top-2 accuracy. In fact, the model shows a top-2 accuracy of 99.69%, that is, for that percentage of the samples the correct class is among the model's top two predictions.

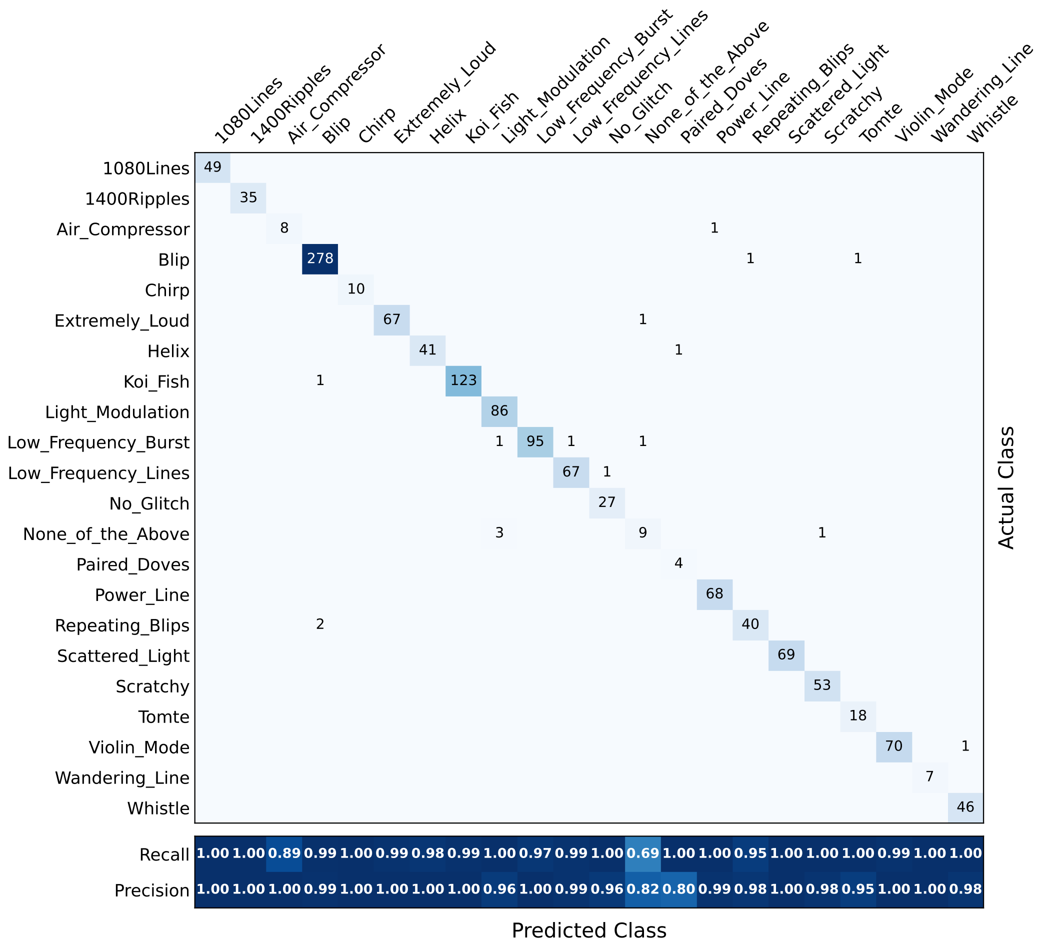

Figure 4.18: Confusion matrix of the predictions of the baseline model over the test set. The model shows precision and recall of at least 95% for 19 of the 22 classes, and perfect scores for `Chirps`.

Finally, the two models were compared to the literature in terms of their F1 score and accuracy[5] in Table 4.5. Both selected models of this work obtained competitive results, higher than the merged view CNNs and even than an ensemble of models, in case of the baseline. The baseline model, which uses a ResNet18 architecture trained from scratch, is only slightly worse than the fine-tuned ResNet50 from [87], with a decrease in F1 score of less than 0.5 percentage points. However, the authors of the paper used a different dataset split, without a validation dataset, which was distributed over the training and test sets. This lack of a validation dataset raises the question of how the final model was chosen, since in a related pre-print paper [78] the authors state that the models were check-pointed after each epoch and subsequently the best check-points were chosen. This would not be possible without either basing the choice on training set metrics and almost surely overfitting the data (thus yielding lower test scores), or making the decision based on the test dataset, which means that the real generalization performance of the model would be worse than reported.

Table 4.5: Performance of different models on the Gravity Spy dataset. F1 scores of other works were calculated from the reported confusion matrices, when available. The best model of this work is only outperformed by the fine-tuned ResNet50.

