Data Ad Infinitum

Bootstrapping Gravitational-Wave Data Science with Machine Learning

Michael R K Norman

Contents

1 Introduction

We live in an exciting time. Now is a time of remarkable confluence and innovation; machine learning and artificial intelligence stand poised to remake almost every aspect of science, technology, and everyday life. At the same time, our understanding of the world is increasing with ever more fervour. The era of gravitational-wave astronomy has well and truly begun, and bountiful amounts of cosmological data will soon be at our fingertips. The machine learning community is unlocking novel new tools every month, and it would be an odious missed opportunity not to turn some of these powerful techniques onto the data hidden beneath interferometer noise.

2 Gravitational Waves

Since time immemorial, humanity has gazed at the stars. With wonder rooted deep in their minds, they imagined strange and divine mechanisms in order to try and make sense of what they saw. Over the millennium, the vast skies have revealed much about their working, and with ever-improving tools, we have come much closer to understanding their mysteries, but there is still much to be uncovered. It is unclear exactly how deep the truth lies. Perhaps we have, but only scratched at the surface. The depths are so vast we simply do not know.

Almost all of that knowledge, all of that understanding and academia has been built upon the observation of just a single type of particle. Until very recently, the only information we had about the world above us came from light, and although the humble photon has taught us a great deal about the scope of our universe, the discovery of new messengers promises pristine, untapped wells of science. It has only been in the last century that we have achieved the technological prowess to detect any other extraterrestrial sources of data except that which fell to us as meteors. We have brought rocks home from the moon. We study data sent back from probes shot out across the solar system and even ones that have peaked just beyond the sun's mighty sphere of influence. We have seen neutrinos, tiny, almost massless particles that pass through entire planets more easily than birds through the air, and, most recently of all, we have seen the skin of space itself quiver — gravitational waves, the newest frontier in astronomy.

Practical gravitational-wave astronomy is in its infancy; compared to the other fields of astrophysics, it has barely left the cradle, with the first confirmed gravitational-wave detection occurring in 2015 [1]. Although the field has progressed remarkably quickly since its inception, there is still a lot of work to be done — a lot of groundwork to be performed whilst we uncover the best ways to deal with the influx of new data that we are presented with. It seems likely, assuming both funding and human civilisation prevail, that work undertaken now will be but the first bricks in a great wall of discovery. New gravitational-wave observatories that are even today being planned will increase our sensitive range by orders of magnitude. With any luck, they will open our ears further to previously undiscovered wonders.

This chapter will introduce a small part of the science of gravitational waves; it will not be an extensive review as many of the particularities are not especially relevant to the majority of the content of this thesis. Instead, this section aims to act as a brief overview to give context to the purpose behind the data-analysis methods presented throughout. We will cover the theoretical underpinning of gravitational waves, as well as a glancing tour through the experiments used to detect them.

2.1 Gravity — A Brief History of Down

Gravity is one of the four fundamental forces of nature, the other three being the electromagnetic force and the strong and weak nuclear forces. It is, in some ways, the black sheep of the interactions, as it is the only one not explained by the standard model of particle physics, which just so happens to be the most accurate theory of physics ever created by human hands (and brains). Luckily, gravity has its own extremely accurate descriptive theory, even if it doesn't like to play well with the others. It has a storied history, which, if you are unfamiliar, is worth skimming for context.

This thesis is written with the intent to be at least somewhat passable to all; my dad is attempting to read it, but this next subsection is mostly for context, and if you're primarily focused on examining the author, feel free to skip to more relevant time, Section 2.1.2. I fear the page count may have got somewhat out of hand by the time I'm finished, so I understand your plight.

2.1.1 Ye Old Times

In the beginning, men threw rocks at each other and were entirely unsurprised when they hit the floor. Over time, people became more and more confused as to why this was the case. Many philosophers proposed many reasons why one direction should be preferred over all others when it came to the unrestrained motion of an object. For a long time, there was much confusion about the relationship between mass, density, and buoyancy, and the nature and position of various celestial objects. Still, sometime after, we had decided that the Earth was not, in fact, at the centre of the universe and that objects fell at the same speed irrespective of their densities. We reached the time of Sir Issac Newton, and along with him arrived what many would argue was the beginning of modern physics.

The idea of gravity as a concept had been around for many thousands of years at this point, but what Newton did was to formalise the rules by which objects behave under the action of gravity. Newton's universal law of gravitation states that all massive objects in the universe attract all others [2], acting upon each other, much to his displeasure, whether surrounded by a medium or not. Gravity appeared to ignore all boundaries and was described in a simple formula that seemed to correctly predict everything from the motion of the planets (mostly) to the fall of an apple

$$F = \frac{Gm_1m_2}{r^2}. \tag{2.1}$$

where F is the force along the direction between the two masses, G is the gravitational constant equal to $6.67430(15) \times 10^{-1}$, m_1 is the mass of the first object, m_2 is the mass of the second object, and r is the distance between the two objects. Newton's law of universal gravitation describes the force every massive object in the universe experiences because of every other — an equal and opposite force proportional to the product of their two masses; see . Though we now know this equation to be an imperfect description of reality, it still holds accurate enough for many applications to this day.

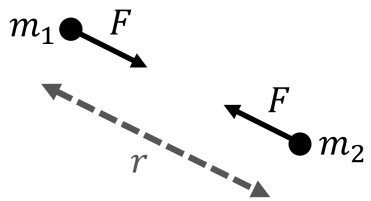


Figure 1 | An illustration of Newton's law of universal gravitation, as described by Equation 1. Two particles, here distinguished by their unique masses m_1 and m_2 are separated by a distance, r, are pulled toward each other by the force of gravity acting on each object F, each being pulled directly toward the other by a force that is equal and opposite.

It was also around this time that a fierce debate raged over the nature of space and time. Newton proposed a universal time that ticked whether in the presence of a clock or not and a static, ever-present grid of space that never wavered nor wandered. Both would continue to exist whether the rest of the universe was there or not. Leibniz, on the other hand, argued that space and time were little more than the relations between the positions of objects and sequences of events. If there were no objects, there would be nothing to measure, and there would be no space. If there were no events, there would be nothing to time, and there would be no ticks; see . At the time, they did not come to a resolution, but as we will see, the truth lies somewhere in between.

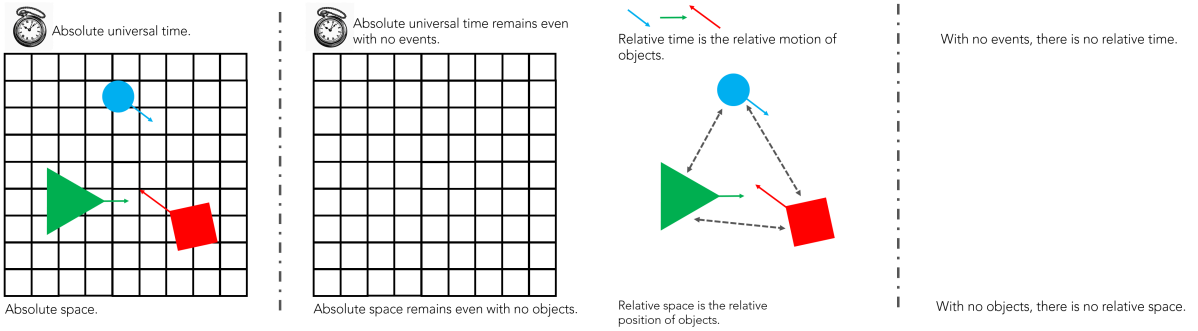


Figure 2 | An illustration of two competing historical views on the nature of space and time. *Left:* Newton's vision of absolute universal time and absolute space, wherein time moves forward at a constant and uniform rate across the universe and space is immobile and uniform. In this model, both time and space can exist independently of objects within, even in an entirely empty universe. *Right:* Leibniz's view proposed a wherein time and space did not and could not exist independently of the objects used to measure them. Within this model, space is simply a measure of the relative distances between objects, and time is a measure of the relative motion of objects as their relative positions change. In this model, it makes little sense to talk of a universe without objects since time and space do not exist without objects to hold relative positions and velocities.

For a good few centuries, Newton's law of universal gravitation stood as our fundamental understanding of gravity, with its impressive descriptive and predictive power. As our measurements of the solar system became more precise, however, a major discrepancy was noted, one that Newton's law failed to describe. The planet Mercury, so close to the sun and so heavily pulled by its gravity, was found to be behaving ever so slightly strangely. Under Newton's laws, the orbits of the planets should have been ellipses fixed in space, their paths never deviating; what was observed, however, was a precession of that ellipse over time it rotated, processing around the sun by the barest fractions of a degree per century. If true, which it was, the difference was enough to state with confidence that Newton's universal law of gravitation was not a universal and complete description of gravity.

2.1.2 General Relativity

By the start of the 19th century, two more thorns in Newton's side had been revealed. Experiments failed to detect a change in the speed of light irrespective of the Earth's motion through space — if light behaved as we might expect from ordinary matter, then its measured speed should change depending on whether we are moving toward its source, and hence in opposition to its own direction of motion, or against and in unison. That is not what was observed. Light moved at the same speed no matter how fast you were going when you measured it relative to its source or any other point in the universe. There was no explanation for this behaviour under Newtonian mechanics. The second tantalising contraction arrived when attempting to apply Maxwell's hugely successful equations describing electromagnetism, which proved incompatible with Newtonian mechanics, again in large part because of the requirement for a constant speed of light at all reference frames.

In 1905, Einstein proposed his theory of Special Relativity as an extension beyond standard Newtonian mechanics in a successful attempt to rectify the previously mentioned shortcomings.

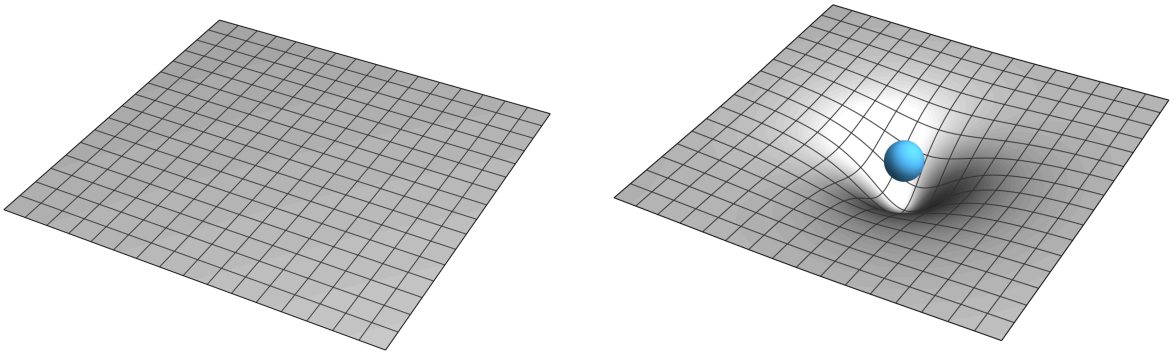


Figure 3 | Two illustrative depictions of Einsteins's spacetime. For illustrative purposes, since we are not 4D beings and the paper on which this will be printed very much isn't, the four dimensions of our universe have been compacted down into two. It should also be noted that these illustrations were not generated with correct physical mathematics but only to give an impression of the concepts being described. *Left:* Minkowski space — in the absence of any mass, spacetime will not experience any curvature. This is the special case that Einstien's special relativity describes. If we were to place a particle into this environment, it would not experience any acceleration due to gravity. If the particle itself were massive, it would create a distortion in the spacetime, and the spacetime would no longer be considered Minkowski space even though alone it would not experience any acceleration. Often, when dealing with particles of low mass, their effects on the distortion of spacetime are ignored, and we can still treat the scenario with special relativity. *Right:* Spactime distorted by a massive object, shown in blue. Curved space is described by Einstein's more general theory, General Relativity. In this scenario, we can see how the presence of mass imprints a distortion into the shape of spacetime. Any particles also present in the same universe as the blue object, assuming it has existed indefinitely, will experience an apparent acceleration in the direction of the blue sphere. A beam of light, for example, comprised of photons and entirely massless, would deflect its path when moving past the sphere. Even though light will always travel in a straight line through the vacuum of space, the space itself is distorted; therefore, a straight line path will manifest itself as an apparent attraction toward the sphere. Notice that using Newton's universal law of gravitation, Equation 1, the mass of the photon is zero; therefore, it should not experience any gravitational attraction, and indeed, gravitational lensing of the passage of starlight, as it moves past the sun, was one of the first confirmations of Einstein's theory of General Relativity. Were this sphere several thousand kilometres in diameter, any lifeforms living on its surface, which would appear essentially flat at small scales, would experience a pervasive and everpresent downward force. Note that the mass of the object is distributed throughout its volume, so in regions near the centre of the sphere, the spacetime can appear quite flat, as equal amounts of mass surround it from all directions.

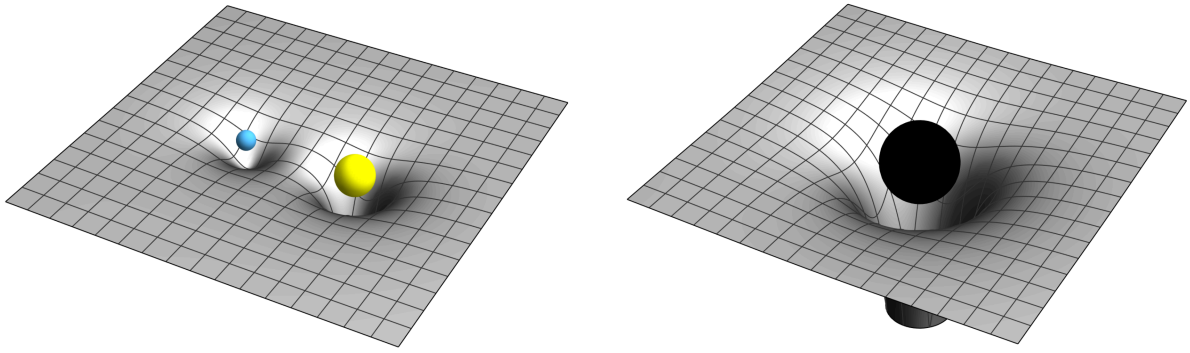


Figure 4 | Two further depictions of spacetime. Again, these images are a 2D representation of the 4D spacetime, and they were generated without correct physical descriptions but for illustrative purposes alone. *Left:* Two objects, one in blue with a lesser mass and one in yellow with a greater mass. Objects with a larger mass distort spacetime to a greater extent, so objects close to the yellow sphere will experience a stronger apparent acceleration as the space curves and the objects continue to move on a straight line. In this scenario, if stationary, the yellow and blue objects will accelerate and move toward each other and, without outside interference, inevitably collide. However, if either the blue or yellow ball is given an initial velocity perpendicular to the direction of the other sphere so that its straight-line path orbits the other sphere, they can remain equidistant from each other in a stable orbit for potentially very long periods of time. As we will see, this orbit will eventually lose energy and decay, but depending on the masses of the two objects, this could take an extremely long time. *Right:* A black hole. Thus far, we have assumed that the mass of the yellow and blue objects are evenly distributed through their volume, so the spacetime at the very centre of the object is, at its limit, entirely flat. In many scenarios, this is a physically possible arrangement of matter, as although gravity pulls on every particle within the object, pulling it toward the centre, it is a very weak pull compared to the other forces of nature, which push back out and stop the particles continuing on their naturally preferred trajectory. This prevents a complete collapse of the object. Gravity, however, has one advantage on its side, and that is that there is no negative mass, only positive, so whereas large bodies tend to be electrically neutral as positive and negative charges cancel each other out, gravity always grows stronger. If enough mass congregates in the same place, or if the forces pushing matter away from the centre stop, there's nothing to stop gravity from pulling every particle in that object right to the centre, right into a singular point of mass with infinite density known as the singularity. As this collapse occurs, the curvature of spacetime surrounding the object gets stronger and stronger, eventually reaching the point where within a region around the singularity, known as the event horizon, all straight-line paths point toward the singularity. Meaning that no matter your speed, no matter your acceleration, you cannot escape, even if you are light itself. Consequently, no information can ever leave the event horizon, and anything within is forever censored from the rest of the universe.

2.2 Orbits are Not Forever

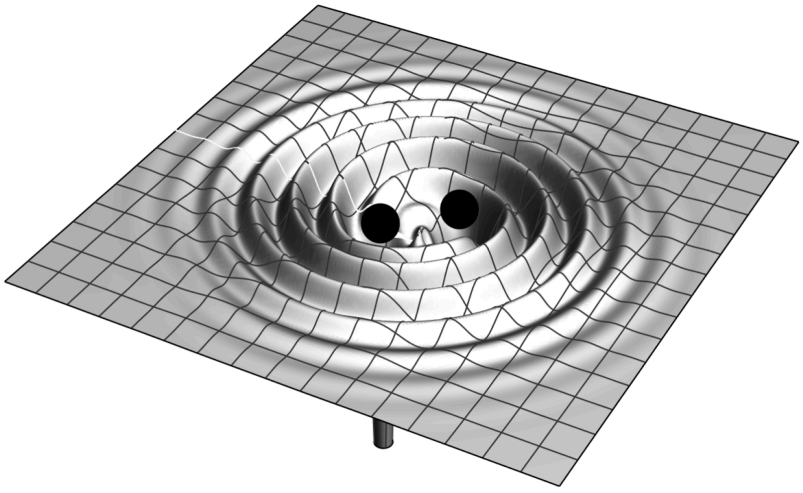


Figure 5 | A depiction of the region of spacetime surrounding two inspiraling black holes. The spacetime grid visible is a 2D representation of the true 4D nature of our universe as described by general relativity. This depiction was not produced by an accurate simulation but was constructed as a visual aid alone. Two massive objects can orbit each other if they have sufficient perpendicular velocity to

each other; this is a natural state for objects to find themselves trapped in because the chances of direct collisions between objects are low, and any objects that find themselves gravitationally bound together and do not experience a direct collision will eventuate in an orbit. The same is true for black holes; whether they form from pairs of massive stars that both evolve into black holes after the end of their main sequence lives or whether they form separately and through dynamical interaction, end up adjoined and inseparable, the occurrence of two black holes orbiting is not inconceivable. Over time, small amounts of energy will leak from these binaries; ripples are sent out through the cosmos, carrying energy away from the system and gradually reducing the separation between the companions. As they get closer, the curvature of the spacetime they occupy increases and thus, their acceleration toward each other grows. They speed up, and the amount of energy that is lost through gravitational radiation increases, further increasing the speed of their inspiral in an ever-accelerating dance. If they started out just close enough, this process would be enough to merge them within the lifetime of the universe; they will inevitably collide with an incredible release of energy out through spacetime as powerful gravitational waves. It is these waves, these disturbances in the nature of length and time itself, that we can measure here on Earth using gravitational wave observatories.

2.3 Detecting Gravity

In order to sense disturbances in the force ...

2.3.1 Interfereometer Noise

3 Machine Learning

Machine learning techniques can be applied to almost any area of gravitational-wave data science; therefore, an exhaustive list would be difficult to compile and quickly outdated. However, here are some current areas of investigation: transient detection and parameterisation, including compact binary coalesces, bursts, and detector glitches; continuous waveform detection and parameterisation; stochastic background detection and parameterisation; detector noise characterisation and cleaning; detector calibration; and approximant generation. This thesis will focus on the application of machine learning to transients, including compact binary coalesces and other burst events. To contextualise this research, this chapter will serve as a brief introduction to machine learning.

Many ambiguous, sometimes contradictory definitions exist within machine learning and artificial intelligence. The definitions used throughout this thesis will be discussed here, attempting to use the most technically correct, or failing that, most commonly used definitions available.

Artificial Intelligence is perhaps the broadest of the terms associated with machine learning and perhaps also the vaguest. It has various definitions but is often defined as a property of human-designed intelligent agents — systems that take, as input, information about the world and process that data, along with any internal state, to produce an output that maximises the chance of achieving a specific goal [3]. This broad definition can be applied to an extensive range of artificial devices, from a toaster, which takes as an input the twist of a dial and tries to maximise its goal of applying heat for an amount of time relating to the position of the dial, to a chess engine with the all-consuming goal of checkmating its opponent. Most people would probably not consider a toaster artificially intelligent, and indeed, in the years since DeepBlue first defeated Garry Kasparov [4], many have come to consider chess engines in much the same light. This phenomenon is known as the 'A.I. effect', wherein a task is only considered something requiring intelligence until a machine has successfully performed it [5]. At that point, it is pushed out of the realm of intellectual endeavour and into the mundane, therefore preserving human supremacy over their cognitive dominion. I fear that with the rise of large language models, a few years is all that separates the act of writing a thesis such as this from the same relegation [6]. This transience can make artificial intelligence a tricky definition to use in a technical sense, so the term will, where possible, be avoided.

Machine Learning is somewhat easier to define. Depending on your definition of artificial intelligence, it could be considered either a subset of that field or merely at an intersection [7]. It is loosely defined as the study of agents who can gain competency at a task without explicit human instruction [8]. This is achieved through the use of specialised algorithms and statistical methods [8]. Since, for the context of this thesis, it is probably more helpful to think of these agents as statistical techniques rather than actors that react to the world, the rest of this thesis will use the term **model** to refer to these agents, as they often model the relationship between a specific distribution of input data and a specific distribution of output data.

Machine learning can be subdivided in multiple ways, but one of the most common distinctions separates it into three basic paradigms: supervised learning, unsupervised learning, and reinforcement learning [9].

Supervised Learning refers to any machine learning task wherein the model attempts to match its outputs with preexisting values labelled by humans or another technique [9]. Training a model through supervised learning requires datasets of labelled training data from which the models learn. After which, if successful, the models should be able to approximate the desired output given new unseen input data.

Unsupervised learning, on the other hand, does not provide the model with any preexisting values to attempt to match its outputs with [9]. This can include techniques that use the input data as the

desired output data, such as in autoencoders, or techniques that attempt to divine patterns within the dataset previously unknown to the model and, often, the model user. For example, clustering tasks look for similar latent features between groups of training examples.

Semi-supervised learning lies, perhaps unsurprisingly, in the lacuna between supervised and unsupervised learning [10]. Whilst training under this paradigm, some of the training data is labelled and some unlabeled. This can be used when the labels are too computationally expensive to compute for the entirety of the training dataset or when some of the labels are intractable by other techniques or simply unknown.

Reinforcement Learning is a paradigm based on slightly different principles. Instead of using extensive data sets to train an agent, reinforcement learning utilises algorithms that try to maximise themselves against an externally defined reward function [9]. While training a model using reinforcement learning, the model can take actions that affect the state of the environment in which the model is allowed to act. The state of its environment will then be mapped to a score; this score is used to update the model. Through an iterative process, the model is updated to improve its ability to maximise the score of its environment.

Reinforcement learning is commonly used in scenarios where huge training datasets are not available, and the model is primarily designed to interact with an environment (virtual or real), such as training a robot to walk [11] or a virtual race car to drive around a virtual track [12]. Though this has proved a powerful technique for many machine learning applications, it has not been investigated in this thesis and thus will not be discussed in detail.

3.1 The Artificial Neural Network

The Artificial Neural Network is a machine-learning technique that has seen rapid innovation, development, and adoption over the last decade [13]. They've shown the ability to solve many long-standing problems in artificial intelligence, including image, audio, and text classification, captioning, and generation, [14], [15], [16], [17], [18], [19], [20], [21], [22] , as well as game-playing algorithms that have attained superhuman performance in previously human-superior games like Go [23]. They can teach themselves the rules from scratch in a matter of hours [24] — compared to the many years of development required for previous game-playing engines. They can compete in complex, highly-dimensional computer games like Starcraft 2 [25] and League of Legends [26]. They have achieved large-scale adoption across many industrial sectors, managing power grids [27], performing quality control [28], and paving the way, albeit slowly, toward fully autonomous self-driving cars [29]. Artificial neural networks have also been applied to many scientific problems, such as AlphaFold [30], solving the protein folding problem.

With their rampant and rapid success across many domains previously thought intractable or at least many decades away from a solution, it is easy to ascribe to artificial neural networks more than what they are, but it is also easy to underestimate their potential to solve previously unthinkable problems. Artificial neural networks are little more than simple statistical structures compiled into complex architectures, which allow them to perform intricate tasks.

They are loosely inspired by the structures of biological neurons inside animal brains [31]. Although they indeed show a greater likeness to the workings of biological systems than most computers, this analogy should not be taken too literally. Biological brains are far more complex than current artificial neural networks, and there is much about them we do not yet understand. There may still be something missing from state-of-the-art models that prevents them from the full range of computation available to a biological brain. Having said that, there are still ample further developments that can be made with artificial neural networks, even considering their possible limits. We do not yet seem close to unlocking their full potential.

Deep Learning is yet another buzzword that describes any artificial neural network with more than one layer, or in other words, one or more hidden layers. In practice, almost all contemporary applications of artificial neural networks are more than one layer deep, so the distinction is not particularly meaningful. The hierarchical relationship between A.I. and machine learning is illustrated by Figure 6.

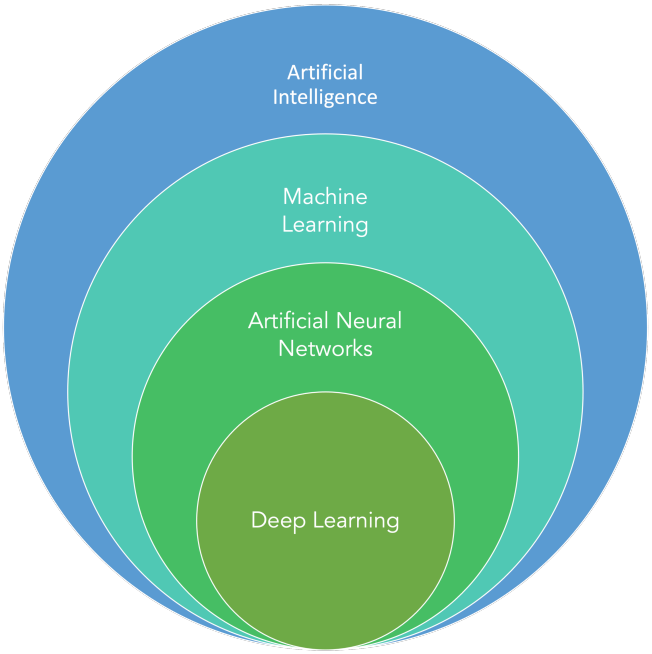


Figure 6 | The loose hierarchical relationship between different umbrella terms used in artificial intelligence.

There are a plethora of different types and arrangements of artificial neural networks, often known as architectures. The following sections will introduce the main concepts surrounding artificial neural networks.

3.1.1 The Artificial Neuron

As mentioned previously, artificial neural networks are loosely inspired by biological neural networks, and as one might expect, their base unit is analogous to the biological base unit, the neuron. Artificial neurons form the basic building block of all artificial neural networks, though their form and design can vary between architectures.

The artificial neuron takes a number, N, of continuous numerical inputs $\vec{x} = [x_1, ...x_i, ...x_N]$ and outputs a single numerical output $A(\vec{x})$. Each neuron has a number of tunable parameters associated with it, $\vec{\theta}$. A single neuron has many weight values $\vec{w} = [w_1, ...w_i, ...w_N]$ and a single bias value b. Suppose these parameters, $\vec{\theta}$, are selected correctly. In that case, the artificial neuron can, in some simple cases, act as a binary classifier that can correctly sort input vectors, $\vec{x}$, drawn from a limited distribution into two classes. This kind of single-neuron classifier is often known as a perceptron.

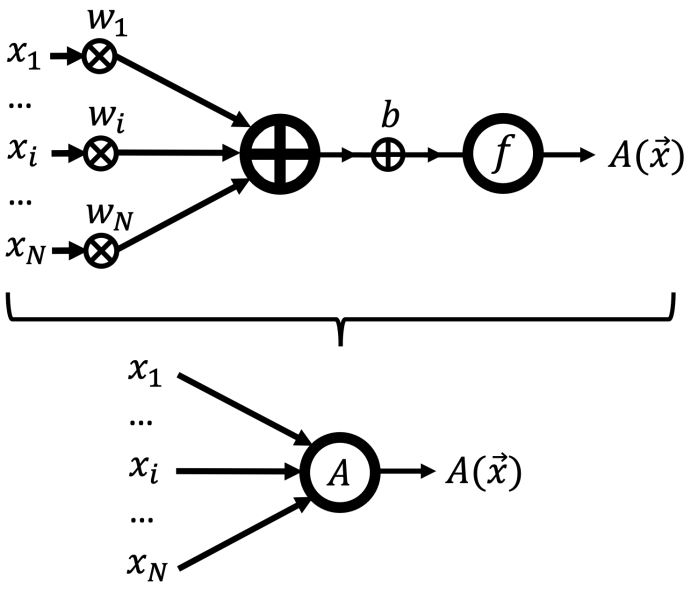


Figure 7 | *Upper*: The Artificial Neuron or Perceptron. This figure illustrates the operations that compose the archetypical artificial neuron, where $\vec{x}$ is the input vector, f is the activation function, $\vec{w}$ is the weights vector, and b is the neuron bias. An artificial neuron takes an input vector, $\vec{x}$, and performs some useful calculations (hopefully). Both the weights vector, $\vec{w}$, and bias value, b, comprise the neuron's adjustable parameters, $\vec{\theta}$, that must be tuned for the neuron to perform any useful operations. *Note*: During computation, the bias, b, is not normally added in a separate operation; instead, it is added as an extra x_0 term included in the same calculation as the summation of the product of the weights, $\vec{w}$, and input values, $\vec{x}$. *Lower*: An abstraction of the more complicated interior structure of the artificial neuron. Abstraction is common and necessary when describing artificial neural networks as networks are often comprised of thousands if not millions of artificial neurons.