Model	F1 score (%)	accuracy (%)	Notes
merged view CNN [77]	not reported	96.89	different dataset version (20 classes)
merged view CNN [75]	not reported	97.67	improved version of [77]
hard fusion ensemble [75]	not reported	98.21	combines four CNNs
fine-tuned ResNet50 [78, 87]	97.65	98.84	different split (no validation set)
tl_best5 [this work]	96.84	98.14	fine-tuned ConvNeXt_Nano
baseline [this work]	97.18	98.68	ResNet18 trained from scratch

4.6 Gravitational wave detection with the glitch classifiers

As mentioned before, the Gravity Spy dataset contains a few examples of chirps, which are simulated GW signals from CBCs. In addition, the best models have shown perfect precision and recall for chirp classification in the Gravity Spy test set. These two reasons motivated a brief investigation of whether the models trained in this work would be able to identify real gravitational waves as `Chirps`, using data from the first two aLIGO and AdV observing runs. As a note of caution, it is worth mentioning that the models were trained with the available hardware injections in the dataset, which do not include software injections (from PyCBC approximants) in real noise experiments that would allow a bigger coverage of the CBC parameter space.

The strain data for the 11 confident detections in the O1 and O2 runs, for each aLIGO detector, Hanford ("H1") and Livingston ("L1"), was obtained from the Gravitational Wave Open Science Center (GWOSC) [88]. For each data sample, a Q-graph of the strain data in the 5 seconds around the GPS time of the detection was obtained using the `q_transform` method from the `GWpy` library [89]. Furthermore, four crops of this Q-graph, centered at the time with the highest energy, were obtained, one for each different single view duration in the Gravity Spy dataset. From the resulting 22 samples (11 events times 2 detectors), 10 were discarded due to being buried in the background noise, with no visible chirp, or, in the case of GW170817 for the Livingston detector, being in the middle of a very energetic glitch. The event and detector for selected sample is found in the titles of Fig. 4.19

Then, the views with 0.5, 2.0 and 4.0 seconds of duration were combined taking advantage of the RGB color channels, as done previously to obtain the `encoded134` view. Finally, the resulting images were passed through the baseline model, with no further training, and the resulting predictions are depicted in Fig. 4.19, ordered by descending order of loss. The model correctly identified 3 out of the 12 examples as `Chirps`, thus having a 25% recall. Moreover, in the two misclassified samples with the lowest loss, the correct class was among the top 3 classes. In all the other cases, `Chirp` was not even between the top five predictions of the model.

The poor performance of the model could be partially explained by the fact that, despite trying to replicate the Gravity Spy spectrogram creation pipeline as closely as possible, the lack of information on how to do so demanded some choices that may not agree with the original dataset. Moreover, examples like the first three in the top row show a slightly different chirp morphology, being longer in duration than the chirp examples in Gravity Spy and different enough than other classes, resulting in a `None_of_the_Above` prediction. Additionally, the samples classified as `Scratchy` have a chirp-like morphology, but most of the chirps are much less distinguishable from the background noise than in the original dataset. In these cases, the GW signal appears to be ignored, and the ripply path noise present in the mid frequencies leads the model to predict `Scratchy`.

[5]As previously stated, accuracy is not a good metric for very imbalanced datasets. Nevertheless, its value is reported to facilitate comparisons with the literature.

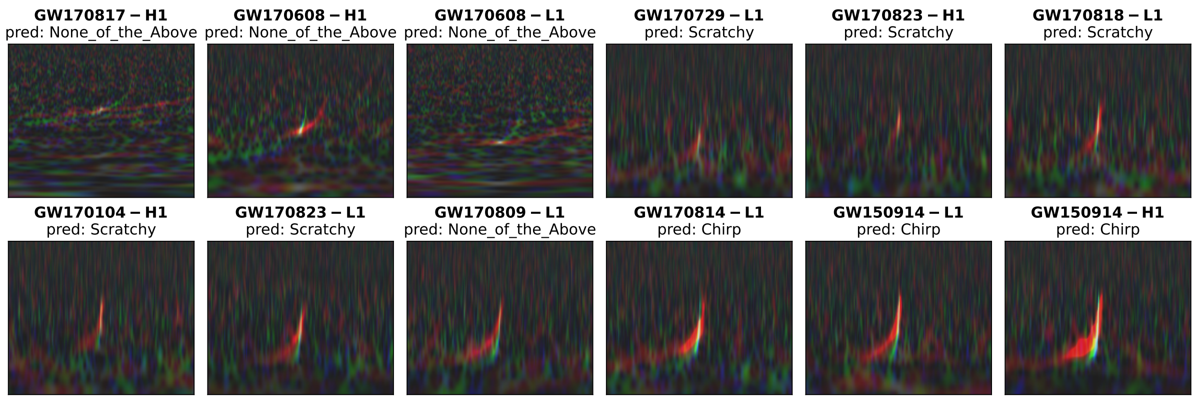

Figure 4.19: Real GW examples fed to the baseline model and the respective predictions. Only 3 of the 12 examples were correctly identified as `Chirp`. The error could be due to examples having a morphology not seen during training or being less energetic than training examples.

The `tl_best5` model was also evaluated on the same samples, and it was found that its performance was much better, even when using samples obtained with previously tested pipelines where the views had not been aligned and/or contained excessive background noise. For these pipelines, the baseline model would classify less than 3 examples correctly, while the transfer learning model always outputted at least 7 correct predictions.

The predictions for the `tl_best5` model using the same samples as previously are shown in Fig. 4.20. For these samples, the model successfully identifies 8 of the 12 chirps, achieving a 75% recall. Three of the misclassified examples are chirps with a much longer duration, which are very different than the examples used to train the model. In fact, they all result from low mass CBCs, as GW170608 was originated in the merger of two low-mass Black Holes [90] and GW170817 in a Binary Neutron Star inspiral [90]. In the case of the signal identified as `Blip`, the GW signal is from GW170823, the merger of two high mass black holes [90], resulting in a narrow shape very similar to a blip.

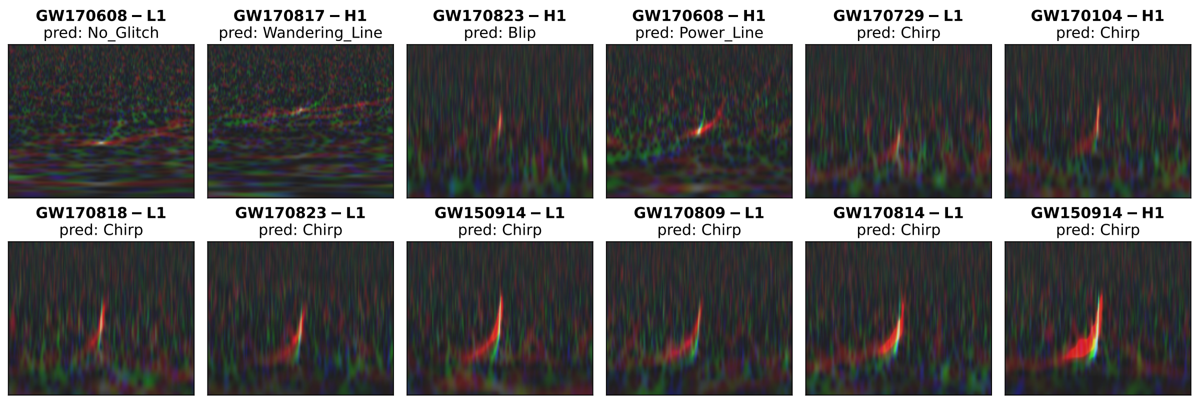

Figure 4.20: Real GW examples fed to the `tl_best5` model and the respective predictions. The model shows a recall of 75%, successfully identifying 8 of the examples as `Chirp`. Most of the misclassified samples are chirps that are much longer than the ones in the Gravity Spy dataset.

The performance of the two models tested with real gravitational wave signals is very distinct, despite both having perfect performance in the Chirps of the test set. Aside from achieving better performance, the model trained with transfer learning is much more robust to samples which are different than the training samples in terms of background noise and alignment. One possible explanation for the better performance and behaviour with samples slightly out of distribution could be the benefit of pre-trained features, which were reported in [91] to produce a boost in generalization even after fine-tuning.

Chapter 5

Conclusion

In this work, classifiers were developed to reliably categorize different noise transients found in the LIGO detectors, using Convolutional Neural Networks. Moreover, these models were capable of identifying real gravitational wave events from Compact Binary Coalescences, despite being trained on a low number of relevant examples.

Initially, a baseline model was quickly chosen for comparison with later optimized models. During this phase, it was found that encoding the different spectrogram time windows in different color channels was an efficient way of presenting additional information to the model with small impact in time complexity. Moreover, a simple residual network with just 18 layers was found enough to achieve good results. Then, a class-dependent weighted loss and the focal loss function were tried in order to address the imbalance of the dataset, but they were found not to improve the baseline performance. Subsequently, another approach based on transfer learning was tried, taking advantage of a pre-trained state of the art small CNN architecture. Using a bayesian hyperparameter optimization sweep, it was possible to find model configurations that could be trained faster and performed better than the baseline on the validation dataset.

When the better model trained from scratch, which was the baseline model, and the best model obtained with transfer learning, `tl_best5`, were compared on the test set, it was found that the baseline generalized slightly better, with a macro-averaged F1 score of 97.18%, higher than the 96.84% achieved by the `tl_best5` model. This unexpected result may indicate that during this work the validation dataset has been overfitted, which is plausible given the amount of models tested. In future works which rely on many optimization experiences, more robust validation techniques such as cross-validation can be useful to address this problem. Nevertheless, both results are very good, in line with the previous results found in the literature.