As can be seen in Figure 7, the standard artificial neuron is comprised of several consecutive mathematical operations. First, the input vector, $\vec{x}$, is multiplied by the weights vector, $\vec{w}$, and then the result of this operation is summed along with the bias value, b. Finally, the output is then fed into an activation function f; see Section 3.1.5. This sequence of operations is given by:

$$A(\vec{x}) = f\left(\sum_{i=1}^{N} w_i x_i + b\right) = f(\vec{x} \cdot \vec{w} + b), \tag{3.2}$$

where N is the number of elements in the input vector. In the case of the single-layer perceptron, the output of the neuron, $A(\vec{x})$, is equivalent to the output of the perceptron, $\hat{y}$, where our desired ground-truth output value is y. Since each element of the weights vector, $\vec{w}$, is multiplied by each component of the input vector, $\vec{x}$, the weights can be thought of as representing the significance of their corresponding input value, x_i, to the desired output value, y. The bias, b, acts as a linear shift to the activation function, and tuning this value can make it more or less difficult for the neuron to activate. Having well-tuned parameters, $\vec{\theta}$, is crucial for the performance of the artificial neuron.

The purpose of the activation function, f, is to coerce the distribution of the output value, $A(\vec{x})$, into a particular shape. The intricacies of why you might want to do this will not become apparent until the model training is understood. Therefore a more detailed discussion of activation functions follows in Section 3.1.5.

3.1.2 Training Artificial Neurons

Now that the structure of the artificial neuron has been described, the question becomes, how does one go about ascertaining useful values for the neuron's tunable parameters, $\vec{\theta}$, namely the weights vector, $\vec{w}$, and the bias, b. It would, in theory, be possible to approach this problem by manually discovering values for each parameter, θ_i, using human-guided trial and error. Whilst this would be unwise, we can use this thought experiment to arrive at the automated solution to the problem. This section will describe the step-by-step process of training an artificial neuron, or in this case, multiple neurons, and for each step, illustrate how the manual approach can be automated, displaying a Python [32] function demonstrating this. Listing 1 shows the required library imports to run all subsequent code listings in this section. An ipython notebook containing the described code can be found here: https://colab.research.google.com/github/mrknorman/data_ad_infinitum/blob/main/chapter_x_single_layer_perceptron.ipynb.

```
# Importing necessary libraries
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from bokeh.plotting import figure, show
from bokeh.io import output_notebook
```

Listing 1 | *Python [32]* . Required imports to run subsequent code listings in this section. NumPy [33] is used for its fast numerical CPU operations. TensorFlow [34] is used for fast numerical GPU operations, machine learning functionality, and loading the Modified National Institute of Standards and Technology (MNIST) dataset [35]. Bokeh [36] is used to plot figures.

We will attempt to train an ensemble of ten artificial neurons to classify the Modified National Institute of Standards and Technology (MNIST) example dataset [35] correctly. The MNIST dataset consists of 70,000 black-and-white images of handwritten numbers with a resolution of 28 by 28. Pixel values range from 0 for black pixels to 255 for white pixels, with the integer values representing 253 shades of grey. 10,000 images are reserved for testing, with the remaining 60,000 used for training. See Figure 8 for examples of the images contained within the dataset.

Though slightly confusing, this ensemble of multiple neurons is often known as a single-layer perceptron, as it consists of many neurons acting (almost) independently in a single layer; see Figure 9. The only collaboration between neurons is the normalisation that is applied to each neuron by the softmax activation function, which ensures the produced output vector sums to one and can act as a probability; see Section 3.1.5.4. Because we are moving from a single neuron with one bias value b, and a vector of weights values $\vec{w}$, to multiple neurons, the bias value becomes a vector $\vec{b}$, and the weights vector becomes a matrix W. W is defined by

$$W = \begin{pmatrix} w_{1,1} & \cdots & w_{1,j} & \cdots & w_{1,P} \\ \vdots & \ddots & \vdots & \iddots & \vdots \\ w_{i,1} & \cdots & w_{i,j} & \cdots & w_{i,P} \\ \vdots & \iddots & \vdots & \ddots & \vdots \\ w_{N,1} & \cdots & w_{N,j} & \cdots & w_{N,P} \end{pmatrix}, \tag{3.3}$$

where N is the number of neurons in the layer, and P is the number of weights per neuron, typically determined by the number of neurons in the previous layer or the number of elements in the input vector if the layer is the input layer.

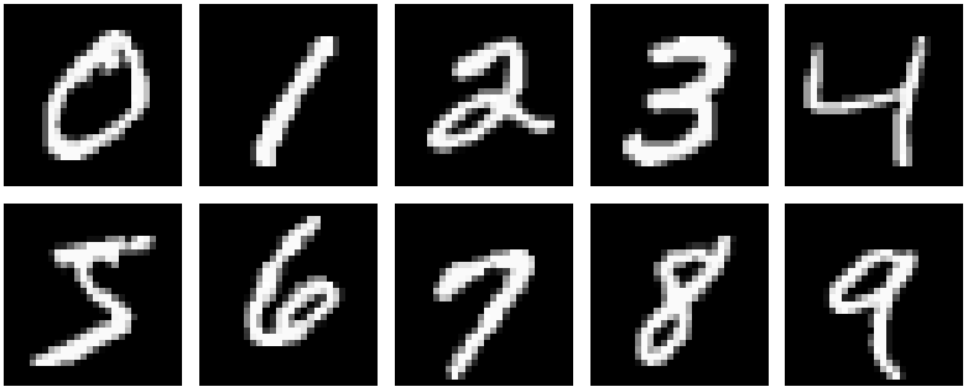
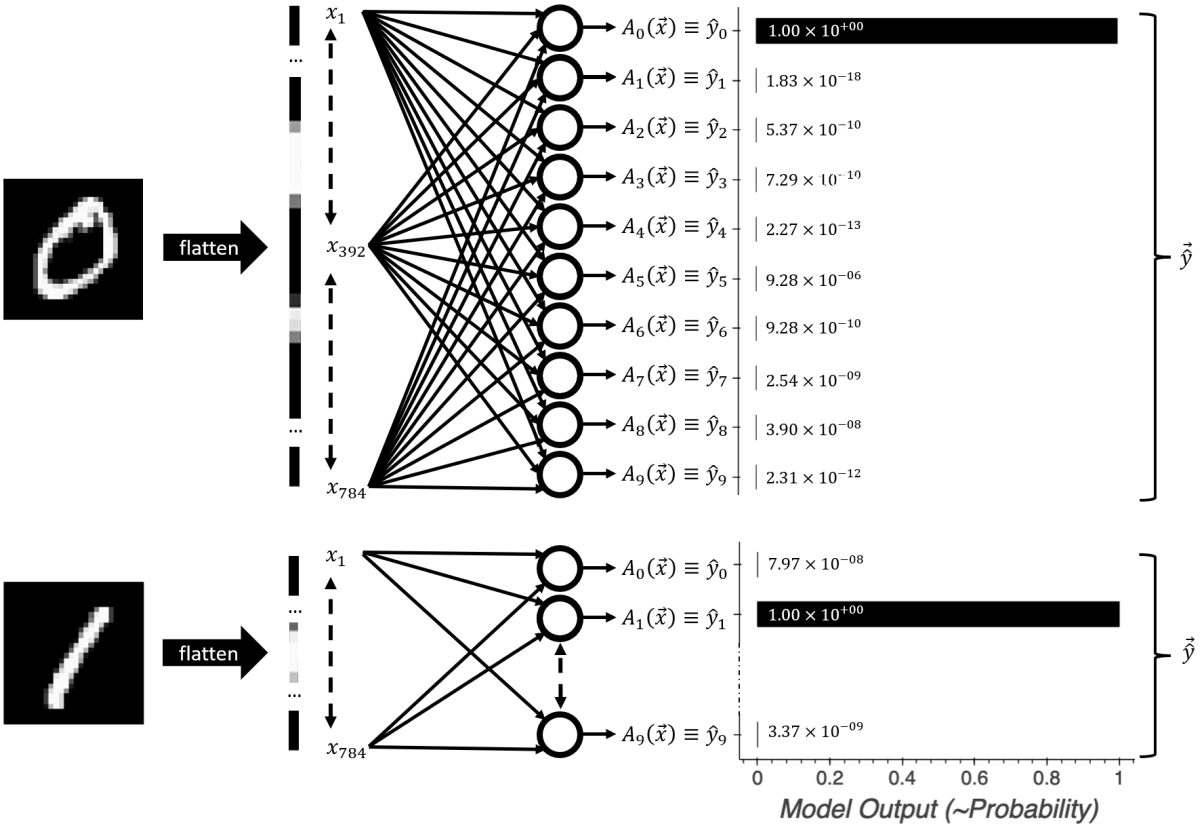


Figure 8 | Example MNIST data [35]. A single example of each of the ten classes within the MNIST example dataset. As can be seen, the classes range from zero to nine inclusive. Each example consists of a grid of 28 by 28 pixels containing one float value between 0.0 and 1.0. In the above image, values near one are represented as nearly white, and values near 0.0 as black. When ingested by our single-layer perception, they will be flattened into a 1D vector; see Figure 9.



Type	Num Filters / Neurons / Heads	Filter / Pool / Head Size	Stride	Activation Function	Dropout
Dense	10	-	-	SoftMax	-

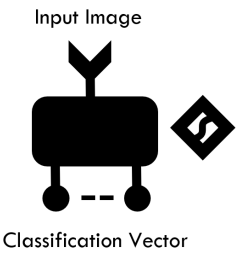


Figure 9 | Various representations of a Single-Layer Perceptron or Single-Layer Artificial Neural Network. *Upper:* Diagram illustrating the structure and operation of a single-layer perceptron. In the example shown, a handwritten zero is fed into the single-layer perceptron. The 2D image is first flattened into a 1D vector; then, the entire vector is fed into each neuron. If the training process has worked correctly, each neuron will have learned to identify one of the possible classes, in this case, digits. As can be seen from the output values, $\vec{\hat{y}} = \left[\hat{y}_0, ... \rightarrow \hat{y}_9\right]$, which are taken from a real trained model, this model can correctly identify this input as a zero with high confidence. *Middle:* An abridged version of the upper diagram demonstrating the operation of feeding a handwritten one into the perceptron. This shows how future network diagrams will be abstracted for simplicity and that the perceptron outputs a different, correct value when it ingests a one rather than a zero. *Lower:* A further abstraction of the network. This type of abstraction will be used commonly throughout this thesis when dealing with networks consisting of multiple layers. A dense layer, wherein all neurons are attached to all previous neurons, will be shown as a filled black rectangle, and the icon next to it represents that the activation function applied is a softmax activation function; see Section 3.1.5.4.

Step 1: Dataset Acquisition: When we train a machine learning model, we are attempting to model the relationship between an input and an output distribution. In some ways, the model can be considered a compressed version of the matched input and output distributions. After training, when you feed in a single data point from the input distribution, the model will, hopefully, be able to map that input value to the correct value in the output distribution. This makes the training data a fundamental part of the training process. Whether naively attempting a manual solution or optimising through more efficient means, we must acquire a suitable training dataset.

In many cases, the input distribution will be very large or even continuous, so an exhaustive training dataset covering every possible value in the distribution will be either technically or literally impossible. For this reason, we have to ascertain or generate a training dataset which will appropriately sample the entire input distribution. There are many preexisting example training sets; as stated, we will use the MNIST dataset [35] for illustrative purposes.

Automating the process of acquiring a dataset is simple. TensorFlow [34] has built-in functions to allow us to acquire the MNIST dataset [35] easily. Listing 2 below shows us how this process can be performed. The listed function also prepares the data for ingestion by the ensemble of artificial neurons. **One hot encoding** changes a single numerical class label, i.e. $0, 1, ..., 9$ into a Boolean vector where each index of the vector represents a different class; for example, 0 becomes $[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]$, whereas 1 becomes $[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$. This is because each neuron will learn to distinguish a single class by returning a float value closer to 0.0 if the input falls outside its learned distribution or closer to 1.0 if the input falls within its learned distribution. Therefore to perform the vector operations necessary for training, one hot encoding must be performed.

```
# Step 1: Load and prepare the MNIST dataset.
def load_and_prepare_data():

    # This data is already split into train and test datasets.
    (x_train, y_train), (x_test, y_test) = mnist.load_data()

    # Reshape and normalize the images to feed into the neural network.

    x_train, x_test = x_train.reshape(-1, 784)/255.0, x_test.reshape(-1, 784)/255.0

    # Convert labels to one-hot vectors. This is necessary as our output layer will
```

```
have 10 neurons,
    # one for each digit from 0 to 9.
    y_train, y_test = tf.one_hot(y_train, depth=10), tf.one_hot(y_test, depth=10)

    return x_train, y_train, x_test, y_test
```

Listing 2 | *Python [32]* . Function to load and prepare the MNIST dataset [35]. The MNIST dataset consists of many examples of handwritten Arabic numerals from one to nine. The images, `x`, are reshaped, and the labels, `y`, are one-hotted.

Step 2: Parameter Initialization: For an artificial neuron to produce a result when it consumes an input vector, all parameters, $\vec{\theta}$, must be initialised to some value. One could imagine choosing these initial parameters, $\vec{\theta}_0$, in a few distinct ways. Perhaps most intuitively, you could decide on the parameters based on some prior knowledge about the dataset, aiming to get as close as possible to the optimal tunings in order to minimise the number of steps required during training. However, this option is impossible if the human tuner lacks such intuition or if the input size is too large for any human to form such an intuition. That leaves choosing a uniform value for all parameters or randomly initialising all parameters.

In any automated process, a uniform initialisation is a bad choice. If one sets all initial parameters, $\vec{\theta}_0$, to the same value, this creates symmetry. Suppose we subsequently try to use a mathematical method to adjust these parameters. In that case, the method will have no way to choose one parameter over another, meaning all parameters will be tuned identically. We will need the parameters to be adjusted independently in order to model complex relationships. For this reason, we will initiate the weights matrix, W, randomly by sampling values from a normal distribution. This choice of random distribution will not be discussed here, but note that there is an open area of research hoping to speed up and/or improve the training process by selecting more optimal distributions for parameter initialisation. The bias values, $\vec{b}$, will be initialised to zero. Since there is only one bias value per neuron, we don't have to worry about creating asymmetry, as that is provided automatically by values passed through the neuron's weights.

Listing 3 demonstrates the initialisation of two variable tensors to hold the weights and biases of our artificial neurons. Because there are ten classes of numbers in the training dataset, we will initialise ten artificial neurons — one to recognise each class of digit. There will be a single bias value for each neuron. Hence there are C bias elements in the bias tensor, `biases`, where $C = \text{num\_classes} = 10$, and the input size is $N = 28 \times 28 = 784$, so there are $N \times C = 784 \times 10 = 7840$ elements in our weights tensor, now a matrix, $W =$ `weights`, arranged in the shape `[784, 10]`. This means the total number of tunable parameters in our set of ten neurons is $7840 + 10 = 7850$.

```
# Step 2: Define the model
# We are using a simple single-layer perceptron model
# This is essentially a single fully-connected layer

def define_model():
    # Define weights and biases. We initialise the weights with a random normal
    # distribution.
    # There are 784 input neurons (one for each pixel in the 28x28 image) and ten
output
    # neurons. We initialise biases to zero.
```

```
    weights = tf.Variable(tf.random.normal([784, 10]), name="weights")
    biases = tf.Variable(tf.zeros([10]), name="biases")
    return weights, biases
```

Listing 3 | *Python [32]* . Function to initialise TensorFlow [34] tensors to store the artificial neuron's parameters, $\vec{\theta}$. In the case of MNIST [35] digit recognition, there are ten neurons being trained, so we have ten bias values, $\vec{b}$, and the input images are of dimension $28 \times 28 = 784$. Therefore, our weights matrix, W, is shaped `[784, 10]` .

Step 3: Define the model's action: To perform any optimisation method, there must be a way to test the model. Thus we must define the action of the model, $M(\vec{x})$. We have already shown what form this must take in Section 3.1.1 and Equation 2. This is very easily defined by a Python [32] function, as seen in Listing 4.

```
# Step 3: Define the model's computations:
def model(x, W, b):
    return tf.nn.softmax(tf.matmul(x, W) + b)
```

Listing 4 | *Python [32]* . Function to perform the computation of artificial neurons in our single-layer perceptron. Since TensorFlow [34] is natively vectorised, this function will calculate the output of all our tensors simultaneously. This function performs the same operation described in Equation 2, with a softmax function as the activation function, f. Softmax activation functions are described in Section 3.1.5.4.

Step 4: Define the loss function: Now that we have set up a procedure to run the model with a set of randomised parameters, $\vec{\theta}$, we must define a measure of success so that we can see how well the model is performing whilst we perform our parameter tuning operation. If we have no performance metric, then we have no indication of how to tune the model to improve its performance. To do this, we define a loss function, L, a function which takes in some information about the state of the model after it has ingested data, usually including the model's output, and returns a numerical output value: the loss of the model with a given set of parameters, $L\left(M_{\vec{\theta}}(\vec{x}), \vec{y}\right)$, where $\vec{x}$, is a particular instance, or batch, of input vectors, and $\vec{y}$ is, in the case of classification, the data label. Note that in unsupervised learning, the loss function does not ingest a label, $\vec{y}$, as the data is not labelled.

By convention, a high loss value indicates that the model performance is worse than that which would be indicated by a lower loss value. Our optimisation process, therefore, should attempt to minimise the average of this loss value across all potential input vectors.

There are many possible metrics for measuring the performance of a model, a large number of which can be used as the loss function. The loss function is an important aspect of the training process, which can alter the efficiency of the training significantly. They can be highly specialised to particular scenarios, to the point where using an inappropriate loss function can completely remove any possibility of training. A more detailed description of loss functions is available in Section 3.1.6.

For this model, we elect to use the categorical cross-entropy loss function as described in Section 3.1.6. An implementation of that loss function is shown by Listing 5.

```python
# Step 4: Define the loss function
def compute_loss(y_true, y_pred):
    return tf.reduce_mean(-tf.reduce_sum(y_true * tf.math.log(y_pred), axis=[1]))
```

Listing 5 | *Python [32]* . Function to compute the loss of the model. The loss function utilised in this case is categorical cross-entropy loss, a loss function commonly used for multi-class, single-label datasets. A more detailed description of the function of this loss function can be found in Section 3.1.6.2.

Step 5: Train the model: Finally, after we have assembled all the pieces, we can start to tune the parameters, $\vec{\theta}$, so that our perceptron can output useful values when fed input. As we have previously stated, we will initialise our weights parameters, W, randomly; this means that no matter what images we feed into the untrained model, we will get non-sensical classification values with no correlation to the ground truth labels unless by extremely unlikely fluke. Using some process, we want to move the model toward successful categorisation.

If we again move back to our analogy of attempting to perform this operation manually, what we might imagine is that we would start by feeding it an image from our training dataset. We could then examine the model's output and see which parameters we would need to tune in order to move our network, for that particular image, toward the correct answer. We could achieve this by determining how much each parameter moves the current model's output, $\vec{\hat{y}}$, toward or away from the ground truth value, $\vec{y}$, and then adjusting each parameter accordingly.

If we tuned the parameters by a large amount, then the model could easily become overtuned to a particular image, so we might instead choose to move it a little bit toward the correct input value and then repeat this process over hundreds, if not thousands, of examples, moving the network slowly toward a useful configuration.

Gradient descent is an algorithmic implementation of this thought experiment. In its most simple case, the loss that is given by the loss function, $L(M_{\vec{\theta}}(\vec{x}), \vec{y})$, measures the distance between the model output, $\vec{\hat{y}}$, and the ground truth, $\vec{y}$. Since the model is largely defined by its parameters, $\vec{\theta}$, the loss function can be thought of as a function which takes in an input vector, $\vec{x}$, the model parameters $\vec{\theta}$, and the ground truth label, $\vec{y}$. So the output of the loss function for a particular input vector and set of parameters becomes

$$L(M_{\vec{\theta}}(\vec{x}), \vec{y}) = L\Big(M\big(\vec{\theta}, \vec{x}\big), \vec{y}\Big) = L_M\big(\vec{\theta}, \vec{x}, \vec{y}\big), \tag{3.4}$$

where L is the model-architecture-agnostic loss function, L_M is the loss function for a particular model architecture, $M_{\vec{\theta}}$, is a model with a fixed set of parameters, $\vec{\theta}$, M is a model with a set of parameters as a functional input, $\vec{x}$, is a particular input vector to the model, and, $\vec{y}$, is the label vector corresponding to the model input vector.

The gradient of the model is defined as the vector of partial derivatives of the model's loss function with respect to its parameters. If $L_{M\vec{x}\vec{y}}\big(\vec{\theta}\big)$ is the loss function with a fixed model architecture, input vector, and ground-truth label, then the gradient of the model is

$$\vec{\nabla} L_{M\vec{x}\vec{y}}\big(\vec{\theta}\big) = \left[\frac{\partial L_{M\vec{x}\vec{y}}}{\partial \theta_1}, \ldots \frac{\partial L_{M\vec{x}\vec{y}}}{\partial \theta_i}, \ldots \frac{\partial L_{M\vec{x}\vec{y}}}{\partial \theta_N}\right] \tag{3.5}$$

where N is the total number of tunable parameters.

Equation 5 describes a vector, $\vec{\nabla} L_{M\vec{x}\vec{y}}\left(\vec{\theta}\right)$. Each element of the vector, $\frac{\partial L_{M\vec{x}\vec{y}}}{\partial \theta_i}$, is a gradient which describes the effect of changing the value of the corresponding parameter, θ_i, on the model loss. If the gradient is positive, then increasing the value of the parameter will increase the value of the loss, whereas if it's negative, increasing the value of that parameter will decrease the model loss. The magnitude of the gradient is proportional to the magnitude of that parameter's effect on the loss.

Since we want to reduce the model loss, we want to move down the gradient. Therefore, for each parameter, we subtract an amount proportional to the calculated gradient.

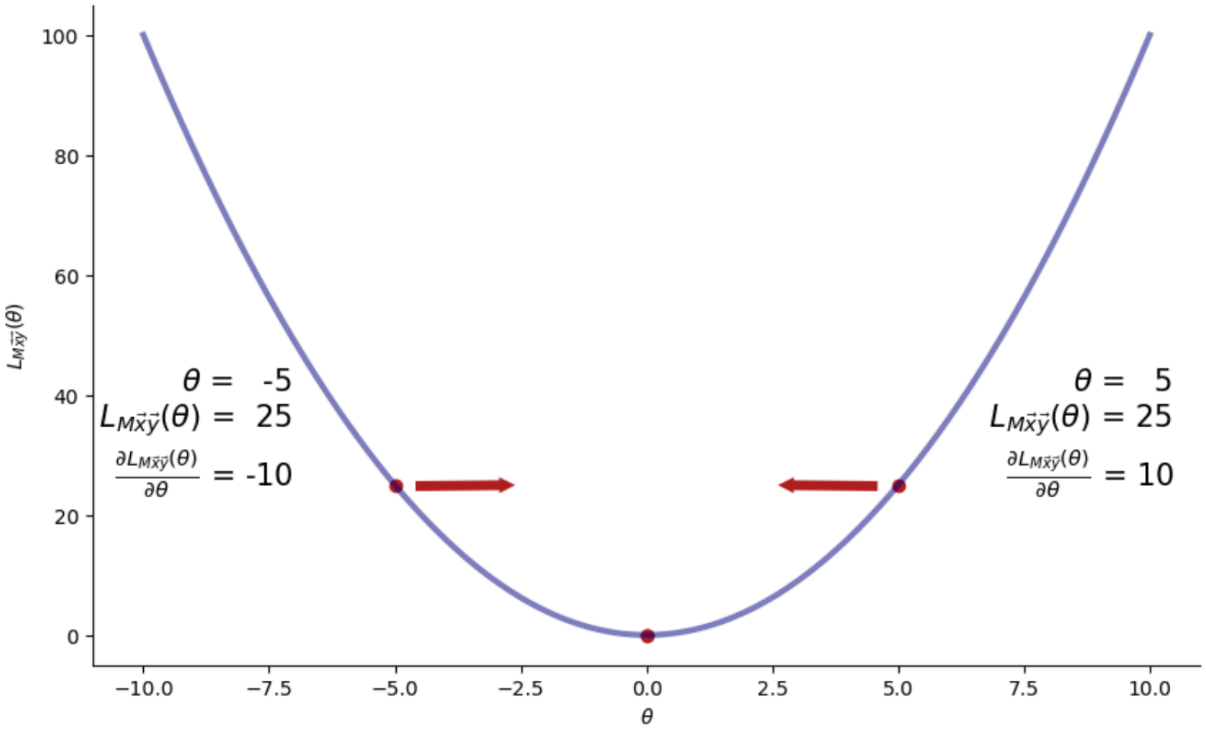


Figure 10 | An illustration of gradient descent, where $\vec{\nabla} L_{M\vec{x}\vec{y}}\left(\vec{\theta}\right)$ is the loss at a fixed model architecture, M, input vector $\vec{x}$, and data label $\vec{y}$. This simplified example of the shape of a 1D parameter space shows how the gradient of the loss function with respect to the model parameters can be used to move toward the minimum of the loss function. The shape of the loss function in this example is given by $L_{M\vec{x}\vec{y}}\left(\vec{\theta}\right) = \theta^2$. In almost all cases, the parameter space will be much more complex than the one depicted in both dimensionality and shape complexity. Usually, the shape of the loss function will be an N-dimensional surface, where N is the number of parameters, $\vec{\theta}$, in the model, but the principle is still the same. For a 2D example of a gradient space; see Figure 17. This plot can be recreated with the code found here: https://github.com/mrknorman/data_ad_infinitum/blob/main/chapter_x_gradient_descent.ipynb

We need to be able to control the magnitude of the parameter adjustment because the gradient is only measured for the current parameter values, $\vec{\theta}$. Therefore we are unsure of the shape of the loss function. It's possible for the tuning process to overshoot the loss function minimum. In order to apply this control, we introduce a constant coefficient to scale the gradient, known as the learning rate, η.

Therefore, if we want to find the new adjusted parameters after one optimisation step, we can use

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \eta \vec{\nabla} L_{M\vec{x}_t\vec{y}_t}\left(\vec{\theta}_t\right) \tag{3.6}$$

where t is the step index, we can see this process in a Python [32] form in Listing 6. In this function, the gradients are captured using the tf.GradientTape scope, which automatically captures the gradients of all "watched" tensors within its scope. This automatic differentiation utilises a process called back-propagation, which will be discussed in more detail in Section 3.2.5.

```python
# Step 5: Define the training step
@tf.function
def train_step(x, y, W, b, η):
    with tf.GradientTape() as tape:
        y_pred = model(x, W, b)
        current_loss = compute_loss(y, y_pred)
    gradients = tape.gradient(current_loss, [W, b])
    W.assign_sub(η * gradients[0]) # update weights
    b.assign_sub(η * gradients[1]) # update biases
    return current_loss
```

Listing 6 | *Python [32]* . Function to execute a single training step. This function runs an example, `x` $= \vec{x}_t$, through the model (usually multiple examples at once as explained in Section 3.2) and computes the loss, `loss` $= L_{M\vec{x}_t\vec{y}_t}\left(\vec{\theta}_t\right)$ of the output of that model, `y_pred` $= \hat{\vec{y}}_t$ compared with the ground truth label of that example, `y` $= \vec{y}_t$. The gradients, `gradients` $= \vec{\nabla} L_{M\vec{x}_t\vec{y}_t}\left(\vec{\theta}_t\right)$, are automatically computed for each parameter by `tf.GradientTape()` , which produces a list of gradients for the weights, `w` $= W$, and biases, `b` $= \vec{b}$, which are then used multiplied by the learning rate `η` $= \eta$ and used to update the parameters, $\vec{\theta}$, for the next training step; see Equation 6.

If we repeat this process over T steps, where T is the number of training examples in our dataset, then the model will hopefully begin to gain aptitude at the classification task. The process of tuning the model parameters once with all examples in the training dataset is called a training epoch. Oftentimes, if our training dataset is not large enough, we can improve the model performance by running for multiple epochs, hence training the model with the same examples multiple times. Between epochs, the training dataset is usually shuffled in order to explore new areas of parameter space and avoid repeating exactly the same pathway.

Pulling all the functions we have defined together; we can now implement our main training loop, Listing 7.

```python
def train_model(epochs, batch_size, η, x_train, y_train):
    # Define model
    W, b = define_model()

    # Store loss and accuracy for each epoch
    loss_per_epoch = []
    accuracy_per_epoch = []

    # Training loop
    for epoch in range(epochs):
        i = 0
        while i < len(x_train):
            start = i
            end = i + batch_size
            x_batch = x_train[start:end]
            y_batch = y_train[start:end]
```

```
            current_loss = strategy.run(train_step, args=(x_batch, y_batch, W, b,
η))
            i += batch_size

        # Compute loss and accuracy for each epoch
        y_pred = strategy.run(compute_model, args=(x_test, W, b))
        loss_per_epoch.append(current_loss)
        accuracy_per_epoch.append(compute_accuracy(y_test, y_pred))
        print(f'Epoch {epoch+1} completed')

    return loss_per_epoch, accuracy_per_epoch, W, b
```

Listing 7 | *Python [32]* . Function to execute multiple training steps across multiple epochs. This function runs the function defined in Listing 6 for each example in the training_dataset, `x_train`, and repeats this process for each requested epoch, `num_epochs`, updating the model parameters each time. It returns the model parameters, `W, b`, and some metrics measuring the model's performance; see Equation 6.

3.1.3 Testing the Model

Once we have trained our model using the aforementioned procedure, we can evaluate its performance. Often the first step toward this is to look at the model's performance at each step during training; see Figure 11.

The model training progresses quickly at first but soon reaches a point of diminishing returns at about 85 per cent accuracy. Although we may be able to squeeze out a little more performance by running the training for more epochs, this can lead to overfitting, where a model becomes tailored too specifically to its training dataset and cannot generalise well, or at all, to other points in the training distribution. In most cases, we will want our model to classify new unseen data drawn from a similar distribution as the training dataset but not overlapping with any existing points, so we try to avoid this.