Finally, both models were tested on real GW samples obtained from the first two aLIGO and AdV observing runs. Despite the small number of relevant samples in the dataset, which had less than 50 Chirps in the training dataset, the models could correctly identify real gravitational wave events. In the case of the baseline model, it was found that its predictions were very dependent on the parameters chosen for the spectrogram creation, and it failed to identify signals where chirps could be visually found. The `tl_best5` model, on the other hand, was much more robust to those parameters, possibly due to improved generalization from the usage of transfer learning. It always achieved at least 75% recall, having most difficulties with longer chirps, which were not present on the Gravity Spy dataset.

One limitation to this work is the fact that the dataset only contained glitches from the two first aLIGO and AdV observing runs, as currently the data from O3 is being analyzed and new glitch classes are still being proposed. In order to keep up with the glitch classes present in the O4 run, starting on March 2023, it is important to train the classifiers in a dataset that includes O3 data, so a future step could be to train models on the portion of O3 data that is already labelled. Still, the models developed during the course of this work can be useful for a preliminary categorization of glitches on new data, as they could work in real time to classify the spectrograms generated by the Omicron trigger pipeline [92]. In addition, the insights gathered regarding the best architectures, sample formats and hyperparameters will probably be useful for an updated version of the dataset, and the code developed can be easily changed to train models for the new dataset.

An important factor which affects the quality of the classifiers is the number of samples available, particularly from the minority classes, as the performance of supervised Deep Learning algorithms tends to improve with the increase of the number of labelled examples. In this regard, Generative Adversarial Networks [93] are a good prospect for the generation of synthetic data, which could help to populate the less represented classes and increase the classifiers' performance even for the rarer glitches.

Appendix A

Description of the most used architectures

A.1 ResNet18 and ResNet34

ResNet18 and ResNet34 are the smallest versions of the original Residual Neural Networks introduced by [53] and presented in section 3.5.1. The architectures have 18 and 34 layers, respectively, and both consist of one stem block followed by four stages of stacked ResNet blocks like the ones seen in Fig. 3.10, and a final fully connected layer. Each stage downsamples the image size to a half, and there are skip connections between the beginning of each residual block. There is a Batch Normalization block after each single convolution and each convolutional layer (group of 2 convolutions) is followed by the ReLU activation function. The structures of the ResNet architectures used are summarized in Table A.1. Note that the only difference between the 18-layer and 34-layer variants is the number of residual blocks stacked in each stage.

Table A.1: Structure of the ResNet architectures. The building blocks are shown in brackets; inside them, $n \times n$ indicates the kernel size and the number that follows is the number of feature maps. † denotes that the first convolution of the stage performs down-sampling using a stride of 2. The output size considers an input of 140 × 170 as done in this work, and the head of the network is already customized for the number of classes in Gravity Spy.

layer group name	output size	ResNet18	ResNet34
stem	35×43	7×7, 64, stride 2	
		3×3 max pool, stride 2	
stage 1	35×43	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$
stage 2†	18×22	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$
stage 3†	9×11	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$
stage 4†	5×6	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$
head	1×1	5 × 6 average pool	
		512 × 22 fully connected, softmax	

A.2 ConvNeXt_Nano

ConvNeXt is a family of CNNs which were created by iteratively modifying the ResNet architecture while being guided by the ideas used in the latest Vision Transformer networks [83]. Hence, ConvNeXt networks take advantage of Transformer-inspired design principles such as:

- a different stage compute ratio (1:1:3:1);
- a simpler downsampling strategy in the stem;
- the use of depthwise convolutions, a special type of convolution which applies a different filter for each channel, only mixing information in the spatial dimension (height and width);

- the inverted bottleneck, i.e., having an inner convolutional layer in the ConvNeXt block which has a feature map size bigger than the first and last convolutions of the block;
- larger kernel sizes;
- less and distinct activation functions (using the Gaussian Error Linear Unit (GELU) [94], a smoother version of the ReLU) and normalization layers (Layer Normalization [95] layers instead of Batch Normalization);
- the usage of separate downsampling layers.

The ConvNeXt family preserves the successful principle of residual connections, allowing to skip each building block similarly to the ResNets, as shown in Fig. A.1. More details can be found in the ConvNeXt paper [56].

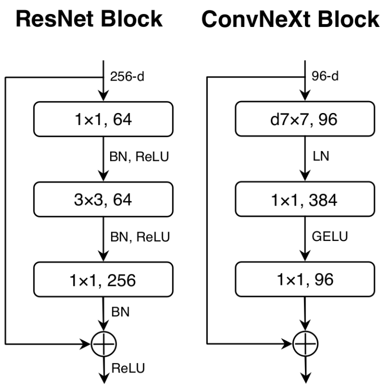

Figure A.1: Comparison between the ResNet and ConvNeXt building blocks. Adapted from [56].

ConvNeXt_Nano is a custom design created by the `timm` library, which downsamples the ConvNeXt_Tiny network by reducing the number of blocks in each stage by a third and slightly reducing the feature map sizes, achieving a size between ResNet18 and ResNet34. The summary of the structure of the network can be found on Table A.2.

Table A.2: Structure of the ConvNeXt_Nano architecture. The notation is the same as in Table A.1. The "d" before some filter sizes indicates a depthwise convolution. As previously, the output sizes and head take into account the image size and dataset used in this work.

layer group name	output size	ConvNeXt_Nano
stem	35×42	4×4, 80, stride 4
stage 1	35×42	$\begin{bmatrix} \text{d}7\times7,\ 80 \\ 1\times1,\ 320 \\ 1\times1,\ 80 \end{bmatrix}\times2$
stage 2	17×21	2×2, 160, stride 2
		$\begin{bmatrix} \text{d}7\times7,\ 160 \\ 1\times1,\ 640 \\ 1\times1,\ 160 \end{bmatrix}\times2$
stage 3	8×10	2×2, 320, stride 2
		$\begin{bmatrix} \text{d}7\times7,\ 320 \\ 1\times1,\ 1280 \\ 1\times1,\ 320 \end{bmatrix}\times6$
stage 4	4×5	2×2, 640, stride 2
		$\begin{bmatrix} \text{d}7\times7,\ 640 \\ 1\times1,\ 2560 \\ 1\times1,\ 640 \end{bmatrix}\times2$
head	1×1	4 × 5 average pool
		640 × 22 fully connected, softmax

Bibliography

[1] The LIGO Scientific Collaboration *et al.*, “Advanced LIGO,” *Classical and Quantum Gravity*, vol. 32, no. 7, p. 074001, 2015. doi: 10.1088/0264-9381/32/7/074001.

[2] The LIGO Scientific Collaboration, the Virgo Collaboration *et al.*, “Observation of gravitational waves from a binary black hole merger,” *Physical Review Letters*, vol. 116, no. 6, pp. 1–16, 2016. doi: 10.1103/PhysRevLett.116.061102.

[3] The LIGO Scientific Collaboration, the Virgo Collaboration *et al.*, “Tests of general relativity with the binary black hole signals from the LIGO-Virgo catalog GWTC-1,” *Physical Review D*, vol. 100, no. 10, p. 104036, 2019. doi: 10.1103/PhysRevD.100.104036.

[4] The LIGO Scientific Collaboration, the Virgo Collaboration *et al.*, “A Gravitational-wave Measurement of the Hubble Constant Following the Second Observing Run of Advanced LIGO and Virgo,” *The Astrophysical Journal*, vol. 909, no. 2, p. 218, 2021. doi: 10.3847/1538-4357/abdcb7.

[5] F. Acernese *et al.*, “Advanced Virgo: a second-generation interferometric gravitational wave detector,” *Classical and Quantum Gravity*, vol. 32, no. 2, p. 024001, 2015. doi: 10.1088/0264-9381/32/2/024001.

[6] The LIGO Scientific Collaboration, the Virgo Collaboration, the KAGRA Collaboration *et al.*, “GWTC-3: Compact Binary Coalescences Observed by LIGO and Virgo During the Second Part of the Third Observing Run,” pp. 1–82, 2021. arXiv preprint: 2111.03606.

[7] L. Nuttall, “Characterizing transient noise in the LIGO detectors,” *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 376, no. 2120, p. 20170286, 2018. doi: 10.1098/rsta.2017.0286.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, ISBN: 9780262035613.

[9] Y. Lecun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015. doi: 10.1038/nature14539.

[10] T. Damour, “Poincaré, the dynamics of the electron, and relativity,” *Comptes Rendus Physique*, vol. 18, no. 9, pp. 551–562, 2017. doi: 10.1016/j.crhy.2017.10.006.

[11] A. Einstein, “Die grundlage der allgemeinen relativitätstheorie,” *Annalen der Physik*, vol. 354, no. 7, pp. 769–822, 1916. doi: 10.1002/andp.19163540702.

[12] J. Taylor, L. Fowler, and P. McCulloch, “Measurements of general relativistic effects in the binary pulsar PSR1913+16,” *Nature*, vol. 277, no. 5696, pp. 437–440, 1979. doi: 10.1038/277437a0.

[13] K. Riles, “Gravitational waves: Sources, detectors and searches,” *Progress in Particle and Nuclear Physics*, vol. 68, no. 9, pp. 1–54, 2013. doi: 10.1016/j.ppnp.2012.08.001.

[14] “LIGO - Sources and Types of Gravitational Waves,” https://www.ligo.caltech.edu/page/gw-sources, Accessed: 06/07/2022.