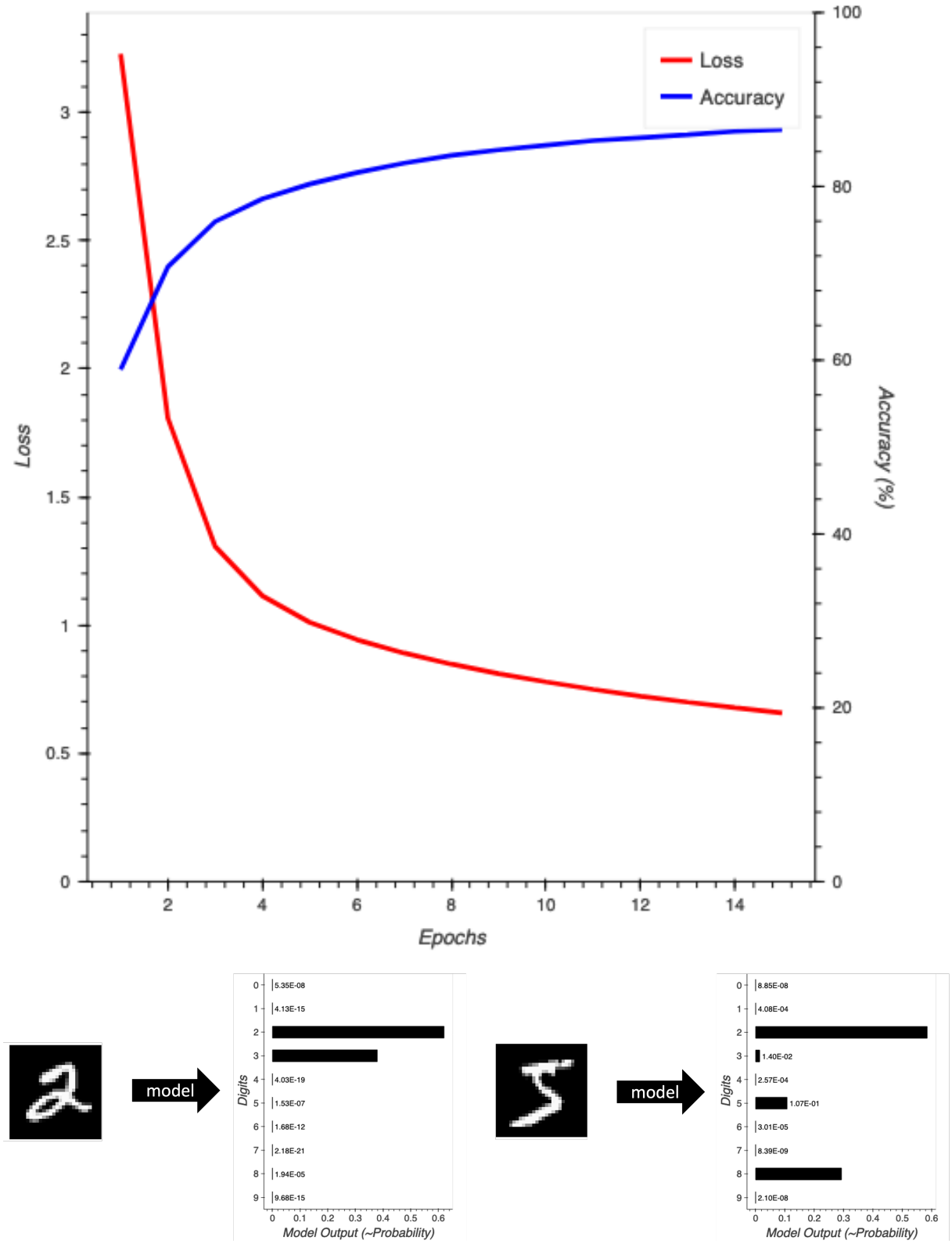


Figure 11 | *Upper:* The performance of the single layer perceptron model described in Section 3.1.2 over 15 epochs, where one epoch consists of training the model on all training examples in the MNIST dataset of handwritten Arabic numerals [35]. The model loss is defined as the categorical cross-entropy of the model's output vector, $\vec{\hat{y}}$ and the ground-truth label, $\vec{y}$, whereas the accuracy metric is defined as the number of examples in the test dataset that are correctly classified, where a correct classification is any output with 50 per cent or more probability in the correct class. *Lower*: Two examples of less

successful classifications. The left example would still be measured as a successful classification by our accuracy metric, whereas the right example would be marked as an unsuccessful classification.

We can also investigate what parameters the neurons have actually learned over this process; see Figure 12. It is often very difficult to come to much of a conclusion about the true inner workings of artificial neural networks, especially dense layers, which are the most general but also the most non-intuitive. Network interpretability is a large and ongoing area of machine learning research for many obvious reasons. Being able to see why a model has given you the answer that it has can massively boost confidence in that answer. However, this thesis will not focus heavily on interpretability, as that could be a whole other thesis on its own.

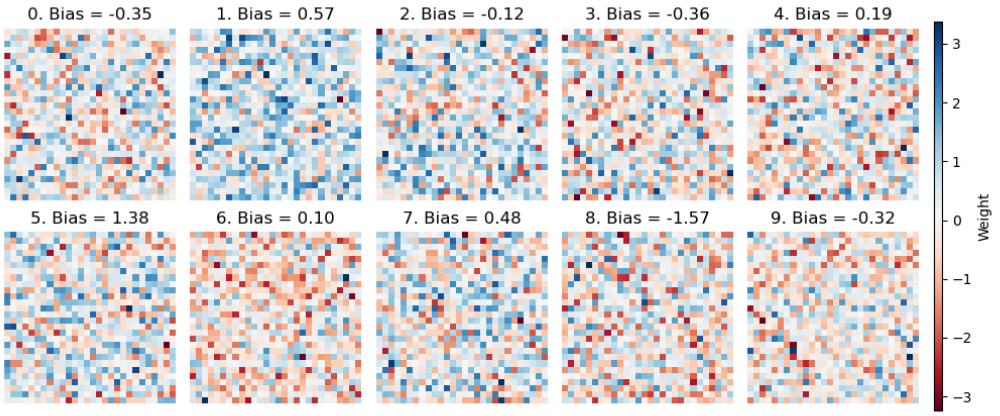


Figure 12 | Learned model parameters. Each artificial neuron in our single-layer perception is represented by a labelled parameter map shaped into the same dimensions as the input images. These maps show the learned weight values that correspond to each pixel of the input images. Very little structure can be made out by the human eye. Perhaps in the weight maps for the zero-classifier neuron, we can see an area toward the centre of the map that is negatively weighted. This might be expected as there are rarely high-value pixels at the centre of the circular zero. A similar but opposite effect might also be attributed to the one-classifier, where the centre of the image often contains high-value pixels. In general, unless you squint very hard, it is difficult to make out patterns in the parameters. This "black-box" effect means that after even one more layer is added to the network, it becomes very difficult to determine the action of dense layer neurons intuitively.

Whilst it is difficult to make specific claims on how artificial neural networks are doing what they are doing, we can often speculate on general methods of operation. In the case of the single-layer perceptron, like the one we have built here, the only kinds of operations that can be learned are linear ones. The only path each neuron has available to it is to learn which pixels are often highly valued in its class of digit and which pixels are very rarely highly valued in its class which are more likely to be highly valued in another class. Then it can adjust the bias value so that the neuron only activates when a certain criterion is met. If we were distinguishing between ones and zeros, for example, which have, in general, very different highlighted pixels, then this might be enough for a high degree of classification efficiency. However, there are a multitude of digits which can share many common pixel values, which makes this problem more difficult.

In order to solve the problem with more accuracy, we must add a non-linear element to the computation and the ability for the model neurons to work collaboratively on the problem. This allows the

model to extract more complex "features" from the input vector. We, therefore, introduce the concept of multi-layered neural networks and deep learning.

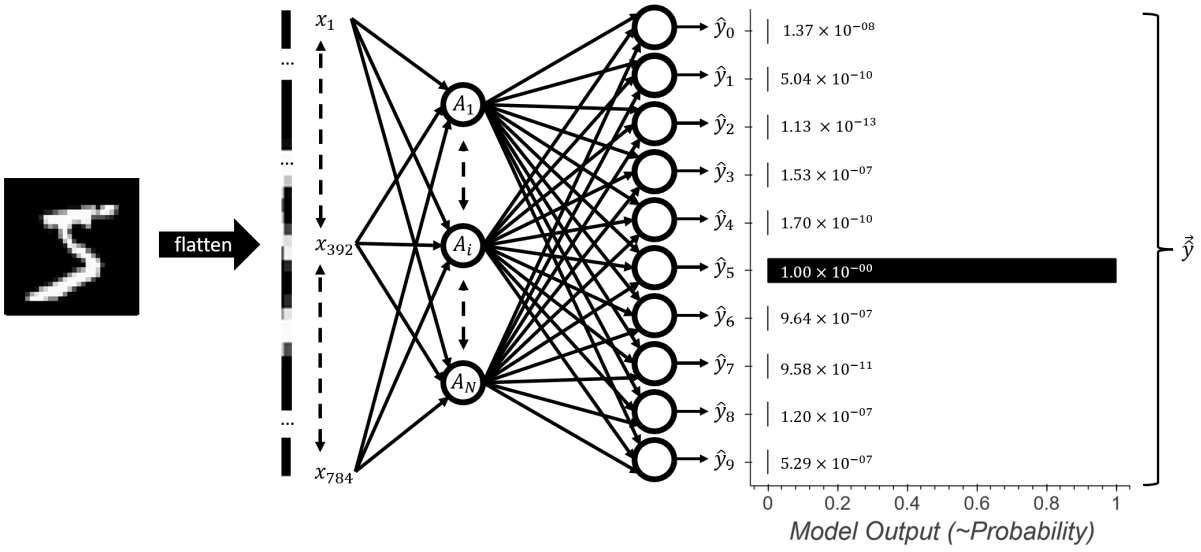
3.1.4 Neurons Together Strong

As we have seen, a single layer of artificial neurons can be trained to perform a small amount of computation, which is often enough for many simple problems. There are, however, a great many problems which require more complex solutions. In order to do this, we can add what is known as "hidden layers" to our network. They are called hidden layers because the exact computation that is performed by these additional layers is much more difficult to divine than in output layers, layers that directly output solution vectors, as we have seen in our single-layer perceptron.

For simplicity of design, artificial neural networks are usually organised into layers of neurons, which are usually ordered, and interactions are usually limited to between adjacent layers in the network. Layer one will usually only pass information to layer two, and layer two will receive information from layer one and pass information to layer three if there are three layers; see Figure 13. This is not always the case, and there are exceptions to all the rules mentioned in this paragraph, including skip connections, recurrent neural networks, and Boltzmann machines.

Artificial neural network layers come in many varieties, the most simple of which are feed-forward dense (or sometimes linear) layers. Dense layers consist of N neurons, where every neuron takes as an input vector the output of every neuron on the previous layer unless the dense layer is acting as the input layer, in which case every neuron takes in as input every element of the input vector. If the dense layer is acting as the output layer, as was the case for our single-layer perceptron where one layer was both the input and output layer, then N must equal the required size of our output vector, $\vec{y}$. In the case of a classification problem, this is equal to the number of classes, C. In hidden layers, the number of neurons, N, can be any number and is, therefore, a customisable non-trainable parameter known as a hyper-parameter that must be chosen before network training by some other method; see Section 5.1.

As can be imagined, finding the gradient for networks with one or more hidden layers is a more complex problem than for a single layer. Backpropagation allows us to do this and, in fact, is the tool that unlocked the true power of artificial neural networks; see Section 3.2.5.



Type	Num Filters / Neurons / Heads	Filter / Pool / Head Size	Stride	Activation Function	Dropout
Dense	128	-	-	ReLU	-
Dense	10	-	-	SoftMax	-

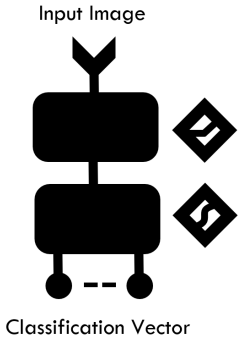


Figure 13 | *Upper:* Diagram of a multi-layer network with one output layer and one hidden layer. The non-linear computation introduced by the ReLU activation function applied to the hidden layer allows this network to solve considerably more complex problems than the previously described single-layer perceptron model. *See Section 3.1.5.3* . As can be seen, by the displayed output, which again is taken from a real instance of a trained model, this network has no problem classifying the previously difficult image of a five. *Lower:* An abstraction of the same model.

This model performs considerably better than the previous model; see Figure 13; and seems to be complex enough to more or less solve the problem; see Figure 14. We might, in fact, even try reducing the number of neurons in the hidden layer. It is often beneficial to find the simplest possible network, by the number of parameters, that is able to achieve the desired computation, as more complex networks are more computationally expensive and time-consuming to train, require more training data, have an increased inference time (inference meaning to run the model on new unseen data), and crucially are more prone to overfitting to the training dataset. This model reaches a high percentage of test accuracy within just a few epochs, and unless we are very concerned about false alarm rates, then there is no need to add extra complexity to our model.

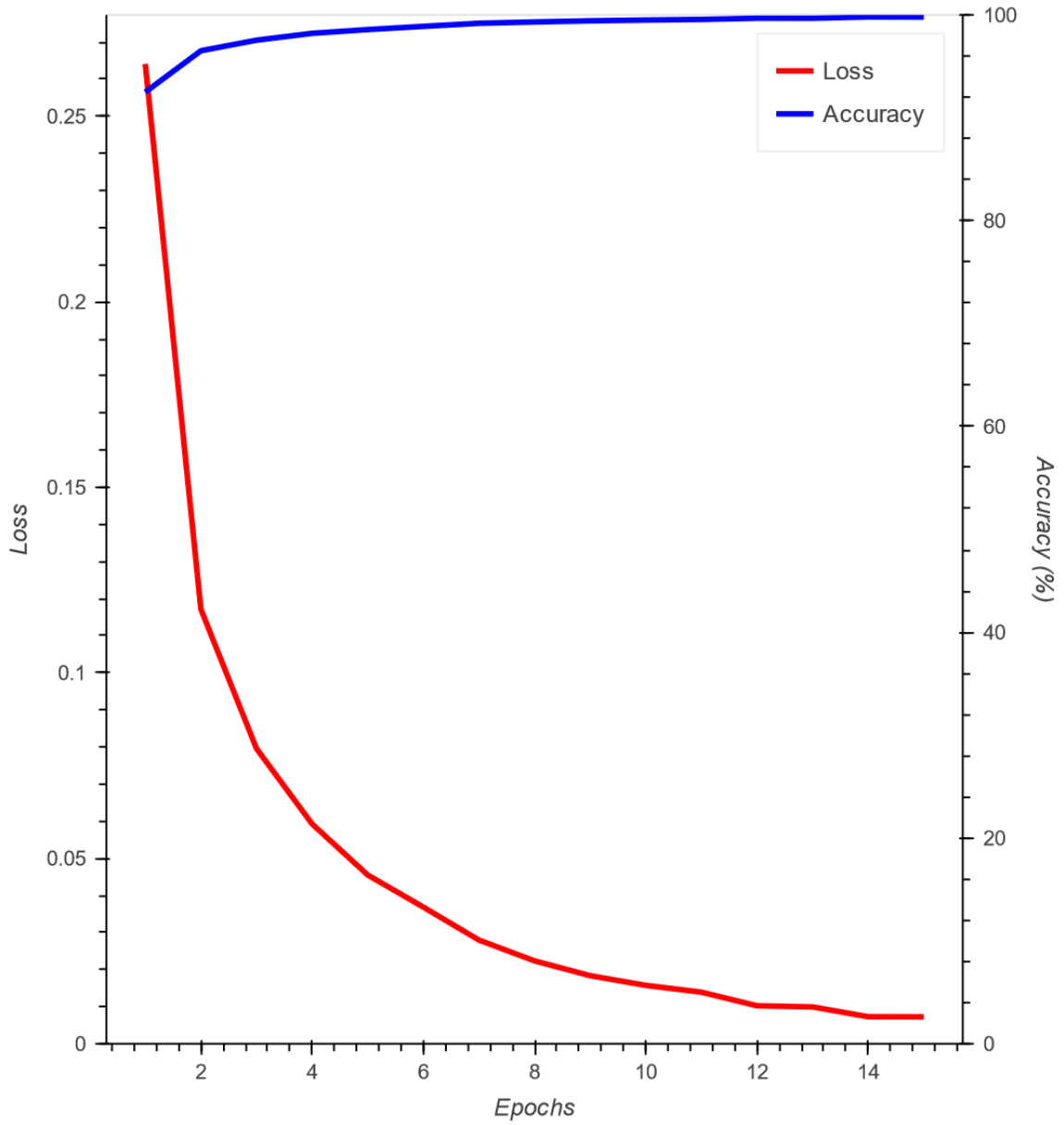


Figure 14 | The performance of the multi-layer perceptron model described in Section 3.1.4 over 15 epochs. As can be seen in comparison to Figure 11, the training is both faster and with a better final result.

The addition of hidden layers to our network architectures introduces the possibility of all kinds of structural variations. For more complex problems than the recognition of simple handwritten digits, we have many tools in our arsenal to increase performance. The first thing you might do is try the addition of more than one hidden layer; in fact, there is no inherent theoretical limit to the number of hidden layers that can be added to a network. Of course, at some point, you would run into computational limits, and although the gradient can be calculated, there are problems when attempting to run gradient descent algorithms on very deep networks that are designed without careful consideration. Gradients that vanish over many layers can lead network training to become an almost impossible task.

These problems with increasing complexity lead researchers to explore types of layers beyond the dense layer. Although the dense layer alone can be thought of as a universal function approximator, there exists no perfect training algorithm to find the ideal set of parameters to achieve every possi-

ble function, and this statement is technically only true for all possible arbitrarily complex functions as the number of layers approaches infinity. For this reason, different layer designs and network architectures can create easier environments for training, saving computational resources and allowing feasible routes to massively increase network ability. Most often, these non-dense layers are designed with some insight into the structure of the input vectors. An example of this would be the convolutional neural network, see Section 5.2.2, which uses the spatial information of the input, as well as the notion that there will be transform-invariant features within the image, to create layers that can perform a similar or better job than dense layers with far fewer parameters.

One could also experiment by moving away from the paradigm of feed-forward networks, although this can increase solution complexity significantly. Within feed-forward neural networks, neuron connections only ever move toward neurons that have not yet causally influenced the emitting neurons. Within recurrent networks, however, signal paths can loop, taking either previous inferences as inputs or looping within the calculation itself. This can allow the network memory of previous inferences, something feed-forward networks do not possess.

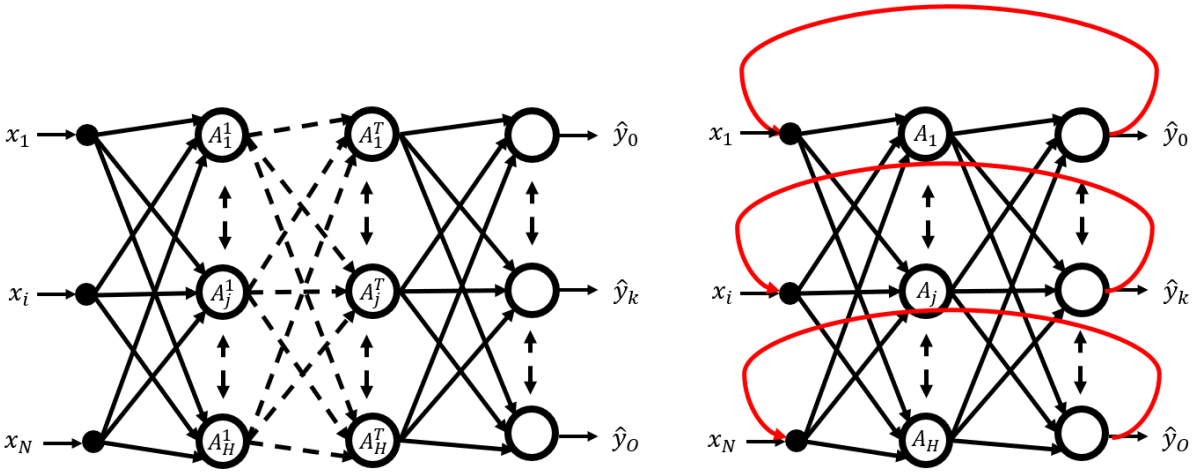


Figure 15 | *Left:* The generalised dense feed-forward artificial neural network. Where T is the number of hidden layers in your network, H is the number of neurons at that layer, N, is the number of elements in the input vector, $\vec{x}$, and O is the number of elements in the output vector $\vec{\hat{y}}$. As can be seen in the diagram, the number of hidden layers in your network is unconstrained, as is the number of neurons in each of those layers, which should be noted does not have to be the same. This is opposed to the output layer, which must have the same number of neurons as is expected by your loss function. *Right* A very simple illustration of a recurrent neural network. This network illustrates the retroactive data flow that is possible in a recurrent neural network. In this example, the output of the network from one inference operation is added to the input of the next inference operation. It should be noted that this is a very naive implementation of a recurrent neural network. In actuality, the networks usually have a much more complex structure, such as LSTMs (Long Short Term Memory) networks.

There will be a more detailed discussion of many of these different network layers and architectures further on in the thesis; the following few sections will explore the concepts outlined in the last two sections in more detail.

3.1.5 Activation Functions

There are many different activation functions; as with most things in machine learning, they are an active area of research. As such, this small section will only give a brief introduction plus a few examples. Since the activation function normally acts on the weighted sum of the inputs plus the bias, in

this section, we will define $z = \sum \vec{x}W + \vec{b}$ to avoid confusion with the raw input values previously denoted x. It should be considered that z could also be other values in some cases. We will define the vector of all z values in a single network layer of N neurons, i.e. $\vec{z} = [z_1 ..., z_i ..., z_N]$.

As noted in Section 3.1.1, the activation function aims to coerce an artificial neuron's output into a particular shape. This has several purposes. Firstly, it can act as a thresholding function, which along with a specific value of bias, b, can activate or deactivate the neuron depending on the weighted sum of the input vector, z. The activation function also limits the output values to a specific range, ensuring that values within the network do not grow without bounds along favoured pathways and destabilise the network. These values can be considered in some way analogous to the maximum firing rate of a biological neuron. Without activation functions, instability can cause values to explode to infinity or vanish to zero. Finally, activation functions provide a non-linear component to the neuron. Without non-linear activation functions, neuron output, hence network outputs, could only be linear combinations of the input values and so would need to be, in general, much more complex to solve non-trivial problems.

There are some limits to the type of function we can use within a neural network, primarily since we must be able to flow gradients through the function during backpropagation; the function must be differentiable at all points. For example, if we tried to use a step function as an activation function, the derivative would be 0 at all points, except for at the step where it would be undefined. This would make backpropagating through this function very difficult, as it would fail to update the weights and bias of its corresponding neuron. In other non-continuously differentiable functions, like the ReLU function, we can use a trick to avoid the undefined derivative by defining the value of the derivative at that point, $z = 0$ in this case, to 0 or 1.

As well as the distinction between linear and non-linear activation functions, a few further distinctions can be made. Outside of the linear function, we can split activation functions into three types: ridge, radial basis, and fold.

Ridge functions are standard activation functions that change an input's shape based on directionality around a specific point or ridge. The most common example is the ReLU function and its variants described below. Ridge functions are computationally efficient and introduce non-linearity without requiring exponentiation or other computationally expensive operations.

Radial basis functions, on the other hand, are less commonly used. They are symmetric around a specific point rather than just directional. Their value, therefore, depends entirely on the magnitude of the distance to this point rather than in ridge functions where the sign is also vital. Radial basis functions can create complex surfaces which can localise to a specific region, which can be helpful if you believe your data structure to be localised in such a manner. A typical example of a radial basis function would be the Gaussian function, which can localise a neuron's activation to a particular region. However, they can be computationally expensive and lead to overfitting due to their ability to form complex surfaces.

Fold functions are complex activation functions that aggregate over multiple neuron z values, such as mean or max functions in pooling layers, or even over the entirety of $\vec{z}$, such as in softmax layers described below. Calculating these can be computationally expensive, so they are used in moderation.

3.1.5.1 Linear

The most straightforward activation function is the linear activation, represented simply by Equation 7 below. The linear activation function will not change the shape of the data and is crucial for many applications where this is a desired feature:

$$\text{linear}(z) = kz. \tag{3.7}$$

Evidently, in the case where $k = 1$, this is equivalent to not applying any activation function and thus, all the previously stated problems resulting from no activation function will apply. The derivative of the linear activation function is always a constant irrespective of the input values, so it is straightforward to compute. This simplicity brings a significant drawback when dealing with complex data. If it is the only activation function used, the entire network, regardless of complexity or number of layers, will behave as a single-layer model because of the lack of non-linearity between layers. As we have seen, single-layer perceptrons are insufficient for many tasks we wish to tackle with artificial neural networks.

One of the primary uses of linear activation functions is as the output layer of regression problems, where the output is expected to be a continuous float value not constrained within a particular distribution. The drawbacks are alleviated if the rest of the network before the output layer involves non-linear activation, leaving the output layer to combine inputs into a final output value linearly. They can also sometimes be used in straightforward networks where non-linearity is not required and computational efficiency is highly prized. Therefore, while the linear activation function has its uses, it is not commonly applied in hidden layers of deep learning models, wherein non-linear layers, such as ReLU and its variants, are more valuable; see Figure 16 for a graphical depiction.

3.1.5.2 Logistic

The logistic activation function is a ridge function defined

$$f(z) = \frac{L}{1 + e^{-k(z - z_0)}}, \tag{3.8}$$

where $z_{\{0\}}$ represents the z-coordinate of the function's midpoint, L signifies the maximum value that the function can reach, often referred to as the function's supremum and k is a parameter that controls the steepness of the function's curve, determining the logistic growth rate. The particular case where $L = 1$, $k = 1$, and $x_0 = 0$ is known as the sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-kz}}. \tag{3.9}$$

See Figure 16 for a graphical depiction.

Since this smoothly limits the output of the neuron to between 0 and 1, it is often used on output neurons in a network designed for classification since, in this case, the ground truth vector would consist of entirely Boolean values, meaning an activation function that tends to 0 and 1 in the extreme is very useful. The sigmoid activation function is used in multi-class, multi-label classification problems, where each class variable is independent, and an example can be in multiple classes and single-class single-label problems, where there is only one output neuron. Since each output is calculated independently, it is unsuitable for cases where an example can be in only one class and there are multiple classes; in that case, a Softmax layer, as described in Section 3.1.5.4, is more appropriate.

The sigmoid function's derivative is at maximum at the midpoint $z = 0$ and falls off as z moves in either direction $z \to \infty \vee z \to -\infty$; this is a suitable environment for backpropagation as the derivative is always defined and never 0. There are, however, some limitations since the gradient, although never quite 0, can become very small, leading to the "vanishing gradients" problem wherein the model's parameter updates can become negligible, and hence learning is very slow. Secondly, the sigmoid function is not centred at 0. This can lead to zig-zagging during gradient descent, also slowing down convergence. Finally, the sigmoid function involves the computation of exponentials, which can be computationally expensive, especially for large-scale networks.

Despite these limitations, the sigmoid function is widely used, particularly in the output layer, for multi-class, multi-label classification problems. However, for hidden layers, modern practices prefer other functions like ReLU or its variants to mitigate some of the issues related to the sigmoid function.

3.1.5.3 ReLU (Rectified Linear Unit)

One of the most common activation functions used very widely in neural network hidden layers is the ReLU (Rectified Linear Unit) function, defined by

$$\mathrm{ReLU}(z) = \begin{cases} z \text{ if } z > 0 \\ 0 \text{ if } z \leq 0 \end{cases}. \tag{3.10}$$

Relu is another example of a ridge function. This function is 0 for $z \leq 0$, and z for $z > 0$, meaning it is equivalent to the linear function above 0. It is a very simple function but still provides the neuron with the ability to threshold values and adds a non-linear component to the neuron.

Because of its simplicity, ReLU is very computationally efficient compared to other activation functions that require expensive operations such as exponentiation or division, an essential factor when deciding on activation functions to use, especially in very large networks. The derivative is also very simple, either 1 above z or 0 below z; hence it lacks the possibility of becoming very small. This means that the use of ReLU functions can be efficient for training. Having a large section of the domain with a derivative of 0 does, however, also lead to problems. During the training process, some neurons can "die", only able to emit 0s, and since the gradient is also 0, they can become stuck in this state, unable to reactivate. Evidently, this can reduce the capacity of the network for practical computation since these dead neurons can no longer contribute valuable operations.

To ameliorate some of the downsides, there are a plethora of possible ReLU variants, most of which have a non-zero gradient below $z = 0$. These include but are not limited to Leaky ReLU (LReLU), Randomized Leaky ReLU (RReLU), Parametric ReLU (PReLU), Exponential Linear Unit (ELU), and Scaled Exponential Linear Uni (SELU). The first three variants, LReLU, RReLU, and PReLU, are defined by

$$\mathrm{LeakyReLU}(z) = \begin{cases} z \text{ if } z > 0 \\ \alpha z \text{ if } z \leq 0 \end{cases}, \tag{3.11}$$

where α depends on the variant in question; in standard LeakyReLU, α is a small, predefined value such as 0.05, meaning the slope is much shallower before 0 than after it; this prevents dying neurons whilst still allowing the function to threshold the input value. In the case of Randomised Leaky ReLU, α is randomly sampled from a specified distribution during training but fixed during model inference. This solves the dying neuron problem and adds robustness to the training process. Finally, in Parametric ReLU, α is treated as a trainable parameter that the model can adjust during backpropagation, allowing it to hopefully self-optimise to a good value.