[15] J. Cervantes-Cota *et al.*, “A brief history of gravitational waves,” *Universe*, vol. 2, no. 3, pp. 1–30, 2016. doi: 10.3390/universe2030022.

[16] J. Weber, “Observation of the thermal fluctuations of a gravitational-wave detector,” *Physical Review Letters*, vol. 17, pp. 1228–1230, 1966. doi: 10.1103/PhysRevLett.17.1228.

[17] P. Kafka, *Optimal Detection of Signals through Linear Devices with Thermal Noise Sources, and Application to the Munich-Frascati Weber-Type Gravitational Wave Detectors*. Springer US, 1977, pp. 161–241. doi: 10.1007/978-1-4684-0853-9_10.

[18] The AURIGA collaboration *et al.*, “The ultracryogenic gravitational wave antenna AURIGA,” *Physica B: Condensed Matter*, vol. 194-196, pp. 1–2, 1994. doi: 10.1016/0921-4526(94)90330-1.

[19] T. Schuldt, “An optical readout for the LISA gravitational reference sensor,” Ph.D. dissertation, Humboldt University of Berlin, 2010. url: https://d-nb.info/1014974828/34.

[20] The LIGO Scientific Collaboration, the Virgo Collaboration *et al.*, "GW150914: The advanced LIGO detectors in the era of first discoveries," *Physical Review Letters*, vol. 116, no. 13, pp. 1–12, 2016. doi: 10.1103/PhysRevLett.116.131103.

[21] B. Willke *et al.*, "The GEO 600 gravitational wave detector," *Classical and Quantum Gravity*, vol. 19, no. 7, pp. 1377–1387, 2002. doi: 10.1088/0264-9381/19/7/321.

[22] T. Accadia *et al.*, "Virgo: a laser interferometer to detect gravitational waves," *Journal of Instrumentation*, vol. 7, no. 03, pp. P03 012–P03 012, 2012. doi: 10.1088/1748-0221/7/03/P03012.

[23] D. Davis *et al.*, "LIGO detector characterization in the second and third observing runs," *Classical and Quantum Gravity*, vol. 38, no. 13, pp. 1–51, 2021. doi: 10.1088/1361-6382/abfd85.

[24] J. Powell *et al.*, "Generating transient noise artifacts in gravitational-wave detector data with generative adversarial networks," pp. 1–15, 2022. arXiv preprint: 2207.00207.

[25] D. Bersanetti *et al.*, "Advanced Virgo: Status of the Detector, Latest Results and Future Prospects," *Universe*, vol. 7, no. 9, p. 322, 2021. doi: 10.3390/universe7090322.

[26] The KAGRA Collaboration, the LIGO Scientific Collaboration, the Virgo Collaboration *et al.*, "Prospects for observing and localizing gravitational-wave transients with Advanced LIGO, Advanced Virgo and KAGRA," *Living Reviews in Relativity*, vol. 23, no. 1, p. 3, 2020. doi: 10.1007/s41114-020-00026-9.

[27] "LIGO - Latest Update on Start of Next Observing Run (O4)," https://www.ligo.caltech.edu/news/ligo20220617, Accessed: 14/07/2022.

[28] K. Somiya, "Detector configuration of KAGRA - the Japanese cryogenic gravitational-wave detector," *Classical and Quantum Gravity*, vol. 29, no. 12, p. 124007, 2012. doi: 10.1088/0264-9381/29/12/124007.

[29] T. Mitchell, *Machine Learning.* McGraw-Hill, 1997, ISBN: 9780070428072.

[30] F. Chollet, *Deep Learning with Python*, 2nd ed. Manning Publications, 2021, ISBN: 9781617296864.

[31] O. Russakovsky *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015. doi: 10.1007/s11263-015-0816-y.

[32] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems*, vol. 1, 2012, pp. 1097–1105. url: https://dl.acm.org/doi/10.1145/3065386.

[33] "NVIDIA - CUDA Technology," http://www.nvidia.com/CUDA, Accessed: 29/07/2022.

[34] J. Deng *et al.*, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, vol. 9, no. 8, 2009, pp. 248–255. doi: 10.1109/CVPR.2009.5206848.

[35] H. Sak, A. Senior, and F. Beaufays, "Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition," 2014. arXiv preprint: 1402.1128.

[36] A. Vaswani *et al.*, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 6000–6010. doi: 10.48550/arXiv.1706.03762.

[37] A. van den Oord *et al.*, "WaveNet: A Generative Model for Raw Audio," pp. 1–15, 2016. arXiv preprint: 1609.03499.

[38] T. Karras, S. Laine, and T. Aila, "A Style-Based Generator Architecture for Generative Adversarial Networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 12, pp. 4217–4228, 2021. doi: 10.1109/TPAMI.2020.2970919.

[39] J. Jumper *et al.*, "Highly accurate protein structure prediction with AlphaFold," *Nature*, vol. 596, no. 7873, pp. 583–589, 2021. doi: 10.1038/s41586-021-03819-2.

[40] G. Carleo *et al.*, "Machine learning and the physical sciences," *Reviews of Modern Physics*, vol. 91, no. 4, p. 45002, 2019. doi: 10.1103/RevModPhys.91.045002.

[41] E. Cuoco *et al.*, "Enhancing gravitational-wave science with machine learning," *Machine Learning: Science and Technology*, vol. 2, no. 1, p. 011002, 2021. doi: 10.1088/2632-2153/abb93a.

[42] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, 1943. doi: 10.1007/bf02478259.

[43] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain." *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958. doi: 10.1037/h0042519.

[44] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems.* O'Reilly Media, 2019, ISBN: 9781492032649.

[45] M. Minsky and S. Papert, *Perceptrons; an Introduction to Computational Geometry*, 1969, ISBN: 9780262130431.

[46] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989. doi: 10.1016/0893-6080(89)90020-8.

[47] J. Howard and S. Gugger, *Deep Learning for Coders with Fastai and Pytorch: AI Applications Without a PhD.* O'Reilly Media, 2020, ISBN: 9781492045526.

[48] B. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964. doi: 10.1016/0041-5553(64)90137-5.

[49] D. Rumelhart, G. Hinton, and R. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. doi: 10.1038/323533a0.

[50] M. Nielsen, *Neural Networks and Deep Learning.* Determination Press, 2015. url: http://neuralnetworksanddeeplearning.com.

[51] Y. LeCun *et al.*, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989. doi: 10.1162/neco.1989.1.4.541.

[52] Y. LeCun *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. doi: 10.1109/5.726791.

[53] K. He *et al.*, "Deep residual learning for image recognition," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016. doi: 10.1109/CVPR.2016.90.

[54] C. Szegedy *et al.*, "Inception-v4, Inception-ResNet and the impact of residual connections on learning," pp. 4278–4284, 2017. arXiv preprint: 1602.07261.

[55] M. Tan and Q. Le, "EfficientNetV2: Smaller Models and Faster Training," 2021. arXiv preprint: 2104.00298.

[56] Z. Liu *et al.*, "A ConvNet for the 2020s," 2022. arXiv preprint: 2201.03545.

[57] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *3rd International Conference on Learning Representations*, pp. 1–14, 2014. arXiv preprint: 1409.1556.

[58] H. Li *et al.*, "Visualizing the loss landscape of neural nets," *Advances in Neural Information Processing Systems*, pp. 6389–6399, 2018. arXiv preprint: 1712.09913.

[59] A. Veit, M. Wilber, and S. Belongie, "Residual networks behave like ensembles of relatively shallow networks," *Advances in Neural Information Processing Systems*, pp. 550–558, 2016. arXiv preprint: 1605.06431.

[60] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, vol. 9, 2010, pp. 249–256. url: https://proceedings.mlr.press/v9/glorot10a.html.

[61] X. Glorot, A. Bordes, and Y. Bengio, "Deep Sparse Rectifier Neural Networks," in *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, vol. 15, 2011, pp. 315–323. url: https://proceedings.mlr.press/v15/glorot11a.html.

[62] K. He *et al.*, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2015 Inter, pp. 1026–1034, 2015. doi: 10.1109/ICCV.2015.123.

[63] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *32nd International Conference on Machine Learning*, vol. 1, pp. 448–456, 2015. arXiv preprint: 1502.03167.

[64] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *3rd International Conference on Learning Representations*, pp. 1–15, 2015. arXiv preprint: 1412.6980.

[65] A. Burkov, *The Hundred-Page Machine Learning Book*. Andriy Burkov, 2019, ISBN: 9781999579500.

[66] J. Bergstra and Y. Bengio, "Random Search for Hyper-Parameter Optimization," *Journal of Machine Learning Research*, vol. 13, pp. 281–305, 2012. url: https://dl.acm.org/doi/10.5555/2188385.2188395.

[67] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," *Advances in Neural Information Processing Systems*, vol. 4, pp. 2951–2959, 2012. arXiv preprint: 1206.2944.

[68] J. Howard and S. Gugger, "Fastai: A layered api for deep learning," *Information (Switzerland)*, vol. 11, no. 2, pp. 1–26, 2020. doi: 10.3390/info11020108.

[69] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, 2019, pp. 8024–8035. url: https://dl.acm.org/doi/10.5555/3454287.3455008.

[70] L. Smith, "A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay," pp. 1–21, 2018. arXiv preprint: 1803.09820.

[71] L. Smith and N. Topin, "Super-convergence: very fast training of neural networks using large learning rates," in *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, 2019, p. 36. doi: 10.1117/12.2520589.