ELU and SELU are also both based on a similar definition

$$\mathrm{SELU}(z) = \begin{cases} sz \text{ if } z > 0 \\ s\alpha(\exp(z) - 1) \text{ if } z \leq 0 \end{cases}. \tag{3.12}$$

For any α value if $s = 1$, the equation defines ELU. ELU has all the death-defying properties of the previously mentioned ReLU variants whilst also introducing differentiability at $z = 0$, meaning that the redefinition trick is not required. Unlike other ReLU variants, it saturates as $z \to \inf$, increasing robustness to errors. These extra benefits come at the cost of the computational simplicity of the previous ReLU variants, as the calculation of exponentials is a significant computational expense.

If $\alpha = 1.673263\ldots \wedge s = 1.05070\ldots$, the equation defines SELU, a self-normalising activation function. These very specific values of α and s are designed to work in conjunction with LeCun initialization, a

method that initializes neuron parameters with values drawn from distributions with mean zero and variance $\frac{1}{N}$, where N is the number of neurons in that layer. These values of α and s massage the neurons toward outputs with a distribution centred on zero and with a variance of one. Which can help smooth the training process by avoiding vanishing or exploding gradients.

In practice, ReLU and its variants are the most commonly used activation functions in the hidden layers of deep neural networks due to their efficiency and performance. See Figure 16 for a graphical depiction.

3.1.5.4 Softmax

Softmax is a more complicated fold distribution and is of interest due to its use in multi-class, single-label classification problems. It is an extension of the Sigmoid function described above in Section 3.1.5.2, which aims to convert a vector of continuous unconstrained output values, in our case $\vec{z}$, into a vector representing probabilities, with outputs limited between 0 and 1 and a vector sum equal to exactly 1. It does this by finding the exponential of each z value, then normalising by the sum of the exponential of all elements in $\vec{z}$

$$\operatorname{softmax}\left(\vec{z}\right)_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}, \tag{3.13}$$

where N is the number of elements in $\vec{z}$, equivalent to the number of neurons in the layer and the number of classes in the dataset, and i is the index of the neuron/class whose output value is calculated. See Figure 16 for a graphical depiction.

The softmax function represents a way of mapping the non-normalized output of the network to a probability distribution over predicted output classes, making it invaluable for multi-class, single-label classification problems. It is also differentiable so that it can be used in gradient-decent methods.

Softmax can be computationally expensive, particularly in the case of a large number of classes, as each output classification requires the use of multiple expensive operations such as exponentiation and division; it can also suffer from numerical instability when the scores in the input vector are very large or small which may result in numerical overflow or underflow problems. This is not typically too much of an issue as it is usually only used in the output layer of a network.

The Softmax function remains the standard choice for multi-class classification problems due to its ability to provide a probabilistic interpretation of the outputs, handle multiple classes, and its differentiability.

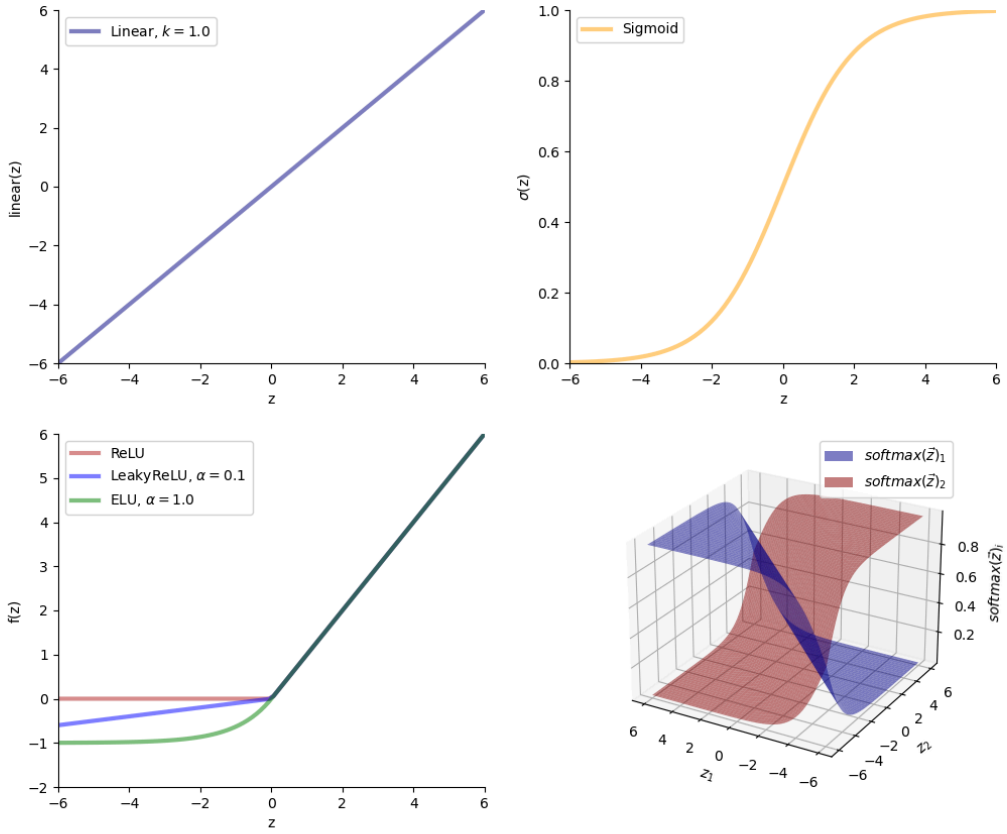


Figure 16 | Four of the most common activation functions. *Upper Left:* A linear activation function. In this case, the slope, k, is 1, meaning that the shape of the output is unchanged vs the input. *Upper Right:* Sigmoid activation function, a special case of the logistic activation function, which limits the output value between 0 and 1. *Lower Left:* ReLU (Rectified Linear Unit) activation function and its variants, an easy way to provide non-linearity to multi-layer networks. *Lower Right:* SoftMax activation function. In the case of multi-neuron outputs, when using softmax, the output of each neuron depends on the value of each other neuron. For this reason, the simplest non-trivial case, where the length of the output vector, N, is 2, has been chosen, and the outputs are represented on a 3D plot. This figure can be recreated with the notebook found at: https://colab.research.google.com/github/mrknorman/data_ad_infinitum/blob/main/chapter_x_activation_functions.ipynb

3.1.6 Loss Functions

The loss function (sometimes cost or objective function) is an important part of the model training process. The purpose of the loss function is to act as a measure of the effectiveness of the model when acting on a particular batch of examples. In doing this, the loss function gives us a metric to evaluate the performance of the model, compare it against other models, and act as a guide during the training process. In specific cases, it can also act to regularise the model to prevent overfitting or to balance multiple objectives.

In supervised learning, this loss function is some measure of the distance between the model's output and the ground truth labels of the examples fed into the model. These are one of the more common types of loss functions, but it should be noted that as long as it is differentiable, a great many terms

can be included as part of the loss function, and indeed some of the more complex architectures have complex loss functions.

In unsupervised learning, Autoencoders are a formulation of a regression problem where the model input is equal to the model output, and thus, they follow the same principles as the typical regression problem, only the difference between their output, $\vec{\hat{y}}$, and their input, $\vec{x}$ in the loss, rather than an external label, $\vec{y}$. Clustering, on the other hand, attempts to split a data distribution into groups by minimising the distance between elements in a given group while maintaining some balance with the number of groups generated — there are a variety of different ways to do this depending on your desired outcome.

3.1.6.1 Binary Cross Entropy

The Binary Cross Entropy loss is used primarily for binary classification problems wherein each class label is independent. This can be the case either for single-class single-label tasks (binary classification tasks) or multi-class multi-label tasks. It is defined by Equation 14.

$$L\left(\vec{y}, \vec{\hat{y}}\right) = -\sum_{i=1}^{N} y_i \log\left(\hat{y}_i\right) - (1 - y_i) \log\left(1 - \hat{y}_i\right) \tag{3.14}$$

where $L\left(\vec{y}, \vec{\hat{y}}\right)$ is the loss function applied to the model output and ground truth vectors, N, is the number of elements in the output vector, y_i is the i$^{\text{th}}$ element of the ground truth vector, and $\hat{y}_i$ is the i$^{\text{th}}$ element of the ground truth vector.

In the single-class single-label case where N = 1, Equation 14 becomes

$$L(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y}). \tag{3.15}$$

Some confusion can arise in the case of binary classification problems, wherein the examples can either be in a class or not in that class since this is the same as the situation where there are two distinct classes. As such, these problems can be treated in two ways, either with a single output neuron and an output vector, $\vec{y}$ of length one, (an output value i.e. $\vec{y} = y$), where a high value indicates inclusion in the one class and a low-value exclusion, or with two output neurons where each neuron represents a "class", one being inside the class and the other being outside the class.

In the first case, we would use a sigmoid activation function and a binary cross-entropy loss, and in the second case, you would use a softmax activation function and categorical cross-entropy loss. These produce very similar outcomes, with the first method being slightly more straightforward, giving a directly interpretable output and reducing the number of parameters, whereas the second case, whilst increasing the model parameter count, can sometimes be more numerically stable.

3.1.6.2 Categorical Cross Entropy

Categorical Cross Entropy loss is very similar to binary cross-entropy loss but is used primarily in multi-class single-label problems, such as the problem we presented in the MNIST [35] classification task. It is a highly effective loss function, and it is often much easier to classify data into one class using this method than it would be to find multiple labels in a multi-class multi-label problem. So this kind of task is often a desirable framing of your problem. The loss is given by

$$L\left(\vec{y}, \vec{\hat{y}}\right) = -\sum_{i=1}^{N} y_i \log\left(\hat{y}_i\right), \tag{3.16}$$

where $L\left(\vec{y}, \vec{\hat{y}}\right)$ is the loss function applied to the model output and ground truth vectors, N is the number of elements in the output vector, y_i is the i$^{\text{th}}$ element of the ground truth vector, and $\hat{y}_i$ is the i$^{\text{th}}$ element of the ground truth vector.

Both binary cross entropy and categorical cross entropy are loss functions that attempt the measure the difference between probability distributions. In the case of binary cross-entropy, it treats each output element as a separate probability distribution, whereas for categorical cross-entropy, the entire output vector is treated as one probability distribution.

They are derived from the concept of entropy in information theory, which quantifies the expected amount of information from a source. Lower information states will have numbers that are closer to one or zero — in that way minimising the function forces the output to values of one or zero, i.e., toward definite yes/no classifications.

3.1.6.3 Mean Square Error

For regression tasks, wherein the output vectors are not limited to boolean values, we must have more flexible activation and loss functions. In these cases, we still want to compare our desired output to our actual output, but we don't want to encourage the output to values near zero and one. There are a number of options to achieve this goal, the choice of which will depend on the specifics of your problem.

One option is mean square error loss, the sum of the squares of the error, $\vec{y} - \vec{\hat{y}}$, normalised by the number of elements in the output vector. It is defined by

$$L_{\text{MSE}}\left(\vec{y}, \vec{\hat{y}}\right) = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \hat{y}_i\right)^2, \tag{3.17}$$

where $L_{\text{MSE}}\left(\vec{y}, \vec{\hat{y}}\right)$ is the loss function applied to the model output and ground truth vectors, N is the number of elements in the output vector, y_i is the i$^{\text{th}}$ element of the ground truth vector, and $\hat{y}_i$ is the i$^{\text{th}}$ element of the ground truth vector.

Mean square error is a good choice for regression problems; it is fully differentiable, unlike mean absolute error; however, unlike mean absolute error, it heavily emphasises outliers which can be beneficial or detrimental depending on your scenario.

3.1.6.4 Mean Absolute Error

The mean absolute error can be used in the same problems that the mean square error is used for. Again it is normalised by the total sum of the output vector. It is given by

$$L_{\text{MAE}}\left(\vec{y}, \vec{\hat{y}}\right) = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|, \tag{3.18}$$

where $L_{\text{MSE}}\left(\vec{y}, \vec{\hat{y}}\right)$ is the loss function applied to the model output and ground truth vectors, N is the number of elements in the output vector, y_i is the i$^{\text{th}}$ element of the ground truth vector, and $\hat{y}_i$ is the i$^{\text{th}}$ element of the ground truth vector.

Unlike mean squared error, it has a non-differentiable point at zero where the gradient must be artificially replaced, which is not a particularly elegant solution. Mean absolute error punishes small errors more than mean squared error, but large errors less, which can be a desired trait in a model training procedure.

3.1.6.5 Huber

Huber loss is an attempt to combine the benefits of both mean square error and mean absolute error and remove some of their respective disadvantages. It uses a combination of both methods to achieve differentiability at all points whilst removing mean squared error's large penalty to outliers. It does, however, introduce a new user-tuned hyperparameter δ, which, as has been discussed, is never ideal. It is defined by

$$L_\delta\left(\vec{y}, \vec{\hat{y}}\right) = \frac{1}{N}\sum_{i=1}^{N}\begin{cases} 0.5\left(y_i - \hat{y}_i\right)^2 \text{ if } |y_i - \hat{y}_i| \leq \delta \\ \delta\ |y_i - \hat{y}_i| - 0.5\delta^2 \text{ if } |y_i - \hat{y}_i| > \delta \end{cases}, \tag{3.19}$$

where $L_\delta\left(\vec{y}, \vec{\hat{y}}\right)$ is the loss function applied to the model output and ground truth vectors, N is the number of elements in the output vector, δ is a user-tuned hyperparameter which controls how much of the loss function obeys mean squared error and how much obeys mean absolute error, y_i is the i$^{\text{th}}$ element of the ground truth vector, and $\hat{y}_i$ is the i$^{\text{th}}$ element of the ground truth vector.

The choice of loss function for regression problems is very much problem-dependent and discoverable only through intuition about the dataset or failing that through investigation.

3.1.7 Network Design

The choice of loss function is largely down to the problem being attempted and, as such, is often correlated with an associated output layer activation function; see Table 1.

Problem	Example Label	Activation Function	Loss Function
Single-class Single-label Classification	[1] or [0, 1]	Sigmoid or Softmax	Binary or Categorical Cross Entropy
Multi-class Single-label Classification	[0, 1, 0, 0]	Softmax	Categorical Cross Entropy
Multi-class Multi-label Classification	[0, 1, 0, 1]	Sigmoid	Binary Cross Entropy
Regression including Autoencoders	[0.12, -1.34]	Often ReLU or Linear	Often MSE, MAE, or Huber

Table 1 | Problems often solvable by artificial neural networks and their associated activation and loss functions. This table demonstrates the most commonly used activation and loss functions for several common problem types that machine learning attempts to solve. The activation functions listed are described in Section 3.1.5, whereas the loss functions were described in this section Section 3.1.6. MSE is an abbreviation of Mean Squared Error, and MAE is an abbreviation of Mean Absolute Error.

3.2 The Gradients Must Flow

Without a method to find useful parameters, artificial neural networks are useful for little more than hopelessly scrambling input data. As mentioned previously, this method is gradient descent, using the local gradient of model parameters to find a path toward the minimum loss.

It is useful to imagine the entirety of all possible model parameters forming a surface defined by the model's loss function. Every combination of parameters is a coordinate on this highly dimensional landscape, where the corresponding loss function tells us the height of the land. We can abstract away all this high dimensionality and reduce the problem to a two-dimensional mountain range, as long

as we keep in mind that, in actuality, the number of dimensions of our surface equals the number of parameters we are fitting to.

Imagine we are lost in these Parameter Peaks. Our goal is to find the lowest point in this mountain range, for that will be the location of our helicopter rescue. In the same way that we can only calculate the gradient for one set of parameters at a time, we can only see the ground right beneath our feet; it is a very foggy day on the peaks. If we were to try to get to the lowest point, naturally, what we would do is look at the ground, examine the slope, and walk downhill; the same is true for gradient descent.

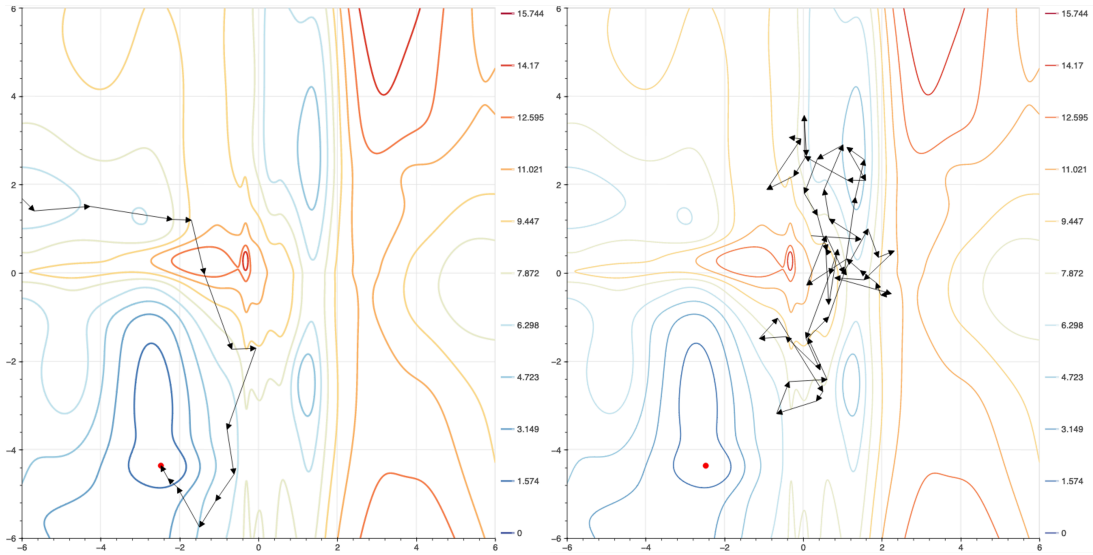


Figure 17 | *Left:* An idealised gradient descent path. This gradient descent process quickly reaches the true function minimum where, in this case, the loss is close to zero. However, this is a constructed example by first finding a point near the function minimum and performing a gradient ascent operation. *Right:* A more realistic gradient descent path. This example shows a simple but real gradient descent function running on the cost function. As can be seen, it takes many more steps and has not yet converged on the true minimum; in fact, the process might be at risk of getting stuck in a local minimum. Both examples were generated with using this notebook: https://github.com/mrknorman/data_ad_infinitum/blob/main/chapter_x_gradient_descent.ipynb.

This works perfectly if all gradients point toward the bottom, what is called a convex parameter space with one global minimum and no local minima. Parameter peaks, however, can often have some nasty tricks in store. Just like real-life mountain ranges, there can be local minima. Divits and valleys look like they might be the bottom of the mountain, but without some global information, it is impossible to tell. The parameter space is non-convex. Thus, we must explore before we settle on our final choice, moving up and down smaller hills, generally moving toward lower regions, but always searching for better outcomes — that is, until our time runs out, model training ends, and we must make a final decision about where to await the helicopter. Although perhaps if our training lasts multiple epochs, we'll have multiple days to figure it out, taking new paths each time.

There are, perhaps unsurprisingly, a number of different algorithms to achieve this purpose beyond the naive descent suggested by Equation 6, which could leave you stuck blindly in a divot on the top of a mountain whilst wide chasms stretch unseen before you. The first misconception to correct beyond what has been previously discussed is that usually when performing gradient descent, it is not performed one example at a time but rather in batches of N_{batch} examples, the gradient of which is calculated simultaneously. This N_{batch} adds a further user-adjustable hyperparameter, batch size,

N_{batch}, to our growing collection of hyperparameters. It also creates a distinction between three distinct gradient descent modes.

Stochastic gradient descent is the method that was illustrated in Section 3.1.2. This involves updating the model parameters based on the gradient of a single example at a time, looping through every example in the dataset one at a time, $N_{\text{batch}} = 1$.

Stochastic gradient descent can converge faster than other methods as it updates the parameters as frequently as possible. This is more useful with larger datasets that cannot fit into memory, as it can make progress long before it has seen all the data, and ample data will help it converge on a correct solution. Stochastic gradient descent does introduce a lot of noise into the dataset as the smoothing effects from averaging across examples are not present. This has advantages and disadvantages. The noise can help prevent the descent from getting stuck in the local minima, but by the same process, it can struggle to settle in even the true minimum, and convergence can take a long time. It can also be slow, as gradients need to be calculated for each example sequentially.

Mini-batch descent is perhaps the most commonly used of the gradient descent paradigms. In contrast to stochastic gradient descent, the gradient is calculated for multiple examples simultaneously. Unlike batch descent, however, it does not calculate the gradient for the entire dataset at once. The only restraints, therefore, is that the batch size, N_{batch}, must be larger than one and smaller than the number of elements in your training dataset, $N_{\text{batch}} > 1 \wedge N_{\text{batch}} < N_{\text{dataset}}$. This number is usually a lot smaller than the size of the entire dataset, however, with power of two values around 32 being commonplace.

This method can produce a more stable convergence than stochastic descent. Because it averages the gradient over many examples at once, there is less noise. It is a compromise between batch and stochastic descent, and its strengths and weaknesses depend largely on the batch size you select. This is also one of its largest downsides; any additional hyperparameter is one more factor that has to be tuned by some other external method.

Batch descent occurs when the gradient is calculated for all examples in our training dataset simultaneously, $N_{\text{batch}} = N_{\text{dataset}}$; it is, therefore, in some ways, the purest form of gradient descent, as the gradient has been calculated with all available data included. It will have the most stable and direct convergence of all the methods. However, whilst sometimes producing good results, this can suffer from problems getting stuck in local minima, as there is no ability for exploration. It also has the considerable downside of being very computationally expensive. For very large training datasets, this could quickly become time and computationally impossible. This method is rarely used in modern machine learning due to infeasibly large training datasets.

What follows is a brief explanation of various optimisation algorithms that can be used during the training process. The choice of optimiser can again be considered another hyperparameter that must be externally selected.

3.2.1 Momentum

In order to avoid local minima and introduce more exploration to our training process, many optimisers introduce the concept of "momentum" to the descent process. This cannot be applied to batch gradient descent since there is only one step in the process.

Adding momentum to a descent algorithm is quite literally what it sounds like; if we consider the descent process to be a ball rolling down a hill, momentum is a property that changes more slowly than the gradient of the terrain beneath it. In that way, it acts to smooth the inherent noise generated from gradient descent by adding a proportion of the previous gradient to the determination of the next

parameter space step. This can help improve convergence and prevent progress from getting stuck in a local minimum.

In order to describe this process mathematically, we introduce the concept of a parameter space velocity, $v_\theta(t)$, which is recorded independently of parameter space position, i.e. the parameter values themselves, $\vec{\theta}$. The two equations that fully describe the descent are

$$v_\theta(t) = \alpha v_\theta(t-1) + \eta \vec{\nabla} L_{M\vec{x}_t\vec{y}_t} \tag{3.20}$$

and

$$\vec{\theta}_{t+1} = \vec{\theta}_t - v_\theta(t), \tag{3.21}$$

where t is the current batch index, $v_{\theta(t)}$ is the parameter velocity at the current batch, $v_{\theta(t-1)}$, is the parameter velocity at the previous batch (initialized to 0 at $t-1$), α is the momentum parameter, η is the learning rate, $\vec{\nabla} L_{M\vec{x}_t\vec{y}_t}$, is the gradient of the model parameters with respect to the loss function, $\vec{\theta}_{t+1}$, are the updated model parameters, and $\vec{\theta}_t$ are the model parameters at the current step. As with the previous training steps, this process can be used for either stochastic or mini-batch descent and will be repeated across all training examples or batches of training examples in the training data set. The momentum parameter is a newly introduced hyperparameter which must be set before the initiation of training. The momentum value indicates what fraction of the previous parameter velocity is added to the current velocity; for any valid descent algorithm, this must be below one, $\alpha < 1$, as otherwise, the velocity will grow unbounded with each step. Common choices for momentum values hover around 0.9.

Momentum can be combined with stochastic or mini-batch descent and is an important aspect of other gradient techniques, including RMSProp and Adam.

3.2.2 AdaGrad (Adaptive Gradient Algorithm)

In standard gradient descent, every parameter, θ_i, within your parameter vector, $\vec{\theta}$, is treated equally by the descent algorithm. We can, however, imagine scenarios where treating all parameters equally is not the ideal method. A given training dataset may not contain an equal representation of all features present in that dataset. Indeed, even individual examples may have some features that are much more common than others. Often, these rarer features can be crucial to the efficient tuning of the network. However, the parameters that represent these features might see far fewer updates than other parameters, leading to long and inefficient convergence.

To combat this problem, AdaGrad, or the adaptive gradient algorithm, was introduced. This method independently modifies the learning rate for each parameter depending on how often it is updated, allowing space parameters more opportunity to train. It achieves this by keeping a record of the previous sum of gradients squared and then adjusting the learning rate independently by using the value of this record. This is equivalent to normalising the learning rate by the L2 norm of the previous gradients. This approach is defined by

$$\vec{g}_t = \vec{g}_{t-1} + \left(\vec{\nabla} L_{M\vec{x}_t\vec{y}_t}\right)^{\circ 2} \tag{3.22}$$

and

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \left(\frac{\eta}{\left(\vec{g}_t + \varepsilon\right)^{\circ \frac{1}{2}}}\right) \circ \vec{\nabla} L_{M\vec{x}_t\vec{y}_t} \tag{3.23}$$

where t is the current batch index, $\vec{g}_t$ is a vector containing the sum of the square of all parameter gradients up to the training iteration, t, $\vec{g}_{t-1}$ is the sum of the square of all parameter gradients except the current gradient squares, $\vec{\nabla} L_{M\vec{x}_t\vec{y}_t}$ is a vector containing the gradients for each parameter at the current iteration, $\vec{\theta}_{t+1}$ are the parameters at the next iteration, $\vec{\theta}_t$ are the parameters at the current iteration, and ε is a very small value to prevent division by zero errors in the calculation.

This method has the advantage of self-tuning the learning rate for individual parameters, removing the need for manual per-parameter tuning, and it helps the model update sparse parameters more quickly by increasing the learning rate for parameters which learn more rarely seen features. These small features are often very important for whatever operation is being optimised for.

AdaGrad still leaves the global learning rate, η, as an open hyperparameter which must be user-defined. It can also lead to problems when training deep networks with many layers. In a similar manner, the vanishing gradient problem can lead to tiny parameter updates when calculating the gradient of the network through very deep networks. The vanishing learning rate problem can arise when training using AdaGrad with very large training datasets. In models with large amounts of parameters, it is crucial that the parameters continue to be updated throughout the training process to ensure that all of the many parameters meet optimally tuned values. However, if the normalisation factor, $\vec{g}$ for some parameters, grows too big over the training process, the gradient updates can become very small, and training can slow to a crawl. Root Mean Square Propagation is a proposed solution to this problem.

3.2.3 RMSProp (Root Mean Square Propagation)

RMSProp, or root mean square propagation, is an alternative method to solve the adaptive learning rate issue, which attempts to alleviate the vanishing learning rate problem by less aggressively normalising the learning rate. Instead of using the L2 Norm of all previous gradients to normalise each parameter learning rate, like AdaGrad, it uses a moving average of the squared gradients. This also deals with non-convex scenarios better, as it allows the gradient descent to escape without the learning rate falling to tiny values. This process is described by

$$\vec{E}_{g^2}(t) = \beta \vec{E}_{g^2}(t-1) + (1-\beta)\left(\vec{\nabla} L_{M\vec{x}_t\vec{y}_t}\right)^{\circ 2} \tag{3.24}$$

and

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \left(\frac{\eta}{\left(\vec{E}_{g^2}(t) + \varepsilon\right)^{\circ \frac{1}{2}}}\right) \circ \vec{\nabla} L_{M\vec{x}_t\vec{y}_t}, \tag{3.25}$$

where t is the current batch index, $\vec{e}_{g^2}(t)$ is the moving average of parameter gradients squared with respect to the loss function, β is the decay rate for the moving average, which controls how quickly the effect of previous gradients on the current learning rate falls off, $\vec{\nabla} L_{M\vec{x}_t\vec{y}_t}$ is a vector containing the gradients for each parameter at the current iteration, $\vec{\theta}_{t+1}$ are the parameters at the next iteration, $\vec{\theta}_t$ are the parameters at the current iteration, and ε is a very small value to prevent division by zero errors in the calculation.

This is a similar method to AdaGrad, so it has many of the same strengths and weaknesses but alleviates the vanishing gradient problem. It also introduces one new hyperparameter, the decay rate, β, which must be decided, and it does not necessarily completely eradicate the vanishing gradient problem in all situations.

3.2.4 Adam (Adaptive Moment Estimation),

Adam (Adaptive Moment Estimation) combines the advantages of AdaGrad and RMSProp. Instead of normalising by the L2 loss alone, like AdaGrad, or the moving squared average alone, like RMSProp, it uses an exponential of the moving average of both the gradient, $E_g(t)$ and the squared gradient, $E_{g^2}(t)$ and uses the parameters, β_1 and β_2 to control the decay rates of these averages respectively. The moving average of the gradient and the moving average of the squared gradient are

$$\vec{E}_g(t) = \beta_1 \vec{E}_{g(t-1)} + (1-\beta_1)\vec{\nabla} L_{M\vec{x}_t\vec{y}_t} \tag{3.26}$$

and

$$\vec{E}_{g^2}(t) = \beta_2 \vec{E}_{g^2}(t-1) + (1-\beta_2)\left(\vec{\nabla} L_{M\vec{x}_t\vec{y}_t}\right)^{\circ 2}. \tag{3.27}$$

As with previous methods, both moving average values are initialised to vectors of zeros at the start of the descent. This poses an issue as early steps would be weighted toward zero. In order to solve this, the algorithm introduces two new terms, $\hat{E}_g(t)$, and $\hat{E}_{g^2}(t)$, to correct this issue:

$$\vec{\hat{E}}_g(t) = \vec{E}_g \frac{t}{1-(\beta_1)^t} \tag{3.28}$$

and

$$\vec{\hat{E}}_{g^2}(t) = \vec{E}_{g^2} \frac{t}{1-(\beta_2)^t}. \tag{3.29}$$

These terms are then collected in Equation 30.

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \eta \vec{\hat{E}}_g \circ \frac{t}{\left(\vec{\hat{E}}_{g^2}(t) + \varepsilon\right)^{\circ \frac{1}{2}}} \tag{3.30}$$

where t is the current batch index, $E_g(t)$ is the moving average of parameter gradients with respect to the loss function, $E_{g^2}(t)$ is the moving average of parameter gradients squared with respect to the loss function, β_1 and β_2 are the decay rate for the moving average and the moving squared averages respectively, which controls how quickly the effect of previous gradients on the current learning rate falls off, $\vec{\nabla} L_{M\vec{x}_t\vec{y}_t}$ is a vector containing the gradients for each parameter at the current iteration, $\vec{\theta}_{t+1}$ are the parameters at the next iteration, $\vec{\theta}_t$ are the parameters at the current iteration, and ε is a very small value to prevent division by zero errors in the calculation.

The Adam optimiser can intuitively be thought of as combining the adaptive learning rate methods with a form of momentum. $E_g(t)$ carries the first moment, the momentum of the past gradients, which, like momentum, will keep you moving in the general direction that you have been travelling, moderated by the β_1 parameter. E_{g^2} carries information about the second moment, which remembers the magnitude of the gradients. This will make the algorithm move more cautiously if it has been encountering steep gradients, which can normally cause large learning rates and make the optimiser overshoot. This can act as a break to the momentum built up in the first moment. The β_2 parameter moderates this aspect.

The Adam optimiser is perhaps the most widely known and widely used optimiser in modern artificial neural network training due in large part to its efficacy. Although there have been many adaptations and variants of the Adam optimiser which have tried to improve its operation, none have been so successful as to overthrow its position as the standard choice for gradient descent algorithms.

3.2.5 Backpropagation

So far, we have been using the parameter gradient vector, $\vec{\nabla} L_{M\vec{x}_t\vec{y}_t}$, without considering how we might calculate this value.

In the case of a single-layer perceptron, this process is not particularly difficult. As discussed before, first, we must pass an example (or batch of examples) through a randomly initiated network. This network, though untuned, will still produce an output vector, $\hat{y}$, albeit a useless one. We can then work backwards from the model output, $\hat{y}$, and, in the case of supervised learning, compare it to our desired output, y, by using the loss function, L. We can do this by applying the chain rule for the weights.

Let's work through an example of how we might do this for a simple single-layer perceptron, with parameters, $\vec{\theta}$, split into a weights matrix, W, and bias vector, $\vec{b}$.

The action of the model is defined by:

$$M(\vec{x}) = f(\vec{z}) \tag{3.31}$$

where $\vec{z} = W\vec{x} + \vec{b}$ is the raw input to the activation function, and f is the activation function. The i^{th} element of the output is given by the softmax function of the raw input, $\vec{z}$:

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}} \tag{3.32}$$

and the loss function is given by

$$L = -\sum_{i=1}^{N} y_i \log\left(\hat{y}_i\right), \tag{3.33}$$

where L is the loss function, N is the number of elements in the output vector and $\hat{y}_i$ is the i^{th} element of the output vector. We want to find the gradients of the model parameters with respect to the loss function. In this case, $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$. We can start by using the chain rule to compute $\frac{\partial L}{\partial z_i}$, the derivative of the loss with respect to the i^{th} component of z:

$$\frac{\partial L}{\partial z_i} = \sum_{j=1} \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial z_i} \tag{3.34}$$

Here, $\frac{\partial L}{\partial y_j}$ is the derivative of the loss with respect to the j^{th} output, and $\frac{\partial y_j}{\partial z_i}$ is the derivative of the j^{th} output with respect to the i^{th} input before activation. In our case, because we are using categorical cross-entropy loss:

$$\frac{\partial L}{\partial y_j} = \frac{\hat{y}_j}{y_j} \tag{3.35}$$

And, due to the softmax activation function, in which the value of all output neurons affects the gradient of all others,

$$\frac{\partial \hat{y}_j}{\partial z_i} = \begin{cases} \hat{y}_j\left(1 - \hat{y}_j\right) \text{ if } i = j \\ -\hat{y}_j \hat{y}_i \text{ if } i \neq j \end{cases} \tag{3.36}$$

Substitution of Equation 35 and Equation 36 into Equation 34 gives

$$\frac{\partial L}{\partial z_i} = -\frac{y_i}{\hat{y}_i}\hat{y}_i\left(1 - \hat{y}_i\right) + \sum_{j \neq i} \frac{y_j}{\hat{y}_j}\left(-\hat{y}_j \hat{y}_i\right). \tag{3.37}$$

Simplifying gives:

$$\frac{\partial L}{\partial z_i} = -y_i\left(1 - \hat{y}_i\right) + \sum_{j \neq i} -y_j \hat{y}_i. \tag{3.38}$$

We can simplify this further because $\sum_j y_j = 1$, as the input label is a one-hot vector and will always sum to one:

$$\frac{\partial L}{\partial z_i} = y_i - \hat{y}_i. \tag{3.39}$$

This shows that the derivative of the softmax function with respect to the sum of the weighted inputs and bias values, $\frac{\partial L}{\partial z_i}$, is equal to the difference between the ground truth label value and the model output value. This provides us with another insight into the design of the softmax function and its use of exponentials.

We can then again use the chain rule to find the gradient of the weights and biases

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \vec{z}} \frac{\partial \vec{z}}{\partial W} = \left(\vec{y} - \vec{\hat{y}}\right) \circ \vec{x} \tag{3.40}$$

and

$$\frac{\partial L}{\partial \vec{b}} = \frac{\partial L}{\partial \vec{z}} \frac{\partial \vec{z}}{\partial \vec{b}} = y - \vec{\hat{y}}. \tag{3.41}$$

Both of the gradients, Equation 40 and Equation 41, are quite intuitively what you might expect from a single-layer network. There is no non-linear behaviour, and as we previously speculated, the network is just training to find pixels that are most often activated by certain classes.

We can use a similar method for artificial neural networks of all complexities and depths. For a feed-forward dense network with N layers, let us denote the weighted sums of the inputs plus the biases of a layer with index i, as $\vec{z}_i$, the output of the activation function, f of layer i as $a_i = f(z_i)$, the weights matrix and biases vector of layer i as W_i and $\vec{b}_i$, and the loss function again as L.

First, we compute the forward propagation by running an input vector, $\vec{x}$, or batch of input vectors, through the network to produce an output vector $\vec{\hat{y}}$. Then follow the following procedure.

1. Compute the derivative of the loss function with respect to the final output values: $\frac{\partial L}{\partial a_N} = \frac{\partial L}{\partial \hat{y}}$.
2. Compute $\frac{\partial L}{\partial z_N} = \frac{\partial L}{\partial a_N} \frac{\partial a_N}{\partial z_N}$, where $\frac{\partial a_N}{\partial z_N}$ is the derivative of the activation function in the final layer. This gives the gradient of the loss function with respect to the final raw outputs, $\vec{z}_N$.
3. Compute $\frac{\partial L}{\partial W_N} = \frac{\partial L}{\partial z_N} \frac{\partial z_N}{\partial W_N}$ and $\frac{\partial L}{\partial b_N} = \frac{\partial L}{\partial z_N} \frac{\partial z_N}{\partial b_L}$. This gives the gradients with respect to the final layer's weights and biases.
4. To propagate the error back to the previous layer, compute $\frac{\partial L}{\partial a_{N-1}} = \frac{\partial z_N}{\partial a_{N-1}} \frac{\partial L}{\partial z_N} = W_N^T \frac{\partial L}{\partial z_N}$.
5. Recursively repeat steps 1 to 4 until you reach the input layer and you have gradients for all parameters.

This method is known as backpropagation because you work backwards from the output of the model toward the input vector.

Missing overfitting, test, train validate datasets, regularisation, false-alarm rates, specificity ect.

4 Application to Gravitational Waves

We have demonstrated that simple artificial neural networks can be used to classify input data drawn from a restricted distribution into a number of classes, N, with a high ($> 99.9\%$) degree of accuracy. Because we didn't design the network with any consideration for the dataset (besides its dimensionality), we can infer that this method should be general enough to classify data drawn from other distributions that contain discrete differentiable classes. It is not clear, however, which other distributions can be classified and what magnitude of the network is required to achieve a similar degree of accuracy. It is easy to imagine distributions that are considerably simpler than the MNSIT dataset and, conversely, ones that are much more complex. There may be a mathematical approach to determine the link between the distribution and required model complexity. One possible metric that touches upon this relation is the Rademacher complexity, $\hat{\mathcal{R}}_M$, given by

$$\hat{\mathcal{R}}_M(H) = \mathbb{E}_{\in}\left[\frac{1}{M}\sum_{i=1}^{M}\varepsilon_i h[\vec{x}_i]\right], \tag{42}$$

where M is the number of data points in a dataset $X = [\vec{x_1}, ..., \vec{x}_i, ..., \overrightarrow{x_M}]$ where each point is a vector, $\vec{x}$, in our case the input vectors of our training dataset, $\vec{\varepsilon} = [\vec{\varepsilon_1}, ..., \vec{\varepsilon}_i, ..., \overrightarrow{\varepsilon_M}]$ uniformly distributed in $\{-1, +1\}^m$, and H is a real-valued function class, in our case the set of functions that can be approximated by our chosen neural network architecture. The Rademacher complexity is a measure of how well functions in H can fit random noise in the data. A higher Rademacher complexity indicates that the function class can fit noise better, which implies a higher capacity to overfit the data. So, one approach to optimising the model would be to attempt to minimise this value whilst maximising model performance. More details about this metric and its use in defining the relationship between data samples and model complexity can be found at [37]. Despite the existence of this metric, however, it would appear that there has not been substantial research into the link between dataset complexity and required model size [38], though it is possible that such a paper has been missed.

One method that we can use to explore this question is to find out the answer empirically. As we move from the MNIST dataset to distributions within gravitational-wave data science, the natural starting point is to repeat the previous experiments with gravitational-wave data, both as a comparison and as a baseline as we move forward. The following subsection will explore the possible areas for detection applications, describe the example datasets and their construction, and explore the results of repeating the previous experiments on gravitational-wave data, comparing our results to similar attempts from the literature. By the end of this chapter, we will have accumulated a large number of possible network, training, and data configurations. We will perform a series of tests to begin to narrow the search space of possible hyperparameters.

4.1 Gravitational-Wave Classifiers

The scope of gravitational-wave data problems to which we can apply artificial neural network models is large. However, we shall limit our investigation to perhaps the simplest type of problem — classification, the same type of problem that we have previously demonstrated with our classification of the MNIST dataset into discrete classes. Though classification is arguably the most straightforward problem available to us, it remains one of the most crucial — before any other type of transient signal analysis can be performed, transients must first be identified.

There are several problems in gravitational-wave data analysis which can be approached through the use of classification methods. These can broadly be separated into two classes of problems — detection and differentiation. **Detection** problems are self-explanatory; these kinds of problems require the identification of the presence of features within a noisy background. Examples include Compact Binary Coalescence (CBC), burst, and glitch detection; see Figure 18 for a representation of the differ-

ent features present in gravitational-wave data. **Differentiation** problems, usually known simply as classification problems, involve the separation of detected features into multiple classes, although this is often done in tandem with detection. An example of this kind of problem is glitch classification, in which glitches are classified into classes of known glitch types, and the classifier must separate input data into these classes.

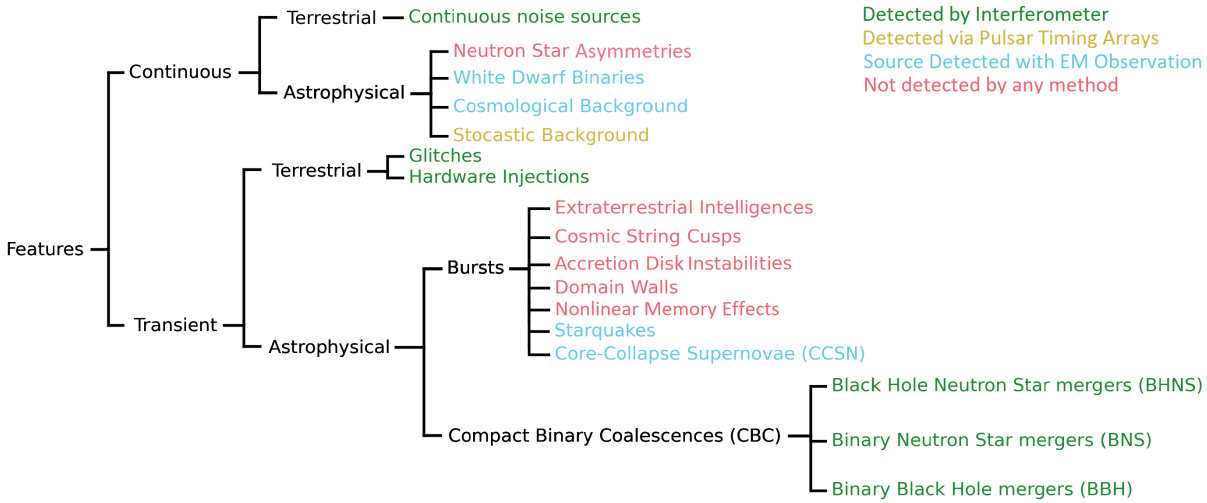


Figure 18 | A non-exhaustive hierarchical depiction of some of the features, and proposed features, of gravitational-wave interferometer data. The first fork splits the features into two branches, representing the duration of the features. Here, **continuous** features are defined as features for which it is extremely unlikely for us to witness their start or end within the lifespan of the current gravitational-wave interferometer network and probably the current scientific community [39]. These features have durations anywhere from thousands to billions of years. **Transient** features have comparatively short durations [1], from fractions of seconds in the case of stellar-mass Binary Black Hole (BBH) mergers [1] to years in the case of supermassive BBH mergers [40]. It should be noted that the detectable period of supermassive BBH binaries could be much longer; although the mergers themselves are transient events, there is no hard cut-off between the long inspiral and merger event. Nevertheless, the mergers are probably frequent enough that some will end within the lifetime of the proposed LISA space probe, so they can be considered transients [40]. The next fork splits features by origin. Features of **astrophysical** origin originate from beyond Earth. This distinction is practically synonymous with the distinction between gravitational waves and signals from other sources since no other astrophysical phenomena are known to have a similar effect in interferometers [1]. Features of **terrestrial** origin, unsurprisingly, originate from Earth. These primarily consist of detector glitches caused by seismic activity or experimental artefacts [41]. Astrophysical transients have a further practical division into CBCs and bursts. The category of **bursts** contains all astrophysical transients that are not CBCs [42]. The primary reason for this distinction is that CBCs have been detected and have confirmed waveform morphologies. As of the writing of this thesis, no gravitational-wave burst events have been detected [42], [43], [44] . Bursts often require different detection techniques; of the proposed sources, many are theorised to have waveforms with a much larger number of free parameters than CBCs, as well as being harder to simulate as the physics are less well-understood [45], [46] . These two facts compound to make generating large template banks for such signals extremely difficult. This means that coherence detection techniques that look for coherent patterns across multiple detectors are often used over matched filtering [47], [48], [49], [50], [51] . The astrophysical leaves of the diagram represent possible and detected gravitational-wave sources; the text's colourings represent their current status. Green items have been detected using gravitational-wave interferometers, namely the merger of pairs of Binary Black Holes (BBHs) [1], Binary Neutron Stars (BNSs) [52], or one of each (BHNSs) [53]; see [54],

[55], [56] for full catalogues of detections. Yellow items have been detected via gravitational waves but using Pulsar Timing Arrays (PTAs) rather than interferometers [57]. Blue items represent objects and systems that are theorised to generate gravitational waves and have been detected by electromagnetic observatories but not yet with any form of gravitational wave detection. This includes white dwarf binaries [58], [59] , the cosmological background [60], [61] , starquakes [62], [63] , and core-collapse supernovae CCSN [64], [65] . This is because they are too weak and/or too uncommon for our current gravitational-wave detector network to have had a chance to detect them. Finally, red items are possible, theorised sources of gravitational waves that have not yet been detected by any means. These are, evidently, the most contentious items presented, and it is very possible that none of these items will ever be detected or exist at all. It should be noted that the number of proposed sources in this final category is extensive, and this is far from an exhaustive list. The presented proposed continious sources are neutron star asymmetries [66], and the presented transient sources are extraterrestrial intelligence [67], cosmic string kinks and cusps [68], accretion disk instabilities [69], domain walls [70], and nonlinear memory effects [71].

Figure 18 shows that several possible transients with terrestrial and astrophysical origins could be targeted for detection. For our baseline experiments and throughout this thesis, we will select two targets.

Firstly, **Binary Black Holes (BBHs)**. We have the most numerous detections of BBH signals, and whilst this might make them seem both less interesting and as a solved problem, they have several benefits. As test cases to compare different machine learning techniques against traditional methods, they have the most material for comparison because of their frequency; they would also see the greatest benefits from any computational and speed efficiency savings that may be wrought by the improvement of their detection methods. These factors may become especially relevant when the 3^{rd} generation detectors, such as the Einstein Telescope and Cosmic Explorer, come online. During their observing periods, they expect detection rates on the order of between 10^4 and 10^5 detections per year, which would stretch computing power and cost if current methods remain the only options. In the shorter term, if detection speeds can be improved, faster alerts could be issued to the greater astronomical community, allowing increased opportunity for multimessenger analysis. Only one multimessenger event has thus far been detected — a Binary Neutron Star (BNS) event, but it is probable, due to the relative similarity in their morphologies, that methods to detect BBHs could be adapted for BNS detection.

Secondly, we will investigate the detection of unmodeled **burst** signals using a machine learning-based coherent detection technique. Bursts are exciting sources whose detection could herald immense opportunities for scientific gain. Possible burst sources include core-collapse supernovae [65], starquakes [63], accretion disk instabilities [69], nonlinear memory effects [71], domain walls [70], and cosmic string cusps [68], as well as a plethora of other proposed sources. It should be noted that whilst many bursts have unknown waveform morphologies, some, such as cosmic string cusps, are relatively easy to model and are grouped with bursts primarily due to their as-yet undetected status.

Our current models of the physics of supernovae are limited both by a lack of understanding and computational intractability; detecting the gravitational-wave signal of a supernova could lead to new insights into the supranuclear matter density equation of state as well other macrophysical phenomena present in such events such as neutron transport and hydrodynamics. We may also detect proposed events, such as accretion disk instabilities, which may be missed by standard searches. We can search for the gravitational-wave signals of electromagnetic events which currently have unknown sources, such as fast radio bursts [72], magnetar flares [73], soft gamma-ray repeaters [74], and long gamma-ray bursts [74]. Although it's possible that some of these events produce simple, modelable waveforms,

it is not currently known, and a general search may one day help to reveal their existence. Some of the more hypothetical proposed sources could fundamentally alter our understanding of the universe, such as evidence for dark matter [75] and/or cosmic strings [68], or if we fail to find them, it could also help to draw limits on theory search space.

It is unknown whether unmodeled burst detection is a solved problem. Currently, the LIGO-Virgo-KAGRA collaboration has a number of active burst detection pipelines, X-Pipeline [47], oLIB [48], Coherent Wave Burst (cWB) [49] and BayesWave [50]. These include both offline and online searches, including targeted searches wherein a known electromagnetic event is used to limit the search space [72], [73], [74] . It could be that the current detection software is adequate and, indeed, the search is hardware rather than software-limited. Even if this is the case, there are probably computational improvements that are possible. It seems unlikely that we have reached the limit of coherent search efficiency.

Traditional coherence techniques require the different detector channels to be aligned for successful detection; therefore, because we don't know a priori the direction of the gravitational-wave sources (unless we are performing a targeted offline search), coherent search pipelines such as X-Pipeline [47] and cWB [49] must search over a grid covering all possible incidence directions. In the case of all-sky searches, this grid will necessarily cover the entire celestial sphere. In targeted searches, the grid can be significantly smaller and cover only the uncertainty region of the source that has already been localised by an EM detection. Higher resolution grids will result in a superior search sensitivity; however, they will simultaneously increase computing time. Covering the entire sky with a grid fine enough to achieve the desired sensitivity can be computationally expensive. It is possible to circumnavigate the need to search over a grid using artificial neural networks, shifting much of the computational expense to the training procedure. This has been demonstrated by the MLy pipeline [51] — the only fully machine-learning-based pipeline currently in review for hopeful deployment before the end of the fourth observing run (O4). Improvements in the models used for this task could be used to improve the effectiveness of the MLy pipeline. Indeed, some of the work discussed in this thesis was used at an early stage in the pipeline's development to help design the architecture of the models; see Section 5.8. It is hoped that in the future, more aspects of the work shown here can find use in the pipeline's development.

We will focus on the binary detection problem rather than multi-class classification, as there is only one discrete class of BBH (unless you want to draw borders within the BBH parameter space or attempt to discern certain interesting features, such as eccentricity), and in the unmodeled burst case, coherent detection techniques are not usually tuned to particular waveforms, which, in any case, are not widely available for many types of burst. In the next subsection, we will discuss how we can create example datasets to train artificial neural networks for this task.

4.2 Dataset Design and Preparation

In the case of CBCs, we have only a very limited number (< 200) of example interferometer detections, and in the burst case, we have no confirmed examples. This means that to successfully train artificial neural network models, which typically require datasets with thousands to millions of examples, we must generate a large number of artificial examples. The following subsection describes the creation of these examples, including the acquisition of noise, the generation and scaling of simulated waveforms, and data conditioning.

4.2.1 The Power Spectral Density (PSD)

The Power Spectral Density (PSD) is an important statistical property that is used by several elements of dataset design. Since a custom function was written for this thesis in order to speed up the calcu-

lation of the PSD, and since it is helpful to have an understanding of the PSD in order to understand many of the processes described in subsequent sections, a brief explanation is presented.

The PSD is a time-averaged description of the distribution of a time series's power across the frequency spectrum. Unlike a Fourier transform, which provides a one-time snapshot, the PSD conveys an averaged view, accounting for both persistent and transient features; see Equation 45 for a mathematical description. The PSD is used during data conditioning in the whitening transform, wherein the raw interferometer data is processed so that the noise has roughly equal power across the frequency domain, see Section 4.2.7. For some types of artificial noise generation, the PSD can be used to colour white noise in order to generate more physically active artificial noise; see Section 4.2.2. The PSD is also used to calculate the optimal Signal to Noise ratio, which acts as a metric that can be used to measure the detectability of an obfuscated feature and thus can be used to scale the amplitude of the waveform to a desired detection difficulty.

Imagine a time series composed of a stationary 20 Hz sine wave. In the PSD, this would materialise as a distinct peak at 20 Hz, effectively capturing the concentrated power at this specific frequency: the frequency is constant, and the energy is localised. If at some time, t, we remove the original wave and introduce a new wave at a different frequency, 40 Hz, the original peak at 20 Hzwould attenuate but not vanish, as its power is averaged over the entire time-series duration. Concurrently, a new peak at 40 Hz would appear. The power contained in each of the waves, and hence the heights of their respective peaks in the PSD, is determined by the integrated amplitude of their respective oscillations; see Figure 19 for a depiction of this example. When applied to a more complicated time series, like interferometer noise, this can be used to generate an easy-to-visualise mapping of the distribution of a time series's power across frequency space.

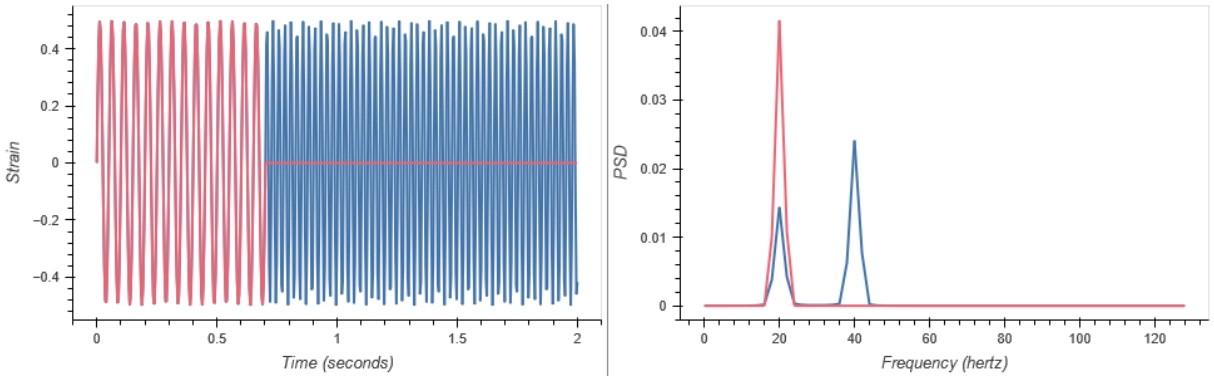


Figure 19 | Examples of Power Spectral Density (PSD) transforms. *Left:* Two time domain series. The red series is a 20 Hz wave with a duration of 0.7 s, and the blue series is this same time series concatenated with a 40 Hz wave from $t = 0.7\,s$ onwards. *Right:* The two PSDs of the time series are displayed in the left panel. The red PSD was performed across only the 0.7 s of the red wave's duration, whereas the blue PSD was taken over the full 2.0 s duration. As can be seen, the blue PSD has two peaks, representing the two frequencies of the two waves combined to make the blue time series — each peak is lower than the red peak, as they are averaged across the full duration, and their respective heights are proportional to their durations as both waves have the same amplitude and vary only in duration.

The PSD can be calculated using Welch's method, which uses a periodogram to calculate the average power in each frequency bin over time. More specifically, the following steps are enacted:

1. First, the time series is split up into K segments of length L samples, with some number of overlapping samples D; if $D = 0$, this method is equivalent to Bartlett's method.

2. Each segment is then windowed with a user-chosen window function, $w(n)$. This is done in order to avoid spectral leakage, avoid discontinuities in the data, smoothly transition between segments, and control several other factors about the method, which allow for fine-tuning to specific requirements.
3. For each windowed segment, i, we then estimate the power of the segment, $I_i(f_k)$, at each frequency, f_k, by computing the periodogram with

$$I_i(f_k) = \frac{1}{L}|X_i(k)|^2 \tag{43}$$

where $I_i(f_k)$ is the result of the periodogram, $X_i(k)$ is the FFT of the windowed segment, and f_k is the frequency corresponding to the k^{th} FFT sample.

4. Finally, we average the periodograms from each segment to get the time-average PSD:

$$S(f_k) = \frac{1}{K}\sum_{i=1}^{K} I_i(f_k) \tag{44}$$

where where $S(f_k)$ is the PSD. Combining Equation 43 and Equation 44 gives

$$S(f_k) = \frac{1}{K}\sum_{i=1}^{K} \frac{1}{L}|X_i(k)|^2 \tag{45}$$

To compute the PSD with enough computational speed to perform rapid whitening and SNR, ρ_{opt}, calculation during model training and inference, an existing Welch method from the SciPy scientific Python library [76] was adapted, converting its use of the NumPy vectorised CPU library [33] to the TensorFlow GPU library [34]; this converted code is seen in Listing 8.

```
@tf.function
def calculate_psd(
        signal : tf.Tensor,
        nperseg : int,
        noverlap : int = None,
        sample_rate_hertz : float = 1.0,
        mode : str ="mean"
    ) -> (tf.Tensor, tf.Tensor):

    if noverlap is None:
        noverlap = nperseg // 2

    signal = detrend(signal, axis=-1, type='constant')

    # Step 1: Split the signal into overlapping segments
    signal_shape = tf.shape(signal)
    step = nperseg - noverlap
    frames = tf.signal.frame(signal, frame_length=nperseg, frame_step=step)

    # Step 2: Apply a window function to each segment
    # Hanning window is used here, but other windows can be applied as well
    window = tf.signal.hann_window(nperseg, dtype = tf.float32)
    windowed_frames = frames * window

    # Step 3: Compute the periodogram (scaled, absolute value of FFT) for each
    # segment
```

```
    periodograms = \
        tf.abs(tf.signal.rfft(windowed_frames))**2 / tf.reduce_sum(window**2)

    # Step 4: Compute the median or mean of the periodograms based on the
    #median_mode
    if mode == "median":
        pxx = tfp.stats.percentile(periodograms, 50.0, axis=-2)
    elif mode == "mean":
        pxx = tf.reduce_mean(periodograms, axis=-2)
    else:
        raise "Mode not supported"

    # Step 5: Compute the frequencies corresponding to the power spectrum values
    freqs = fftfreq(nperseg, d=1.0/sample_rate_hertz)

    #Create mask to multiply all but the 0 and nyquist frequency by 2
    X = pxx.shape[-1]
    mask = \
        tf.concat(
            [
                tf.constant([1.]),
                tf.ones([X-2], dtype=tf.float32) * 2.0,
                tf.constant([1.])
            ],
            axis=0
        )

    return freqs, (mask*pxx / sample_rate_hertz)
```

Listing 8 | *Python [32]* . TensorFlow [34] graph function to calculate the PSD of a signal. `signal` is the input time series as a TensorFlow tensor, `nperseg` is the number of samples per segment, L, and `noverlap` is the number of overlapping samples, D. TensorFlow has been used in order to utilise GPU parallelisation, which offers a significant performance boost over a similar function written in NumPy [33].

A closely related property, the Amplitude Spectral Density (ASD), is given by the element-wise square root of the Power Spectral Density (PSD)

$$A(f_k) = S(f_k)^{\circ\frac{1}{2}}. \tag{46}$$

4.2.2 Noise Generation and Acquisition

There are two possible avenues for acquiring background noise to obfuscate our injections. We can either create artificial noise or use real segments extracted from previous observing runs. As was discussed in Section 2.3.1, real interferometer noise is neither Gaussian nor stationary, and many of the noise sources which compose this background are not accounted for or modelled [41]. This means that any artificial noise will only be an approximation of the real noise — it is not clear, intuitively, how well this approximation will be suited to training an artificial neural network.