[72] P. Micikevicius *et al.*, "Mixed Precision Training," *6th International Conference on Learning Representations*, pp. 1–12, 2017. arXiv preprint: 1710.03740.

[73] F. Zhuang *et al.*, "A Comprehensive Survey on Transfer Learning," *Proceedings of the IEEE*, vol. 109, no. 1, pp. 43–76, 2021. doi: 10.1109/JPROC.2020.3004555.

[74] R. Wightman, "PyTorch Image Models," 2019, doi: 10.5281/zenodo.4414861.

[75] S. Bahaadini *et al.*, "Machine learning for Gravity Spy: Glitch classification and dataset," *Information Sciences*, vol. 444, pp. 172–186, 2018. doi: 10.1016/j.ins.2018.02.068.

[76] M. Zevin *et al.*, "Gravity Spy: Integrating advanced LIGO detector characterization, machine learning, and citizen science," *Classical and Quantum Gravity*, vol. 34, no. 6, 2017. doi: 10.1088/1361-6382/aa5cea.

[77] S. Bahaadini *et al.*, "Deep multi-view models for glitch classification," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2017, pp. 2931–2935. doi: 10.1109/ICASSP.2017.7952693.

[78] D. George, H. Shen, and E. Huerta, "Deep Transfer Learning: A new deep learning glitch classification method for advanced LIGO," 2017. arXiv preprint: 1706.07446.

[79] "NVIDIA RTX A4000 Graphics Card," https://www.nvidia.com/en-us/design-visualization/rtx-a4000/, Accessed: 29/07/2022.

[80] Y. Cui *et al.*, "Class-balanced loss based on effective number of samples," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2019-June, pp. 9260–9269, 2019. doi: 10.1109/CVPR.2019.00949.

[81] T. Lin *et al.*, "Focal Loss for Dense Object Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 2, pp. 318–327, 2020. doi: 10.1109/TPAMI.2018.2858826.

[82] K. Pasupa, S. Vatathanavaro, and S. Tungjitnob, "Convolutional neural networks based focal loss for class imbalance problem: a case study of canine red blood cells morphology classification," *Journal of Ambient Intelligence and Humanized Computing*, 2020. doi: 10.1007/s12652-020-01773-x.

[83] A. Dosovitskiy *et al.*, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," 2020. arXiv preprint: 2010.11929.

[84] Z. Liu *et al.*, "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows," *Proceedings of the IEEE International Conference on Computer Vision*, pp. 9992–10 002, 2021. doi: 10.1109/ICCV48922.2021.00986.

[85] J. Howard, "The best vision models for fine-tuning," https://www.kaggle.com/code/jhoward/the-best-vision-models-for-fine-tuning/notebook, Accessed: 17/07/2022.

[86] S. Raschka, "Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning," 2018. arXiv preprint: 1811.12808.

[87] D. George, H. Shen, and E. Huerta, "Classification and unsupervised clustering of LIGO data with Deep Transfer Learning," *Physical Review D*, vol. 97, no. 10, p. 101501, 2018. doi: 10.1103/PhysRevD.97.101501.

[88] The LIGO Scientific Collaboration, the Virgo Collaboration *et al.*, "Open data from the first and second observing runs of Advanced LIGO and Advanced Virgo," *SoftwareX*, vol. 13, 2021. doi: 10.1016/j.softx.2021.100658.

[89] D. Macleod *et al.*, "GWpy: A Python package for gravitational-wave astrophysics," *SoftwareX*, vol. 13, p. 100657, 2021. doi: 10.1016/j.softx.2021.100657.

[90] The LIGO Scientific Collaboration, the Virgo Collaboration *et al.*, "GWTC-1: A Gravitational-Wave Transient Catalog of Compact Binary Mergers Observed by LIGO and Virgo during the First and Second Observing Runs," *Physical Review X*, vol. 9, no. 3, p. 31040, 2019. doi: 10.1103/PhysRevX.9.031040.

[91] J. Yosinski *et al.*, "How transferable are features in deep neural networks?" in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, 2014, pp. 3320–3328. doi: 10.48550/arXiv.1411.1792.

[92] F. Robinet *et al.*, "Omicron: A tool to characterize transient noise in gravitational-wave detectors," *SoftwareX*, vol. 12, p. 100620, 2020. doi: 10.1016/j.softx.2020.100620.

[93] I. Goodfellow *et al.*, "Generative adversarial networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020. doi: 10.1145/3422622.

[94] D. Hendrycks and K. Gimpel, "Gaussian Error Linear Units (GELUs)," 2016. arXiv preprint: 1606.08415.

[95] J. Ba, J. Kiros, and G. Hinton, "Layer Normalization," 2016. arXiv preprint: 1607.06450.