One perspective argues that using more approximate noise could enhance the network's generalisation capabilities because it prevents overfitting to the specific characteristics of any given noise distribution; this is the approach adopted by the MLy pipeline [51]. Conversely, another perspective suggests that in order to properly deal with the multitude of complex features present in real noise, we should

make our training examples simulate real noise as closely as possible, even suggesting that models should be periodically retrained within the same observing run in order to deal with variations in the noise distribution. These are not discrete philosophies, and the optimal training method could lie somewhere between these two paradigms.

Evidently, in either case, we will want our validation and testing datasets to approximate the desired domain of operation as closely as possible; if they do not, we would have no evidence, other than assumption, that the model would have any practical use in real data analysis. The following subsection will outline the possible types of noise that could be used to create artificial training examples. Throughout the thesis, for all validation purposes, we have used real noise at GPS times, which are not used at any point during the training of models, even when the training has been done on real noise.

White Gaussian: The most simplistic and general approach, and therefore probably the most unlike real noise, is to use a white Gaussian background. This is as simplistic as it sounds; we generate N random variables, where N is the number of samples in our noise segment. Each sample is drawn from a normal distribution with a mean of zero and some variance according to your input scaling; often, in the case of machine learning input vectors, this would be unity; see the two uppermost plots in Figure 20.

Coloured Gaussian: This noise approximation increases the authenticity of the noise distribution by colouring it with a noise spectrum; typically, we use an ASD drawn from the interferometer we are trying to imitate in order to do this; see Section 4.2.1. By multiplying the frequency domain transform of Gaussian white noise by a given PSD, we can colour that noise with that PSD. The procedure to do this is as follows:

1. Generate white Gaussian noise.
2. Transform the Gaussian noise into the frequency domain using a Real Fast Fourier Transform (RFFT).
3. Multiply the noise frequency spectrum by your selected ASD in order to colour it.
4. Return the newly coloured noise to the time domain by performing an Inverse RFFT (IRFFT).

There are at least two choices of PSD we could use for this process. We could use the PSD of the detector design specification. It represents the optimal PSD given perfect conditions, no unexpected noise sources, and ideal experimental function. This would give a more general, idealistic shape of the PSD across a given observing run. Alternatively, we could use the PSD of a real segment of the background recorded during an observing run; this would contain more anomalies and be a closer approximation to the specific noise during the period for which the PSD was taken. Since the PSD is time-averaged, longer segments will result in more general noise. The MLy pipeline [51] refers to this latter kind of noise as **pseudo-real** noise; see examples of these noise realisations in the four middle plots of Figure 20.

Real: Finally, the most authentic type of noise that can be gathered is real interferometer noise. This is noise that has been sampled directly from a detector. Even assuming that you have already decided on which detector you are simulating, which is required for all but white noise generation, there are some extra parameters, shared with the pseudo-real case, that need to be decided. The detector data information, the time period from which you are sampling, and whether to veto any features that may be present in the segment — e.g. segments which contain events, candidate events, and known glitches.

To acquire the real data, we utilise the GWPy Python Library's [77] data acquisition functionality — since there are multiple formats in which we could retrieve the data, we must specify some parameters, namely, the frame, the channel, and the state flag. Interferometer output data is stored in a custom file format called a frame file [78]; thus, the choice of frame determines the file to be read. Within each frame file lies multiple channels — each of which contains data from a single output stream. These

output streams can be raw data, e.g. raw data from the interferometer photodetector itself; various raw auxiliary data streams, such as from a seismometer; conditioned data, e.g., the primary interferometer output with lines removed; or the state flag channel, which contains information about the status of the detector at every time increment — the state flag will indicate whether the detector is currently in observing mode or otherwise, so it is important to filter your data for the desired detector state. For the real noise used in this thesis, we use the frame, channel, and state flag, shown in Table 2. We have excluded all events and candidate events listed in the LIGO-Virgo-Kagra (LVK) collaboration event catalogues [54], [55], [56] but included detector glitches unless otherwise stated.

Detector	**Frame**	**Channel**	**State Flag**
LIGO Hanford (H1)	HOFT_C01	H1:DCS-CALIB_S-TRAIN_CLEAN_C01	DCS-ANALY-SIS_READY_C0- 1:1
LIGO Livingston (L1)	HOFT_C01	L1:DCS-CALIB_S-TRAIN_CLEAN_C01	DCS-ANALY-SIS_READY_C0- 1:1
VIRGO (V1)	V1Online	V1:Hrec_hoft_16384Hz	ITF_SCIENCE:1

Table 2 | The frame, channel, and state flags used when obtaining data from the respective detectors during the 3rd observing run (O3). This data was used as obfuscating noise when generating artificial examples to train and validate artificial neural network models throughout this thesis. It should be noted that although the clean channels were produced offline in previous observing runs, the current observing run O4 produces cleaned channels in its online run, so using the cleaned channels during model development ensures that the training, testing, and validation data is closer to what would be the normal operating mode for future detection methods.

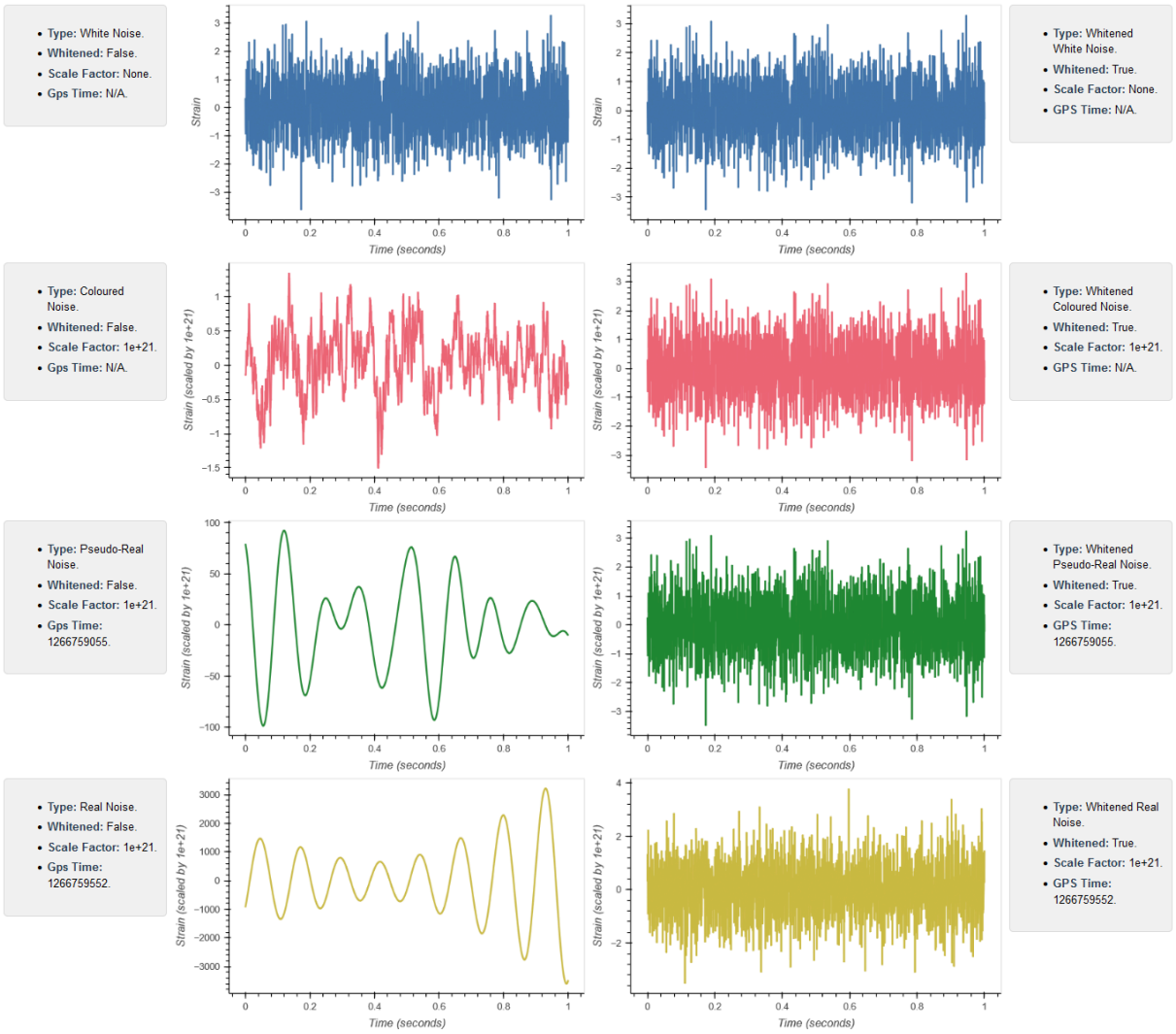


Figure 20 | One-second examples of the four possible types of simulated and real noise considered by this thesis. Where real noise is used, it is taken from the LIGO Livingston detector during the third observing run at the GPS times listed. In order, from top to bottom, these are examples of white Gaussian noise, coloured Gaussian noise, pseudo-real noise, and real noise. A description of these noise types and their generation can be found in Section 4.2.2. The left column shows the unaltered values of the noise. Note that the noise has been scaled in all cases except for the pure white noise, which is generated at the correct scale initially. This scaling is used to reduce precision errors and integrate more effectively with the machine learning pipeline, as most loss and activation functions are designed around signal values near unity; see Section 3.1.6 and Section 3.1.5. The right column shows the same noise realisations after they have been run through a whitening filter. In each case, the PSD of a 16.0 s off-source noise segment not displayed is used to generate a Finite Impulse Response (FIR) filter, which is then convolved with the on-source data; see Section 4.2.7. For the simulated and pseudo-real noise cases, the off-source data is generated using the same method as the on-source data but with a longer duration. In the real noise case, the off-source data consists of real interferometer data drawn from 16.5 s before the start of the on-source segment to 0.5 s before the start of the on-source segment. This 0.5s gap is introduced because 0.5 s must be cropped from the data following the whitening procedure in order to remove edge effects induced via windowing, as well as acting as a buffer to reduce contamination of the off-source data with any features present in the on-source data. Note that the whitened noise plots look very similar for the three simulated noise cases — a close examination of the data reveals that there is some small variation between the exact values. This similarity occurs because the off-source

and on-source noise segments for these examples are generated with identical random seeds and thus have identical underlying noise realisations (which can be seen exactly in the unwhitened white noise plot). Since the PSDs of the on-source and off-source data are nearly identical for the simulated cases, the whitening procedure is almost perfect and reverts it nearly perfectly to its white state. If anything, this similarity boosts confidence that our custom whitening procedure is operating as expected.

For our baseline training dataset used in this section, we will employ a real-noise background. An argument can be made that it is an obvious choice. It is the most like real noise, by virtue of being real noise, and thus it contains the full spectrum of noise features that might be present in a real observing run, even if it does not contain the particular peculiarities of any given future observing run in which we may wish to deploy developed models. We will experiment with different noise realisations in a future chapter Section 5.3.1.

In each case, we will acquire two seconds of data at a sample rate of 2048.0 Hz, which includes 0.5 s of data on either side of the time series, which will be cropped after whitening. The whitening is performed similarly in all cases in order to ensure symmetry when comparing obfuscation methods. A power-of-two value is used as it simplifies many of the mathematical operations which need to be performed during signal and injection processing, which may, in some cases, improve performance, as well as help to avoid edge cases that may arise from odd numbers. This frequency was selected as its Nyquist frequency of 1024.0 Hz will encompass nearly the entirety of the frequency content of BBH signals; it also covers a large portion of the search space of proposed transient burst sources. The duration of 1.0 s is a relatively arbitrary choice; however, it is one which is often the choice for similar examples found in the literature, which makes comparison easier. It also encompasses the majority of the signal power of BBH waves, as well as the theoretically detectable length of many burst sources. For each on-source noise example gathered or generated, 16.0 s of off-source background noise is also acquired to use for the whitening procedure; see Section 4.2.7.

In the case where multiple detectors are being used simultaneously during training or inference, such as coherence detection, noise is generated independently for each interferometer using the same methods, with the restriction that noise acquired from real interferometer data is sampled from each detector within a common time window of 2048.0 s so that the noise all originates from a consistent time and date. This is done as there are periodic non-stationary noise features that repeat in daily, weekly, and yearly cycles due to weather, environmental conditions, and human activity. When validating methods, we want to make our validation data as close as possible to reality whilst maintaining the ability to generate large datasets. As we are only ever training our method to operate in real noise conditions (which our validation data attempts to mimic), there is no need to deviate from this method of acquiring noise for our training datasets.

4.2.3 Waveform Generation

Once the background noise has been acquired or generated, the next step is to introduce some differentiation between our two classes, i.e. we need to add a transient signal into some of our noise examples so that our model can find purpose in its existence. When we add a transient into background noise that was not there naturally, we call this an **injection**, since we are artificially injecting a signal into the noise. This injection can be a transient of any type.

Typically, this injection is artificially simulated both due to the limited (or non-existent) number of real examples in many cases and because we will only be able to obtain the real signal through the lens of an interferometer, meaning it will be masked by existing real detector noise. If we were to inject a real injection into some other noise realisation, we would either have to perform a denoising operation (which, even when possible, would add distortion to the true signal) or inject the injection

plus existing real noise into the new noise, effectively doubling the present noise and making injection scaling a difficult task. Thus, we will be using simulated injections to generate our training, testing and validation datasets.

Luckily, this is not unprecedented, as most other gravitational-wave detection and parameter estimation methods rely on simulated signals for their operation, including matched filtering. Therefore, there is a well-developed field of research into creating artificial gravitational-wave waveform "approximants", so named because they only approximate real gravitational-wave waveforms. Depending on the complexity and accuracies of the chosen approximant and the source parameter range you are investigating, there will be some level of mismatch between any approximant and the real waveform it is attempting to simulate, even when using state-of-the-art approximants.

To simulate BBH waveforms, we will be using a version of the IMRPhenomD approximant [79], which has been adapted to run on GPUs using NVIDIAs CUDA GPU library. We name this adapted waveform library cuPhenom for consistency with other CUDA libraries such as cuFFT. More details on this adaptation can be found in Section 8. PhenomD has adjustable parameters which can be altered to generate BBHs across a considerable parameter space, although it should be noted that it does not simulate eccentricity, non-aligned spins, or higher modes. It is also a relatively outdated waveform, with the paper first published in 2015. Newer waveforms, such as those of the IMRPhenomX family, are now available. IMRPhenomD was initially chosen due to its simpler design and as a test case for the adaptation of Phenom approximants to a CUDA implementation. It would not be ideal for implementation into a parameter estimation pipeline due to its mismatch, but the accuracy requirements for a detection pipeline are significantly less stringent.

The IMRPhenomD [79] approximant generates a waveform by simulating the Inspiral, Merger, and Ringdown regions of the waveform, hence the IMR in the approximant name. The waveform is generated in the frequency domain before being transformed back into the time domain. The inspiral is generated using post-Newtonian expressions, and the merger ringdown is generated with a phenomenological ansatz; both parts of the model were empirically tuned using a small bank of numerical relativity waveforms. Detailed investigation of aproximant generation was out of the scope of this thesis and will not be covered. See Figure 21 for examples of waveforms generated using cuPhenom.

The increased performance of cuPhenom is significant and speeds up the training and iteration process of models considerably. Because of cuPhenom's ability to generate injections on the fly during the training process without significant slowdown, it allows for very quick alteration of dataset parameters for training adjustments. It was felt that this advantage outweighed any gains that would be achieved by using newer waveform models that had not yet been adapted to the GPU, as it seems unlikely, especially in the detection case, that the newer waveform models would make for a significantly harder problem for the model to solve. This statement is, however, only an assumption, and it would be recommended that an investigation is carried out to compare the differences between approximants before any of the methods are used in a real application. A final retraining with these more accurate models would be recommended, in any case.

In the case of unmodelled burst detection, the accuracy of the signal shape is not as fundamental, as the ground truth shapes are not known and, for some proposed events, cover a very large shape space [80]. In order to cover the entire search space, we have used artificially generated White Noise Bursts (WNBs) generated on the GPU via a simple custom Python [32] function utilising TensorFlow [34]. The procedure for generating WNBs with randomised duration and frequency content is as follows.

1. A maximum waveform duration is decided; typically, this would be less or equal to the duration of the example noise that you are injecting the waveform into, with some room for cropping.

2. Arrays of durations, minimum frequencies, and maximum frequencies are generated, each with a number of elements, N, equal to the number of waveforms that we wish to generate. These arrays can be pulled from any distribution as long as they follow the following rules. Duration cannot be larger than our maximum requested duration or less than zero. The frequency bounds cannot be less than zero or greater than the Nyquist frequency.
3. It is enforced that the maximum frequency is greater than the minimum frequency for any waveform by swapping values where this is not the case.
4. Gaussian white noise is generated with as many samples, which, given the selected sample rate, will produce a time series with the same duration as our requested max waveform duration.
5. A number of samples at the end of each waveform are zeroed so that each waveform has a number of samples equivalent to the randomised duration assigned to that signal.
6. Each waveform is transformed into the frequency domain by a RFFT.
7. Samples are zeroed at each end of each frequency-domain signal in order to perform a bandpass and limit the waveform between the assigned frequency constraints for each waveform.
8. The remaining signal is windowed using a Hann window to reduce the effects of the discontinuities generated by the bandpass operation.
9. The frequency domain signal is then returned to the time domain via a IRFFT.
10. Finally, the time-domain waveform is enveloped by a sigmoid window.
11. Assuming the plus polarisation component of the waveform strain was generated first, repeat with the same parameters but different initial noise distributions for the cross polarisation component.

Because we have used random noise across a range of frequency spaces, our distribution will, in theory, cover all possible signals within the specified parameter range. These WNBs can generate waveforms which look qualitatively similar to many proposed burst sources, including current supernovae simulations; see Figure 22. See Figure 21 for examples of our WNBs and Listing 9 for the code used to generate these waveforms.

```
@tf.function
def generate_white_noise_burst(
    num_waveforms: int,
    sample_rate_hertz: float,
    max_duration_seconds: float,
    duration_seconds: tf.Tensor,
    min_frequency_hertz: tf.Tensor,
    max_frequency_hertz: tf.Tensor
) -> tf.Tensor:

    # Casting
    min_frequency_hertz = tf.cast(min_frequency_hertz, tf.float32)
    max_frequency_hertz = tf.cast(max_frequency_hertz, tf.float32)

    # Convert duration to number of samples
    num_samples_array = tf.cast(sample_rate_hertz * duration_seconds, tf.int32)
    max_num_samples = tf.cast(max_duration_seconds * sample_rate_hertz, tf.int32)

    # Generate Gaussian noise
    gaussian_noise = tf.random.normal([num_waveforms, 2, max_num_samples])

    # Create time mask for valid duration
    mask = tf.sequence_mask(num_samples_array, max_num_samples, dtype=tf.float32)
    mask = tf.reverse(mask, axis=[-1])
    mask = tf.expand_dims(mask, axis=1)
```

```
    # Mask the noise
    white_noise_burst = gaussian_noise * mask

    # Window function
    window = tf.signal.hann_window(max_num_samples)
    windowed_noise = white_noise_burst * window

    # Fourier transform
    noise_freq_domain = tf.signal.rfft(windowed_noise)

    # Frequency index limits
    max_num_samples_f = tf.cast(max_num_samples, tf.float32)
    num_bins = max_num_samples_f // 2 + 1
    nyquist_freq = sample_rate_hertz / 2.0

    min_freq_idx = tf.cast(
        tf.round(min_frequency_hertz * num_bins / nyquist_freq), tf.int32)
    max_freq_idx = tf.cast(
        tf.round(max_frequency_hertz * num_bins / nyquist_freq), tf.int32)

    # Create frequency masks using vectorized operations
    total_freq_bins = max_num_samples // 2 + 1
    freq_indices = tf.range(total_freq_bins, dtype=tf.int32)
    freq_indices = tf.expand_dims(freq_indices, 0)
    min_freq_idx = tf.expand_dims(min_freq_idx, -1)
    max_freq_idx = tf.expand_dims(max_freq_idx, -1)
    lower_mask = freq_indices >= min_freq_idx
    upper_mask = freq_indices <= max_freq_idx
    combined_mask = tf.cast(lower_mask & upper_mask, dtype=tf.complex64)
    combined_mask = tf.expand_dims(combined_mask, axis=1)

    # Filter out undesired frequencies
    filtered_noise_freq = noise_freq_domain * combined_mask

    # Inverse Fourier transform
    filtered_noise = tf.signal.irfft(filtered_noise_freq)

    envelopes = generate_envelopes(num_samples_array, max_num_samples)
    envelopes = tf.expand_dims(envelopes, axis=1)

    filtered_noise = filtered_noise * envelopes

    return filtered_noise
```

Listing 9 | *Python [32]* . TensorFlow [34] graph function to generate the plus and cross polarisations of WNB waveforms; see Section 4.2.3 for a description of the generation method. `num_waveforms` takes an integer value of the number of WNBs we wish to generate. `sample_rate_hertz` defines the sample rate of the data we are working with. `max_duration_seconds` defines the maximum possible duration of any signals within our output data. `duration_seconds`, `min_frequency_hertz`, and `max_frequency_hertz` all accept arrays or in this case TensorFlow tensors, of values with a number of elements equal to `num_waveforms`, each duration. Both polarisations of the WNB are generated with parameters determined by the value of these three arrays at the equivalent index.

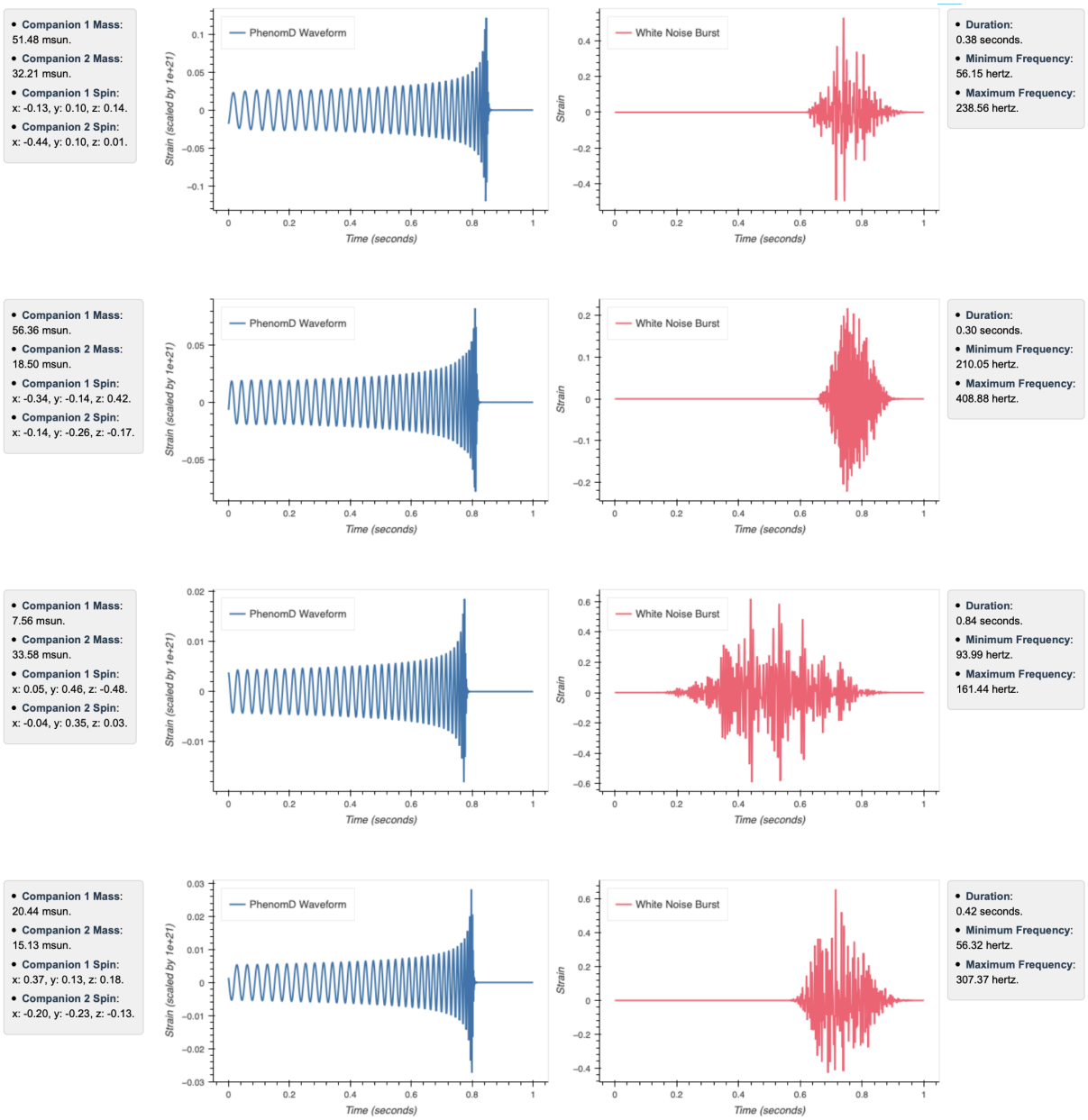


Figure 21 | Eight simulated waveforms that could be used for injection into noise to form an obfuscated training, testing, or validation example for an artificial neural network. Note that only the plus polarisation component of the strain, h_+, has been plotted in order to increase visual clarity. The leftmost four injections are IMRPhenomD waveforms generated using cuPhenom; see Section 8.1, with parameters (shown in the adjacent grey information boxes) drawn from uniform distributions between 5.0 $M_\odot$ and 95.0 $M_\odot$ for the mass of both companions and between −0.5 and 0.5 for the dimensionless spin component. Note that during injection generation, the two companions are always reordered so that the mass of companion one is greater and that the IMRPhenomD waveform ignores the x and y spin components. They are included just for code completion. The rightmost four injections consist of WNB waveforms generated via the method described in Section 4.2.3. Their parameters were again drawn from uniform distributions and are shown in the grey box to their right. The durations were limited between 0.1 s and 1.0 s, and the frequencies were limited to between 20.0 Hz and 500.0 Hz, with the minimum and maximum frequencies automatically swapped.

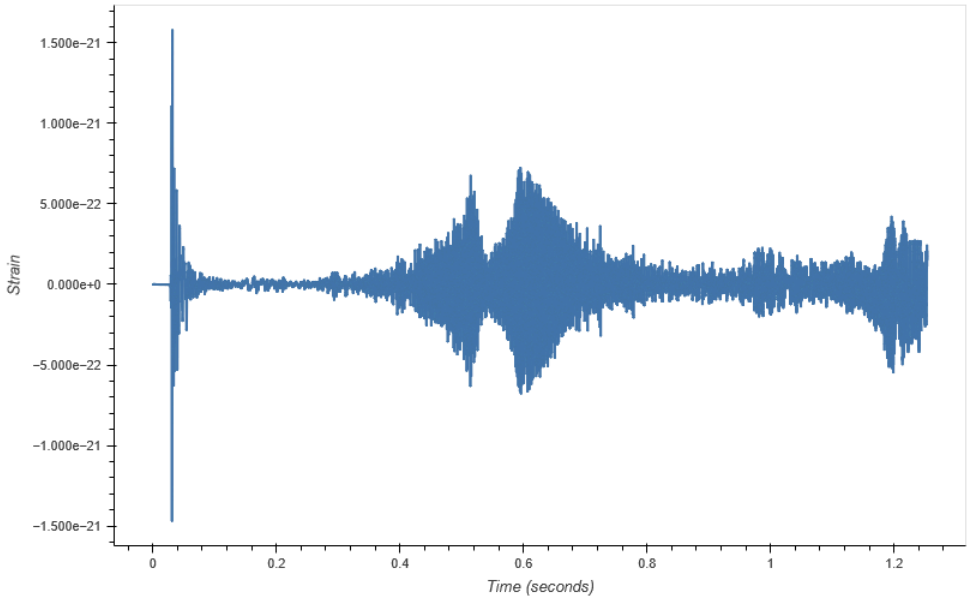


Figure 22 | The plus polarisation component of the gravitational-wave strain of a simulated core-collapse supernova at a distance of 10 kpc, this data was taken from [81]. Although some structures can clearly be observed, it is possible to imagine that a method trained to detect WNB signals, such as those presented in Figure 21, might be able to detect the presence of such a signal.

4.2.4 Waveform Projection

As has been discussed, gravitational waves have two polarisation states plus, $+$, and cross, $\times$, which each have their own associated strain values h_+ and $h_\times$. Since these strain polarisation states can have different morphologies and since the polarisation angle of an incoming signal paired with a given interferometer's response will alter the proportion of each polarisation that is perceptible by the detector, our aproximant signals are also generated with two polarisation components. Before being injected into any data, the waveforms must be projected onto each detector in our network in order to simulate what that signal would look like when observed with that detector. This projection will account for the full antenna response of each detector. Since a given interferometer has different sensitivities depending on both the direction of the source and the polarisation angle of the incoming wave, some waves will be entirely undetectable in a given detector.

If we want accurate data when simulating multi-interferometer examples, we must account for both the polarisation angle and direction of the source so that the relative strain amplitudes and morphologies in each detector are physically realistic.

Since the detectors have a spatial separation, there will usually, depending on source direction, also be a difference in the arrival time of the waves at the different detectors — this discrepancy is especially important for localising sources, as it provides the possibility for source triangulation, which, along with the antenna responses of each detector, can be used to generate a probability map displaying the probability that a wave originated from a given region of the sky. In coherence detection methods, it also allows for the exclusion of multi-interferometer detections if the detections arise with an arrival time difference greater than that which is physically possible based on the spatial separation of the detectors.

None of this is essential when dealing with single detector examples — in those cases, we could choose to forgo projection entirely and inject one of the strain polarisation components directly into the obfuscating noise as there are no time separations to model accurately and signal proportionality between detectors is also irrelevant.

The projection from both the antenna response parameters and the arrival time delay are dependent on the source direction. The plane of the wavefront and the direction of travel of the wave are dependent on the direction of the source. Since the sources are all extremely distant, the wavefront is considered a flat plane. Waves have some time duration, so both the time delay and antenna response parameters will change over the course of the incoming wave's duration as the Earth and the detectors move in space. As we are dealing with relatively short transients ($< 1.0\ s$), the change in these factors will be considered negligible and is not included in projection calculations.

Assuming that we ignore the Earth's motion, the final waveform present in a detector is given by

$$h(t) = F_{+}h_{+}(t + \Delta t) + F_{\times}h_{\times}(t + \Delta t) \tag{47}$$

where $h(t)$ is the resultant waveform present in the detector output at time t; F_{+} and $F_{\times}$ are the detector antenna response parameters in the plus and cross polarisations for a given source direction, polarisation angle, and detector; h_{+} and $h_{\times}$ are the plus and cross polarisations of the gravitational-wave strain of simulated or real gravitational waves; and Δt is the arrival time delay taken from a common reference point, often another detector or the Earth's centre.

We can also calculate the relative times that the signals will arrive at a given detector,

$$\Delta t = \frac{(\vec{x_0} - \overrightarrow{x_d})}{c} \cdot \vec{m} \tag{48}$$

where Δt is the time difference between the wave's arrival at location $\overrightarrow{x_d}$ and $\vec{x_0}$, c is the speed of light, $\vec{x_0}$ is some reference location, often taken as the Earth's centre, $\overrightarrow{x_d}$ is the location for which you are calculating the time delay, in our case, one of our interferometers, and $\vec{m}$ is the direction of the gravitational-wave source. If we work in Earth-centred coordinates and take the Earth's centre as the reference position so that $x_0 = [0.0, 0.0, 0.0]$ we can simplify Equation 48 to

$$\Delta t = -\frac{\vec{x}}{c} \cdot \vec{m}. \tag{49}$$

Finally, combining Equation 47 and Equation 49, we arrive at

$$h(t) = F_{+}h_{+}\left(t - \frac{\vec{x}}{c} \cdot \vec{m}\right) + F_{\times}h_{\times}\left(t - \frac{\vec{x}}{c} \cdot \vec{m}\right). \tag{50}$$

In practice, for our case of discretely sampled data, we first calculate the effect of the antenna response in each detector and then perform a heterodyne shift to each projection to account for the arrival time differences. When multiple detector outputs are required for training, testing, or validation examples, we will perform these calculations using a GPU-converted version of the PyCBC [82] project_wave function; see Figure 23 for example projections.

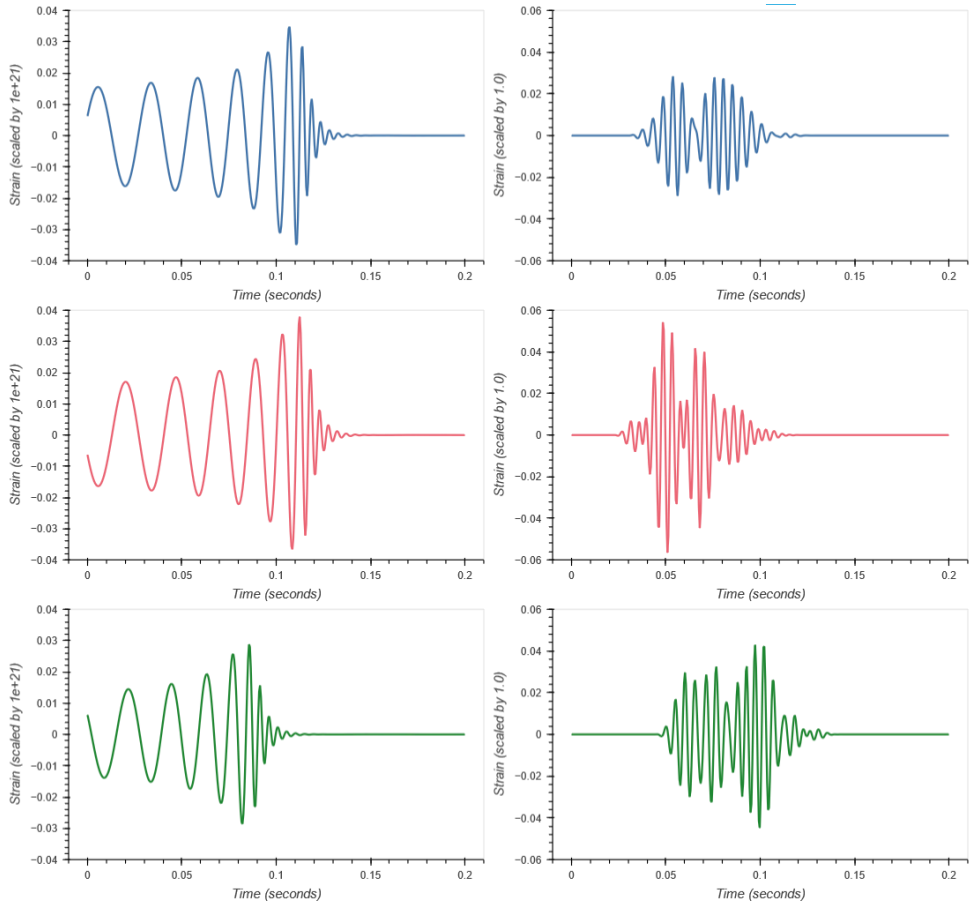


Figure 23 | Example projection of two artificial gravitational-wave waveforms. The blue waveforms have been projected into the LIGO Livingston interferometer, the red waveforms have been projected into the Ligo Hanford interferometer, and the green waveforms have been projected into the VIRGO interferometer. The left column displays different projections of an IMRPhenomD waveform generated with the cuPhenom GPU library; see Section 8.1. The right column displays different projections of a WNB waveform generated with the method described in Section 4.2.3. The projections were performed using a GPU adaptation of the PyCBC Python library's [82] project_wave function. Both waveforms were projected from different source locations; the projection and time displacement were different in each case.

4.2.5 Waveform Scaling

Once waveforms have been projected to the correct proportionality, we must have some method to inject them into obfuscating noise with a useful scaling. If using physically scaled approximants, such as the IMRPhenomD waveform, we could forgo scale by calculating the resultant waveform that would be generated by a CBC at a specified distance from Earth, then injecting this into correctly scaled noise (or simply raw real noise). However, since we are also using non-physical waveforms such as WNBs, and because we would like a more convenient method of adjusting the detectability of our waveforms, we will use a method to scale the waveforms to a desired proportionality with the noise.

Evidently, if we injected waveforms that have been scaled to values near unity into real unscaled interferometer noise (which is typically on the order of 10^{-21}), even a very simple model would not have much of a problem identifying the presence of a feature. Equally, if the reverse were true, no model could see any difference between interferometer data with or without an injection. Thus, we

must acquire a method to scale our injections so that their amplitudes have a proportionality with the background noise that is similar to what might be expected from real interferometer data.

Real data holds a distribution of feature amplitudes, with quieter events appearing in the noise more commonly than louder ones — this is because gravitational-wave amplitude scales inversely with distance, whereas the volume of searchable space, and thus matter and, generally, the number of systems which can produce gravitational waves, scale cubically with distance from Earth.

Features with quieter amplitudes will, in general, be harder for a given detection method to identify than features with louder amplitudes. We must design a training dataset that contains a curriculum which maximises model efficacy across our desired regime, with examples that are difficult but never impossible to classify and perhaps some easier cases that can carve channels through the model parameters, which can be used to direct the training of more difficult examples.

In any given noise distribution, there will, for any desired false alarm rate, be a minimum detectable amplitude below which it becomes statistically impossible to make any meaningful detections. This minimum amplitude occurs because even white Gaussian noise will occasionally produce data which looks indistinguishable from a certain amplitude of waveform.

We can use matched filtering statistics to prove this point, as we know that given an exactly known waveform morphology and perfect Gaussian noise, matched filtering is the optimal detection statistic. The probability that a matched filtering search of one template produces a false alarm is dependant only on the rate at which you are willing to miss true positives. We can use the $\mathcal{F}$-statistic, $\mathcal{F}_0$, for our probability metric to adjust this rate. Assuming that the noise is purely Gaussian, and we are only searching for one specific template, the probability of false detections of this exact waveform, i.e. $P(\mathcal{F} > \mathcal{F}_0)$, can be expressed as

$$P_F(\mathcal{F}_0) = \int_{\mathcal{F}_0}^{\infty} p_0(\mathcal{F}) d\mathcal{F} = \exp(-\mathcal{F}_0) \sum_{k=0}^{\frac{n}{2}-1} \frac{\mathcal{F}_0^k}{k!} \tag{51}$$

where n is the number of degrees of freedom of χ^2 distributions. We can see from Equation 51 that the False Alarm Rate (FAR) in this simple matched filtering search is only dependent on your arbitrary choice of $\mathcal{F}_0$. However, in practice, your choice $\mathcal{F}_0$ will be determined by the minimum amplitude waveform you wish to detect because the probability of detection, P_d given the presence of a waveform, is dependent on the optimal SNR, ρ_{opt} of that waveform, ρ, which has a loose relationship to the amplitude of the waveform. The probability of detection is given by

$$P_D(\rho, \mathcal{F}_0) = \int_{\mathcal{F}_0}^{\infty} \frac{(2\mathcal{F})^{\frac{\frac{n}{2}-1}{2}}}{\rho^{\frac{n}{2}-1}} I_{\frac{n}{2}-1}\left(\rho\sqrt{2\mathcal{F}}\right) \exp\left(-\mathcal{F} - \frac{1}{2}\rho^2\right) d\mathcal{F} \tag{52}$$

where $I_{\frac{n}{2}-1}$ is the modified Bessel function of the first kind and order $\frac{n}{2} - 1$. For more information on this, please refer to [83].

More complex types of noise, however, like real LIGO interferometer noise, could potentially produce waveform simulacra more often than artificially generated white noise.

Louder false alarms are less likely than quieter ones, and at a certain amplitude, your detection method will start producing a greater number of false alarms than your desired false alarm rate. If our training dataset includes waveforms with an amplitude that would trigger detections with a false alarm rate near or less than our desired rate, this could significantly reduce the performance of our network, so we must select a minimum amplitude that maximises our detection efficiency at a given false alarm rate.

Our minimum possible detection amplitude is limited by the combination of the noise and the false alarm rate we desire. There is not a maximum possible signal amplitude, other than some very unuseful upper bound on the closest possible gravitational-wave-producing systems to Earth (a nearby supernova or CBC, for example), but these kinds of upper limit events are so astronomically rare as not to be worth considering. Events will, however, follow a distribution of amplitudes. As is often the case, we can try to generate our training data using a distribution that is as close as possible to the observed data, with the exception of a lower amplitude cutoff, or we can instead use a non-realistic distribution, uniformly or perhaps Gaussianly distributed across some amplitude regime which contains the majority of real signals — making the assumption that any detection methods we train using this dataset will generalise to higher amplitudes, or failing that, that the missed signals will be so loud that they would not benefit greatly from improved detection methods.

Thus far in this subsection, we have been talking rather nebulously about waveform "amplitude", as if that is an easy thing to define in a signal composed of many continious frequency components. There are at least three properties we might desire from this metric. Firstly, magnitude, some measure of the raw energy contained within the gravitational wave as it passes through Earth — this measure contains a lot of physical information about the gravitational wave source. Secondly, significance, given the circumstances surrounding the signal, we may want to measure how likely the signal is to have been astrophysical, and finally, closely related to the significance and perhaps most importantly when designing a dataset for artificial neural network training, the detectability, given a chosen detection method this would act as a measure of how easy it is for that method to detect the signal.

Naively, one might assume that simply using the maximum amplitude of the strain, h_{peak}, would be a good measure, and indeed, this would act as a very approximate measure of the ease of detection — but it is not a complete one. Consider, for a moment, a sine-Gaussian with an extremely short duration on the order of tens of milliseconds but a maximum amplitude that is only slightly louder than a multi-second long BNS signal. You can imagine from this example that the BNS would be considerably easier to detect, but if you were going by h_{peak} alone, then you would have no idea.

Within gravitational-wave data science, there are nominally two methods for measuring the detectability of a signal — the root-sum-squared strain amplitude, h_{rss} and the optimal Signal Noise Ratio, ρ_{opt}. What follows is a brief description of these metrics.

4.2.5.1 The Root-Sum-Squared strain amplitude, h_{rss}

The Root-Sum-Squared strain amplitude, h_{rss}:, is a fairly simple measure of detectability. Unlike ρ_{opt}, it is exclusive to gravitational-wave science. It accounts for the power contained across the whole signal by integrating the square of the strain across its duration, essentially finding the area contained by the waveform. It is given by

$$h_{\text{rss}} = \sqrt{\int \left(h_+(t)^2 + h_\times(t)^2\right) dt} \tag{53}$$

or written in its discrete form, which is more relevant for digital data analysis

$$h_{\text{rss}} = \sqrt{\sum_{i=1}^{N} \left(h_+[t_i]^2 + h_\times[t_i]^2\right)} \tag{54}$$

when h_{rss} is the root-sum-squared strain amplitude, $h_+(t)$ and $h_\times(t)$ are the plus and cross polarisations of the continuous strain, $h_+(t_i)$ and $h_\times(t_i)$ are the plus and cross polarisations of the discrete strain at the i$^{\text{th}}$ data sample, and N is the number of samples in your waveform.

It should be noted that with any measure that utilises the strain, such as h_{peak} and h_{rss}, there is some ambiguity concerning where exactly to measure strain. You could, for example, measure the raw strains h_+ and $h_\times$ before they have been transformed by the appropriate detector antenna response functions, or you could take the strain h after it has been projected onto a given detector. The advantage of the former is that you can fairly compare the magnitude of different gravitational waves independent of information about the interferometer in which it was detected. This is the commonly accepted definition of the h_{rss}.

The h_{rss} is most often used during burst analysis as a measure of the detectability, magnitude, and significance of burst transients. Within CBC detection, ρ_{opt} is often preferred. Whilst h_{rss} is a simple and convenient measure, it ignores noise, so it cannot by itself tell us if a signal is detectable.

4.2.5.2 Optimal Signal-to-Noise Ratio (SNR) (ρ_{opt})

The optimal Signal-to-Noise Ratio (SNR), ρ_{opt}, solves both of these issues by acting as a measure of detectability, magnitude, and significance in comparison to the background noise. Consequently, because it is relative to the noise, the magnitude of a given waveform can only be compared to the optimal SNR of a waveform that was obfuscated by a similar noise distribution. If a real gravitational-wave signal were detected in a single LIGO detector, say, LIGO Hanford, for example, then its optimal SNR would be significantly larger than the same signal detected only in VIRGO, even if the signal was aligned in each case to the original from the optimally detectable sky location. This is because the sensitivity of the VIRGO detector is substantially lower than the two LIGO detectors, so the noise is proportionally louder compared to the waveforms.

It is, however, possibly a good measure of detectability, as detection methods do not much care about the actual magnitude of the signal when they are attempting to analyse one; the only relevant factors, in that case, are the raw data output, consisting of the portion of the gravitational-wave strain perceptible given the detector's antenna response function, see Equation 50, and the interferometer noise at that time.

The SNR can also sometimes be an ambiguous measurement, as there are multiple different metrics which are sometimes referred to by this name, most prominently, a ratio between the expected value of the signal and the expected value of the noise, or sometimes the ratio between the root mean square of the signal and noise. Within gravitational-wave data science, though there is sometimes confusion over the matter, the commonly used definition for SNR is the matched filter SNR, ρ_{opt}. Since matched filtering is the optimal method for detecting a known signal in stationary Gaussian noise, we can use the result of a matched filter of our known signal with that signal plus noise as a measure of the detectability of the signal in a given noise distribution.

The optimal SNR, ρ_{opt}, is given by

$$\rho_{\text{opt}} = \sqrt{4 \int_0^\infty \frac{|\tilde{h}(f)|^2}{S(f)} df} \tag{55}$$

where ρ_{opt} is the optimal SNR, S(f) is the one sided PSD, and

$$\tilde{h}(f) = \int_{-\infty}^{\infty} h(x) e^{-i2\pi ft} dt \tag{56}$$

is the Fourier transform of h(f). The coefficient of 4 is applied since, in order to use only the one-sided transform, we assume that $S(f) = S(-f)$, which is valid because the input time series is entirely real. This applies a factor of two to the output, and since we are only integrating between 0 and ∞ rather than $-\infty$ to ∞, we apply a further factor of 2.

Because, again, for data analysis purposes, the discrete calculation is more useful, the ρ_{opt} of discrete data is given by

$$\rho_{\text{opt}} = \sqrt{4 \sum_{k=1}^{N-1} \frac{|\tilde{h}[f_k]|^2}{S(f_k)}} \tag{57}$$

where N is the number of samples, and, in this case, the discrete Fourier transform $\tilde{h}[f]$ is given by

$$\tilde{h}[f_k] = \sum_{i=1}^{N-1} h[t_i] e^{-\frac{2\pi}{N} ki} \tag{58}$$

For the work during this thesis, we have used a TensorFlow [34] implementation for calculating the ρ_{opt} of a measure. This implementation is show in Listing 10.

```
@tf.function
def calculate_snr(
    injection: tf.Tensor,
    background: tf.Tensor,
    sample_rate_hertz: float,
    fft_duration_seconds: float = 4.0,
    overlap_duration_seconds: float = 2.0,
    lower_frequency_cutoff: float = 20.0,
    ) -> tf.Tensor:

    injection_num_samples      = injection.shape[-1]
    injection_duration_seconds = injection_num_samples / sample_rate_hertz

    # Check if input is 1D or 2D
    is_1d = len(injection.shape) == 1
    if is_1d:
        # If 1D, add an extra dimension
        injection = tf.expand_dims(injection, axis=0)
        background = tf.expand_dims(background, axis=0)

    overlap_num_samples = int(sample_rate_hertz*overlap_duration_seconds)
    fft_num_samples     = int(sample_rate_hertz*fft_duration_seconds)

    # Set the frequency integration limits
    upper_frequency_cutoff = int(sample_rate_hertz / 2.0)

    # Calculate and normalize the Fourier transform of the signal
    inj_fft = tf.signal.rfft(injection) / sample_rate_hertz
    df = 1.0 / injection_duration_seconds
    fsamples = \
        tf.range(0, (injection_num_samples // 2 + 1), dtype=tf.float32) * df

    # Get rid of DC
    inj_fft_no_dc  = inj_fft[:,1:]
    fsamples_no_dc = fsamples[1:]

    # Calculate PSD of the background noise
    freqs, psd = \
        calculate_psd(
            background,
```

```
            sample_rate_hertz = sample_rate_hertz,
            nperseg           = fft_num_samples,
            noverlap          = overlap_num_samples,
            mode="mean"
        )

    # Interpolate ASD to match the length of the original signal
    freqs = tf.cast(freqs, tf.float32)
    psd_interp = \
        tfp.math.interp_regular_1d_grid(
            fsamples_no_dc, freqs[0], freqs[-1], psd, axis=-1
        )

    # Compute the frequency window for SNR calculation
    start_freq_num_samples = \
        find_closest(fsamples_no_dc, lower_frequency_cutoff)
    end_freq_num_samples = \
        find_closest(fsamples_no_dc, upper_frequency_cutoff)

    # Compute the SNR numerator in the frequency window
    inj_fft_squared = tf.abs(inj_fft_no_dc*tf.math.conj(inj_fft_no_dc))

    snr_numerator = \
        inj_fft_squared[:,start_freq_num_samples:end_freq_num_samples]

    if len(injection.shape) == 2:
        # Use the interpolated ASD in the frequency window for SNR calculation
       snr_denominator = psd_interp[:,start_freq_num_samples:end_freq_num_samples]
    elif len(injection.shape) == 3:
                                      snr_denominator     =     psd_interp[:,     :,
start_freq_num_samples:end_freq_num_samples]

    # Calculate the SNR
    SNR = tf.math.sqrt(
        (4.0 / injection_duration_seconds)
        * tf.reduce_sum(snr_numerator / snr_denominator, axis = -1)
    )

    SNR = tf.where(tf.math.is_inf(SNR), 0.0, SNR)

    # If input was 1D, return 1D
    if is_1d:
        SNR = SNR[0]

    return SNR
```

Listing 10 | *Python [32].* TensorFlow [34] graph function to calculate the optimal SNR, ρ_{opt}, of a signal. `injection` is the input signal as a TensorFlow tensor, `background` is the noise into which the waveform is being injected, `sample_rate_hertz` is the sample rate of both the signal and the background, `fft_duration_seconds` is the duration of the FFT window used in the PSD calculation, `overlap_duration_seconds` is the duration of the overlap of the FFT window in the PSD calculation, and `lower_frequency_cutoff` is the frequency of the lowpass filter, below which the frequency elements are silenced.

Once the optimal SNR or h_{rss} of an injection has been calculated, it is trivial to scale that injection to any desired optimal SNR or h_{rss} value. Since both metrics scale linearly when the same coefficient scales each sample in the injection,

$$h_{\text{scaled}} = h_{\text{unscaled}} \frac{M_{\text{desired}}}{M_{\text{current}}} \tag{59}$$

where h_{scaled} is the injection strain after scaling, $h_{\text{unscaling}}$ is the injection strain before scaling, M_{desired} is the desired metric value, e.g. h_{rss}, or ρ_{opt}, and M_{current} is the current metric value, again either h_{rss}, or ρ_{opt}. Note that since h_{rss} and ρ_{opt} are calculated using different representations of the strain, h_{rss} before projection into a detector, and ρ_{opt} after, the order of operations will be different depending on the scaling metric of choice, ie. for h_{rss}: scale → project, and for ρ_{opt}: project → scale.

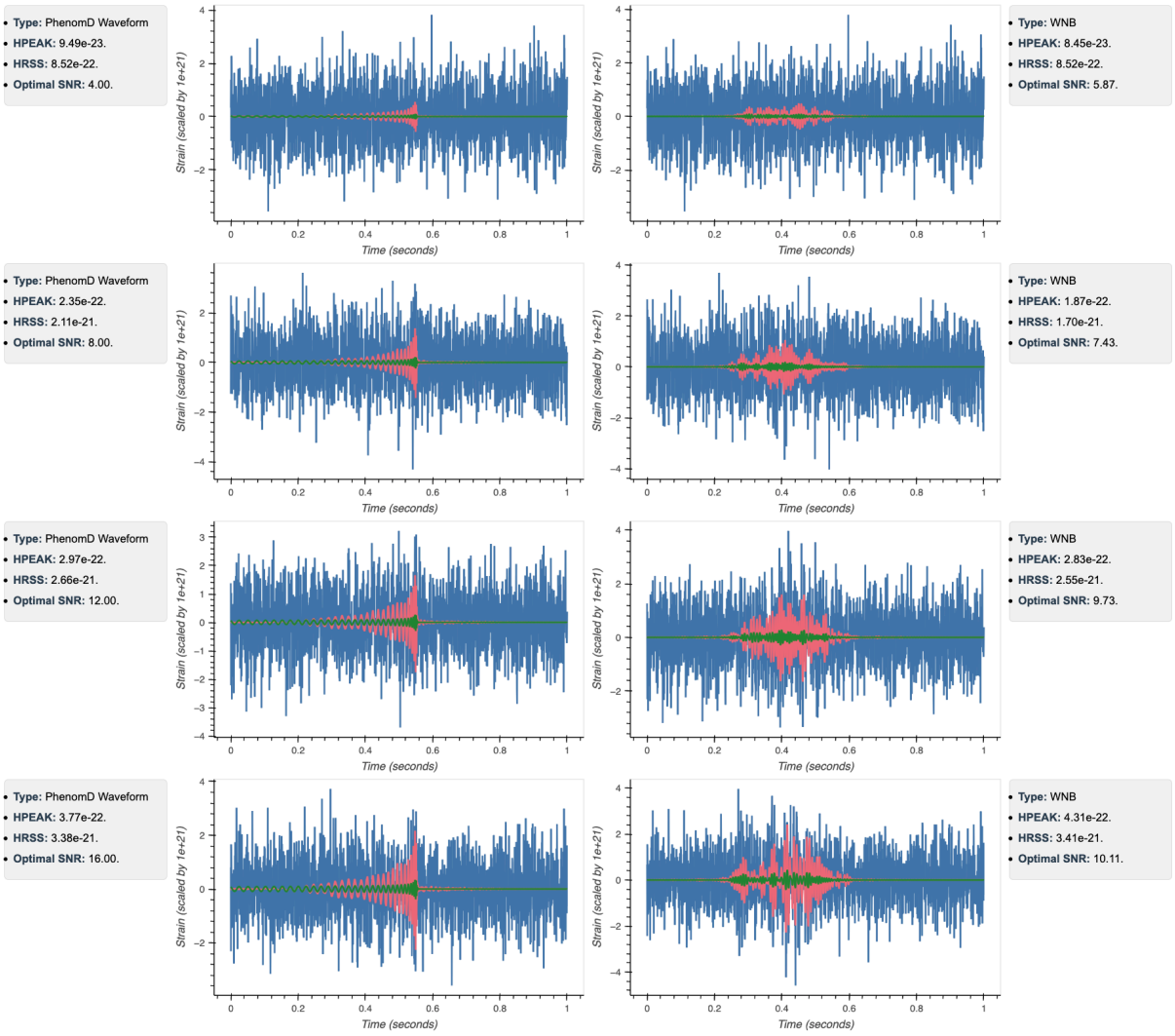


Figure 24 | Eight examples of artificial injections scaled to a particular scaling metric and added to a real noise background to show variance between different scaling methods. The blue line demonstrates the whitened background noise plus injection; the red line represents the injection after being run through the same whitening transform as the noise plus injection, and the green line represents the injection after scaling to the desired metric. The leftmost column contains an IMRPhenomD waveform, generated using cuPhenom, injected into a selection of various background noise segments and scaled using SNR; see Section 4.2.5.2. From upper to lower, the SNR values are 4, 8, 12, and 16, respectively. The rightmost column displays a WNB injected into various noise distributions, this time scaled using

h_{rss}; see Section 4.2.5.1. From upper to lower, the h_{rss} values are as follows: 8.52×10^{-22}, 1.70×10^{-21}, 2.55×10^{-21}, and 3.41×10^{-21}. As can be seen, though both sequences are increasing in linear steps with a uniform spacing of their respective metrics, they do not keep in step with each other, meaning that if we double the optimal SNR of a signal, the h_{rss} does not necessarily also double.

For the experiments performed later in this section, we will use SNR as our scaling metric drawn from a uniform distribution with a lower cutoff of 8 and an upper cutoff of 20. These values are rough estimates of a desirable distribution given the SNR values of previous CBC detections.

If we wish to utilise multiple detectors simultaneously as our model input, we can scale the injections using either the network SNR or the h_{rss} before projection into the detectors. In the case of h_{rss}, the scaling method is identical, performed before detection and injection. Network SNR is computed by summing individual detector SNRs in quadrature, as shown by

$$\rho_{\text{network}} = \sqrt{\sum_{i=1}^{N} \rho_i^2} \tag{60}$$

where ρ_{network} is the network SNR, N is the total number of detectors included in the input, and ρ_i is the detector SNR of the i$^{\text{th}}$ detector given in each case by Equation 57. To scale to the network, SNR Equation 59 can still be used, with the network SNR of Equation 60 as the scaling metric, by multiplying the resultant projected injection in each detector by the scaling coefficient.

4.2.6 Data Dimensionality and Layout

Interferometer output data is reasonably different from the example MNIST data we have been using to train models thus far, the primary difference being that it is one-dimensional rather than two, being more similar to audio than image data. In fact, most of the features we are looking for within the data have a frequency that, when converted to sound, would be audible to the human ear, so it is often useful to think of the problem in terms of audio classification. In many ways, this reduced dimensionality is a simplification of the image case. In pure dense networks, for example, we no longer have to flatten the data before feeding it into the model.

There are, however, multiple interferometers across the world. During an observing run, at any given time, there are anywhere between zero to five operational detectors online: LIGO Livingston (L1), LIGO Hanford (H1), Virgo (V1), Kagra (K1), and GEO600 (G1). GEO600 is not considered sensitive enough to detect any signals other than ones that would have to be so local as to be rare enough to dismiss the probability, so it is usually not considered for such analysis. It should also be noted that during O4, both Virgo and Kagra are currently operating with a sensitivity and up-time frequency that makes it unlikely they will be of assistance for any detection. It is hoped that the situation at these detectors will improve for future observing runs. As such, it is possible to include multiple detectors within our model input, and in fact, such a thing is necessary for coherence detection to be possible.

This multiplicity brings some complications in the construction of the input examples. Currently, we have only seen models that ignore the input dimensionality; however, with other network architectures, such as Convolutional Neural Networks (CNNs), this is not always the case. Therefore, we must consider the data layout. In the simplest cases, where we are not modifying the shape of the data before injection, we can imagine three ways to arrange the arrays; see Figure 25 for a visual representation.

- **Lengthwise**: wherein the multiple detectors are concatenated end to end, increasing the length of the input array by a factor equal to the number of detectors. This would evidently still be a 1D problem, just an extended one. While perhaps this is the simplest treatment, we can imagine that this

might perhaps be the hardest to interpret by the model, as we are mostly discarding the dimensionality, although no information is technically lost.

- **Depthwise**: Here, the detectors are stacked in the depth dimension, an extra dimension that is not counted toward the dimensionality of the problem, as it is a required axis for the implementation of CNNs, in which each slice represents a different feature map; see Section 5.2.2. Often, this is how colour images are injected by CNNs, with the red, green, and blue channels each taking up a feature map. This would seem an appropriate arrangement for the detectors. However, there is one significant difference between the case of the three-colour image and the stacked detectors, that being the difference in signal arrival time between detectors; this means that the signal will be offset in each channel. It is not intuitively clear how this will affect model performance, so this will have to be empirically compared to the other two layouts.
- **Heightwise**: The last possible data layout that could be envisioned is to increase the problem from a 1D problem to a 2D one. By concatenating the arrays along their height dimension, the 1D array can be increased to a 2D array.

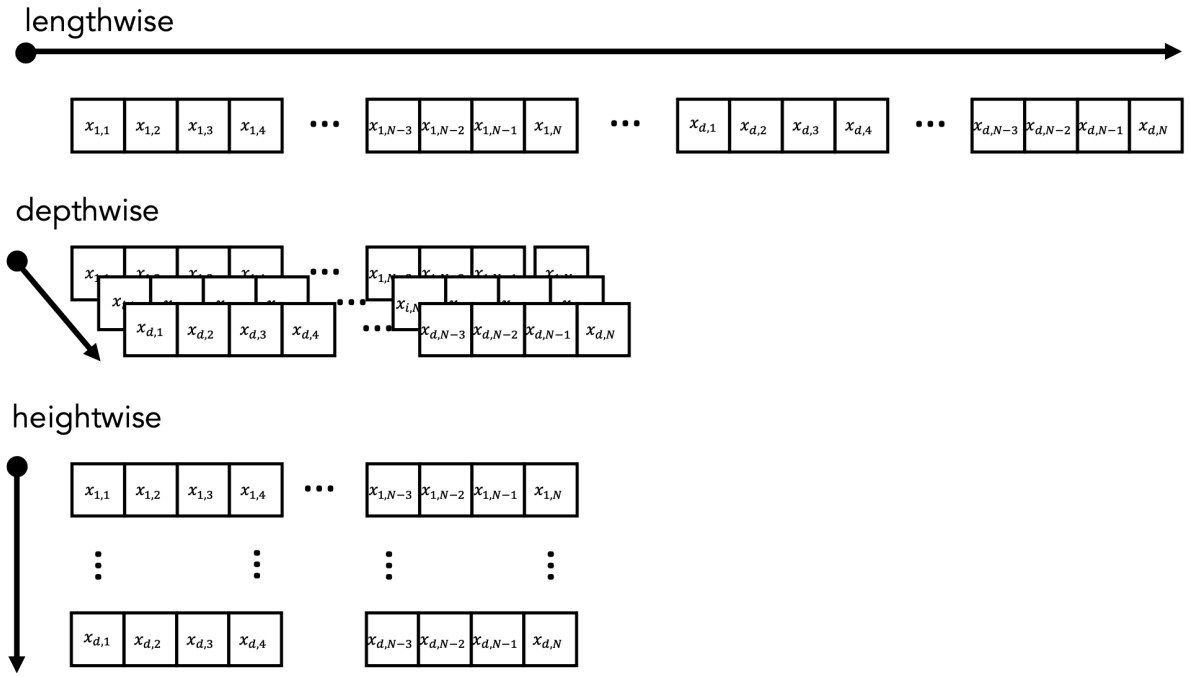


Figure 25 | Possible data layouts for multi-detector examples. Here, d is the number of included detectors, and N is the number of input elements per time series. There are three possible ways to align interferometer time-series data from multiple detectors. These layouts are discussed in more detail in Section 4.2.6.

For pattern-matching methods, like that which is possible in the CBC case, there are also advantages to treating each detector independently. If we do this, we can use the results from each model as independent statistics, which can then be combined to create a result with a far superior False Alarm Rate (FAR). We could combine the score from both models and calculate a false alarm rate empirically using this combined score, or use each detector as a boolean output indicating the presence of a detector or not, and combine the FARs using Equation 62.

For the first case treating the two models as one, the combined score is calculated by

$$S_{\text{comb}} = \prod_{i=1}^{N} S_i \tag{61}$$

where S_{comb} is the combined classification score, which can be treated approximately as a probability if the output layer uses a softmax, or single sigmoid, activation function, see Section 3.1.5.4, S_i is the output score of the i^{th} classifier input with the data from the i^{th} detector, and N is the number of included detectors. Note that one could employ a uniquely trained and/or designed model for each detector or use the same model for each detector.

In the second case, treating each model as an independent boolean statistic and assuming that the output of the detectors is entirely independent except for any potential signal, the equation for combining FARs is

$$\text{FAR}_{\text{comb}} = (w - o) \prod_{i=1}^{N} \text{FAR}_i \tag{62}$$

where FAR_{comb} is the combined FAR, N is the number of included detectors, w is the duration of the input vector, and o is the overlap between windows. This equation works in the case when your detection method tells you a feature has been detected within a certain time window, w, but not the specific time during that window, meaning that $t_{\text{central}} > w_{\text{start}} \wedge t_{\text{central}} < w_{\text{end}}$, where t_{central} is the signal central time, w_{start} is the input vector start time and w_{end} is the input vector end time.

If your detection method can be used to ascertain a more constrained time for your feature ($w_{\text{duration}} <$ light_travel_time), then you can use the light travel time between the two detectors to calculate a FAR. For two detectors, combing the FAR in this way can be achieved by

$$\text{FAR}_{1,2} = 2\,\text{FAR}_1\,\text{FAR}_2\,w_{1,2} \tag{63}$$

where FAR_{comb} is the combined FAR, and $w_{1,2}$ is the light travel time between detectors 1 and 2, as this is the largest physically possible signal arrival time separation between detectors; gravitational waves travel at the speed of light, and detector arrival time difference is maximised if the direction of travel of the wave is parallel to the straight-line path between the two detectors.

In the case where we are using t_{central} and coincidence times to calculate our combined FAR, if we use overlapping data segments to feed our model, we must first group detections that appear in multiple inferences and find one central time for the detection. We can use an empirical method to determine how best to perform this grouping and identify if and how model sensitivity varies across the input window.

4.2.7 Feature Engineering and Data Conditioning

Invariably, there are data transforms that could be performed prior to ingestion by the model. If there are operations that we imagine might make the task at hand easier for the model, we can perform these transforms to improve network performance. Because we are attempting to present the data to the model in a form that makes the features easier to extract, this method of prior data conditioning is known as **feature engineering**. It should be noted that feature engineering does not necessarily add any extra information to the data. In fact, in many cases, it can reduce the overall information content whilst simultaneously simplifying the function that the model is required to approximate in order to operate as intended, see for example the whitening procedure described in Section 4.2.7.2. As we have said before, although the dense neural network is, at its limit, a universal function approximator, there are practical limitations to finding the right architecture and parameters for a given function, so sometimes simplifying the task can be beneficial. This can reduce the model size and training time, as well as

improve achievable model performance when the time available for model and training optimisation is limited.

4.2.7.1 Raw Data

When designing the package of information that will be presented to the network at each inference, the simplest approach would be to feed the raw interferometer data directly into the model. There are certainly some methodologies that consider it optimal to present your model with as much unaltered information as possible. By performing little to no data conditioning, you are allowing the network to find the optimal path to its solution; if all the information is present and an adequate model architecture is instantiated, then a model should be able to approximate the majority of possible conditioning transforms during model training, not only this, but it may be able to find more optimal solutions that you have not thought of, perhaps ones customised to the specific problem at hand, rather than the more general solutions that a human architect is likely to employ. This methodology, however, assumes that you can find this adequate model architecture and have an adequate training procedure and dataset to reach the same endpoint that could be achieved by conditioning your data. This could be a more difficult task than achieving a result that is almost as good with the use of feature engineering.

4.2.7.2 Whitened Data

One type of data conditioning that we will employ is time-series whitening. As we have seen in Section 2.3.1, as well as containing transient glitches, the interferometer background is composed of many different continuous quasi-stationary sources of noise, the frequency distributions of which compose a background that is unevenly distributed across our frequency search space. This leaves us with 1D time series that have noise frequency components with much greater power than any interesting features hidden within the data. This could potentially make detections using most methods, including artificial neural networks, much more difficult, especially when working in the time domain; see Figure 26 for an example of the PSD of unwhitened noise.

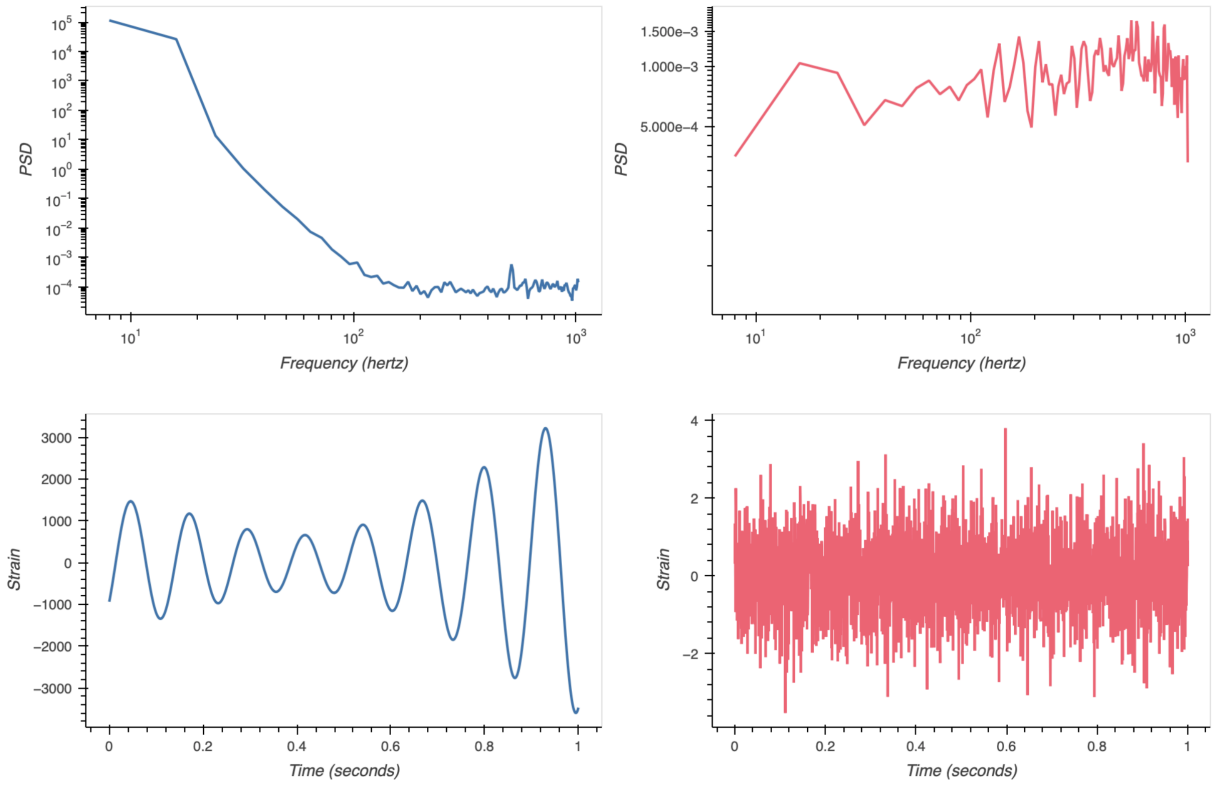


Figure 26 | An example of a segment of interferometer data before and after whitening. The two leftmost plots in blue show the PSD, *upper*, and raw data, *lower*, output from the LIGO Hanford detector

before any whitening procedure was performed. The two rightmost plots show the same data after the whitening procedure described in Section 4.2.7.2 has been implemented. The data was whitened using the ASD of a 16.0 s off-source window from 16.5 s before the start of the on-source window to 0.5 s before. The 0.5 s gap is introduced as some data must be cropped after whitening due to edge effects caused by windowing. This also acts to ensure that it is less likely that any features in the on-source data contaminate the off-source data, which helps reduce the chance that we inadvertently whiten any interesting features out of the data.

Fortunately, there exists a method to flatten the noise spectrum of a given time series whilst minimising the loss of any transient features that don't exist in the noise spectrum. This requires an estimate of the noise spectrum of the time series in question, which does not contain the hidden feature. In this case, this noise spectrum will take the form of an ASD; see Equation 46.

Since the noise spectrum of the interferometer varies with time, a period of noise close to but not overlapping with the section of detector data selected for analysis must be chosen — we call this time series the **off-source** period. The period being analysed, the **on-source** period, is not included in the off-source period so that any potential hidden features that are being searched for, e.g. a CBC signal, do not contribute significant frequency components to the ASD, which may otherwise end up dampening the signal along with the noise during the whitening procedure. It should be noted, then, that whitening via this process uses additional information from the off-source period that is not present in the on-source data. During this thesis, we have elected to use an off-source window duration of 16.0 s, as this was found to be an optimal duration by experiments performed as part of previous work during the development of MLy [51], although it should be noted that we have taken the on-source and crop regions after the off-source as opposed to the initial MLy experiments wherein it was taken at the centre of the off-source window. See Figure 27 for a depiction of the relative locations of the on-source and off-source segments.

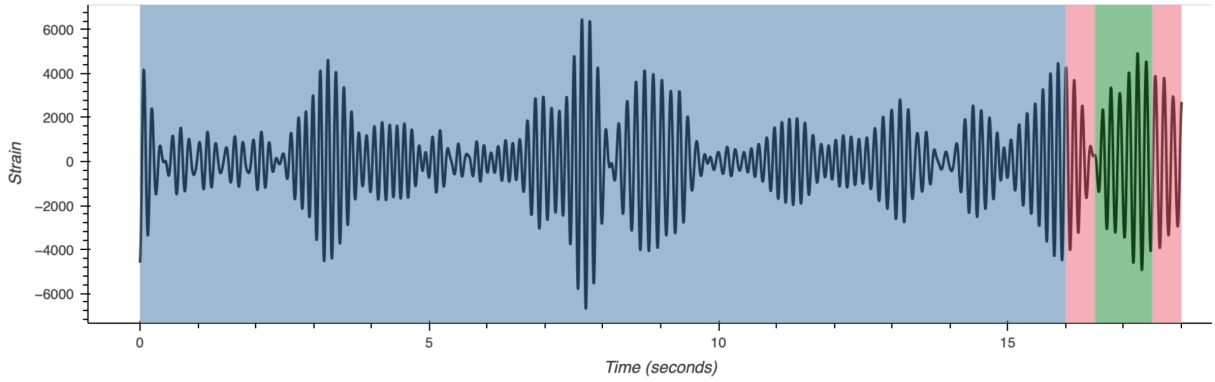


Figure 27 | Demostration of the on-source and off-source regions used to calculate the ASD used during the whitening operations throughout this thesis wherever real noise is utilised. Where artificial noise is used, the off-source and on-source segments are generated independently but with durations equivalent to what is displayed above. The blue region shows the 16.0 s off-source period, the green region shows the 1.0 s on-source period, and the two red regions represent the 0.5 s crop periods, which are removed after whitening. During an online search, the on-source region would advance in second-long steps, or if some overlap was implemented, less than second-long steps, meaning all data would eventually be searched. The leading 0.5 s crop region will introduce an extra 0.5 s of latency to any search pipeline. It may be possible to avoid this latency with alternate whitening methods, but that has not been discussed here.

We can whiten the data by convolving it with a suitably designed Finite Impulse Response (FIR) filter. This procedure is described by the following steps:

1. Calculate the ASD using Equation 46, this will act as the transfer function, $G(f)$, for generating the FIR filter. This transfer function is a measure of the frequency response of the noise in our system, and during the whitening process, we will essentially try to normalise the on-source by this off-source noise in order to flatten its PSD. We generate a filter with a 1 s duration.
2. Next, we zero out the low and high-frequency edges of the transfer function with

$$G_{\text{trunc}}(f) = \begin{cases} 0 \text{ if } f \leq f_{\text{corner}} \\ G(f) \text{ if } f_{\text{corner}} < f < f_{\text{Nyquist}} - f_{\text{corner}}. \\ 0 \text{ if} \geq f_{\text{Nyquist}} - f_{\text{corner}} \end{cases} \tag{64}$$

This stage discards frequency components which we no longer care about both because these frequencies are outside of the band we are most interested in and because discarding them can improve function stability and performance whilst reducing artifacting.

3. Optionally, we can apply a Planc-taper window to smooth the discontinuities generated by step 2; we will apply this window in all cases. The Planc-taper window has a flat centre with smoothly tapering edges, thus the windowing is only applied as such to remove discontinuities whilst affecting the central region as little as possible.

$$G_{\text{smoothed}}(f) = G_{\text{trunc}}(f) \cdot W(f). \tag{65}$$

4. Next we compute the inverse Fourier transform of $G_{\text{smoothed}}(f)$ to get the FIR filter, $g(t)$, with

$$g(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} G_{\text{smoothed}}(f) e^{jft} df. \tag{66}$$

This creates a time-domain representation of our noise characteristics, which can then be used as a filter to remove similar noise from another time-domain signal. In practice, we utilise an RFFT function to perform this operation on discrete data. As opposed to an FFT, this transform utilises symmetries inherent when transforming from complex to real data in order to halve the computational and memory requirements.

5. Finally, we convolve our FIR filter, $g(t)$, with the data we wish to whiten, $x(t)$,

$$x_{\text{whitened}}(t) = x(t) * g(t) \tag{67}$$

where $x_{\text{whitened}}(t)$ is the resultant whitened time-series, $x(t)$ is the original unwhitened data, and $g(t)$ is the FIR filter generated from the off-source ASD. This convolution effectively divides the power of the noise at each frequency by the corresponding value in $G(f)$. This flattens the PSD, making the noise uniform across frequencies; see Figure 26 for an example of this transform being applied to real interferometer data.

This method was adapted from the GWPy Python library [77] and converted from using NumPy functions [33] to TensorFlow GPU operations [34] in order to work in tandem with the rest of the training pipeline and allow for rapid whitening during the training process.

4.2.7.3 Pearson Correlation

A method of feature engineering that is employed prominently by the MLy pipeline [51] involves extracting cross-detector correlation using the Pearson correlation. The Pearson correlation is given by

$$r = \frac{N\left(\sum_{i=0}^{N} x_i y_i\right) - \left(\sum_{i=0}^{N} x_i\right)\left(\sum_{i=0}^{N} y_i\right)}{\sqrt{\left[N\sum_{i=0}^{N} x_i^2 - \left(\sum_{i=0}^{N} x_i\right)^2\right] \times \left[N\sum_{i=0}^{N} y_i^2 - \left(\sum_{i=0}^{N} y_i\right)^2\right]}} \tag{68}$$

where r is the Pearson correlation coefficient, N is the number of data points in each input array, and x_i and y_i are the i$^{\text{th}}$ elements of the $\vec{x}$ and $\vec{y}$ arrays respectively.

Nominally, this produces one scalar output value given two input vectors, $\vec{x}$ and $\vec{y}$, of equal length, N. A value of $r = 1$ indicates perfect correlation between the two vectors, whereas a value of $r = -1$ indicates perfect anti-correlation. Finally, a value of $r = 0$ indicates no correlation between the vectors. Note that if one of the vectors is entirely uniform, then the result is undefined.

This calculation assumes that the two vectors are aligned such that the value in x_i corresponds to the value in y_i. If this is not the case, as would happen for interferometer data if there is an arrival time difference (which there will be for most sky locations), then this will be an imperfect measure of correlation, even discarding the obfuscation of the noise. Because, as was discussed previously in Section 4.2.4, we do not know the direction of the source a priori, MLy [51] calculates the correlation for all possible arrival times given the light travel time between the two detectors in question. It uses minimum increments of the sample duration so that no heterodyning is necessary. This is done with the assumption that any difference in arrival time less than the sample duration will have a negligible effect on the correlation. It should be noted that this method is still hampered by the different polarisation projections dependent on the source polarization and by the obfuscating noise. See Figure 28 for examples of the rolling Pearson correlation calculated for LIGO Hanford and LIGO Livingston interferometer data.

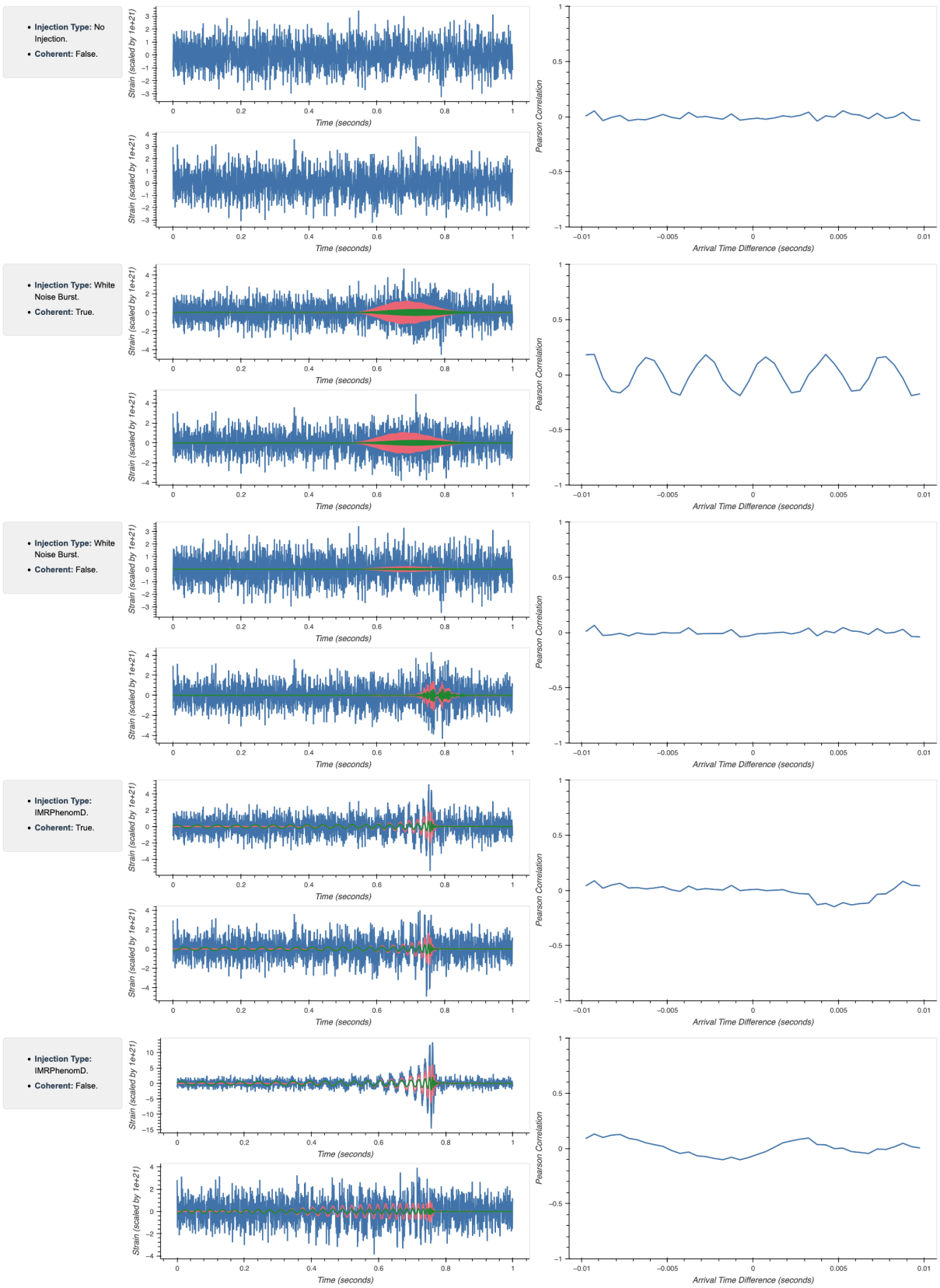


Figure 28 | Example whitened on-source and correlation plots of real interferometer noise from a pair of detectors, in this case, LIGO Livingston and LIGO Hanford, with either coherent, incoherent, or no injections added. The leftmost plots adjacent to the info panels are grouped into pairs. In each case, LIGO Livingston is at the top, and LIGO Hanford is underneath. Identical on-source and off-source noise segments were used for each example of the same detector, and noise for each detector

was gathered with a time difference of no more than 2048.0 s. In the leftmost plots, the green series is the unwhitened but projected waveform to be injected into the real noise from that detector. The red series is that same injection but subject to the same whitening procedure that will also be applied to the on-source plus injections, and the blue series is the whitened on-source plus injections. The rightmost plots each correspond to a pair of detectors and display the rolling Pearson correlation values between those two whitened on-source plus injection series. Since there is approximately a max arrival time difference of 0.01 s between LIGO Livingston and LIGO Hanford, the number of correlation calculations performed corresponds to the rounded number of samples required to represent 0.02 s of data at 2048.0 Hz. This number is two times the maximum arrival time difference because the difference could be positive or negative. In this case, that difference comes to 40 samples. All injections have been scaled to an optimal network SNR of 30 using the method described in Section 4.2.5.2. The upper pair of detectors has no injection. As would be expected, the correlation is low regardless of the assumed arrival time difference. The second pair from the top has been injected with a coherent white noise burst (WNB), see Section 4.2.3, which has been projected onto the two detectors using a physically realistic mechanism previously described in Section 4.2.4. Here, the correlation is much stronger. We can see it rise and fall as the waveforms come in and out of coherence. The third from the top, the central plot, shows an injection of two incoherent WNBs. They are processed identically to the coherent case, but the initial waveforms are generated independently, including their durations. The Pearson correlation looks very similar to the pure noise case in the uppermost plot, as might be expected. The second from the lowest pair has been injected with a coherent IMRPhenomD waveform, which again has been correctly projected. We can observe that a small correlation is observed at an arrival time difference of around 0.005 s, suggesting that the two waveforms arrived at the detectors 0.005 s apart. Finally, the lowest plot depicts two incoherent IMRPhenomD waveforms projected into the noise. Though these are generated with different parameters, the shared similarities in morphology between all CBC waveforms cause correlation to be registered. By maximum amplitude alone, it may even appear as though there is more correlation happening here than in the correlated case. This highlights one potential weakness of using the Pearson correlation, which can sometimes show some degree of correlation even if the two waveforms were not produced using the same physically simulated mechanism.

As with most mathematical functions, we have created a new GPU-based function for the calculation of the Pearson correlation in Python [32], using the TensorFlow GPU library [34] for computational speed and easy integration with the rest of the pipeline.

4.2.7.4 Fourier Transform

So far, we have looked at data conditioning, which produces results in the time domain. As we know, and as has been demonstrated by the previous discussion, many aspects of time series processing are performed in the frequency domain. Often, features that are hard to distinguish in the time domain are relatively easy to spot in the frequency domain, even with the human eye. Many have characteristic morphologies, such as distinct lines due to powerline harmonics and violin modes. If we make the assumption that if it is easier for a human, it might also be easier for a machine learning method, we should certainly examine feature engineering methods that take us into the frequency domain. The most obvious way to do this would be to use a simple Fourier transform, which takes us directly from a time-domain series to a frequency-domain one. The discrete form of the Fourier transform is given above in Equation 58.

4.2.7.5 Power Spectral Density (PSD) and Amplitude Spectral Density (ASD)

As discussed in Section 4.2.1, the PSD is used in many calculations and transforms in gravitational wave data analysis, so it makes sense that along with the closely related property, the ASD, it may

also be useful information to provide to a model. Since the PSD has already been discussed in detail in Section 4.2.1, we will not linger on it here.

4.2.7.6 Spectrograms

The final feature engineering method that we will discuss allows us to represent data in both the time and frequency domains simultaneously. Spectrograms are visualisations of the Short-Time Fourier Transform (STFT) of a time series. The STFT is computed by dividing a time series into many smaller periods, much like in the calculation of a PSD; however, instead of being averaged, you can simply use this 2D output as an image in its own right, which displays how the frequency components of a time series fluctuate over its duration. This retains some information from the time domain. The 2D STFT of a continuous time series, $x(t)$, is given by

$$\mathrm{STFT}(x)(t, f) = \int_{-\infty}^{\infty} x(\tau) w(t - \tau) e^{-i 2 \pi f \tau} d\tau \tag{69}$$

where $\mathrm{STFT}(x)(f, t)$ is the value of the STFT of $x(t)$ at a given time, t, and frequency, f, $w(t)$ is a configurable window function that helps to minimize the boundary effects, and τ is a dummy integration variable used to navigate through the time domain at the expense of losing some information from the frequency domain, making the spectrogram, like whitening, a lossy transform. In its discrete form, this becomes

$$\mathrm{STFT}(x)[n, k] = \sum_{m=0}^{N-1} x[m] w[n - m] e^{\frac{-i 2 \pi k m}{N}} \tag{70}$$

where $\mathrm{STFT}(x)[n, k]$ is the value of the discrete STFT of a discrete time series, $x[m]$ at a given time index, n, and frequency index, k, $w[t]$ is a discrete window function, and N is the number of samples in our discrete time series. It should be noted that there are two time indices present, n and m, because a reduction in dimensionality along the time axis usually occurs since the step between adjacent FFT segments is commonly greater than one.

When creating a spectrogram, the values are typically squared,

$$S[k, n] = \left(\mathrm{STFT}(x)[n, k]\right)^2 \tag{71}$$

to represent the power of the frequency components, similar to the process of calculating the PSD. Alternatively, the magnitude can be taken with

$$S[k, n] = |\,\mathrm{STFT}(x)[n, k]|. \tag{72}$$

Before plotting, the data is often converted into decibels to better visualise the dynamic range,

$$\mathrm{DATA} = 10 \times \log(S[k, n]). \tag{73}$$

We have created a custom Python TensorFlow function [34] to perform these calculations on the GPU; see Figure 29 for illustrations of this in use on real noise with injected waveform approximants. As is the case with multiple 1D time series, the question also remains of how to combine multiple spectrograms in the case of multiple detector outputs, see Section 4.2.6.

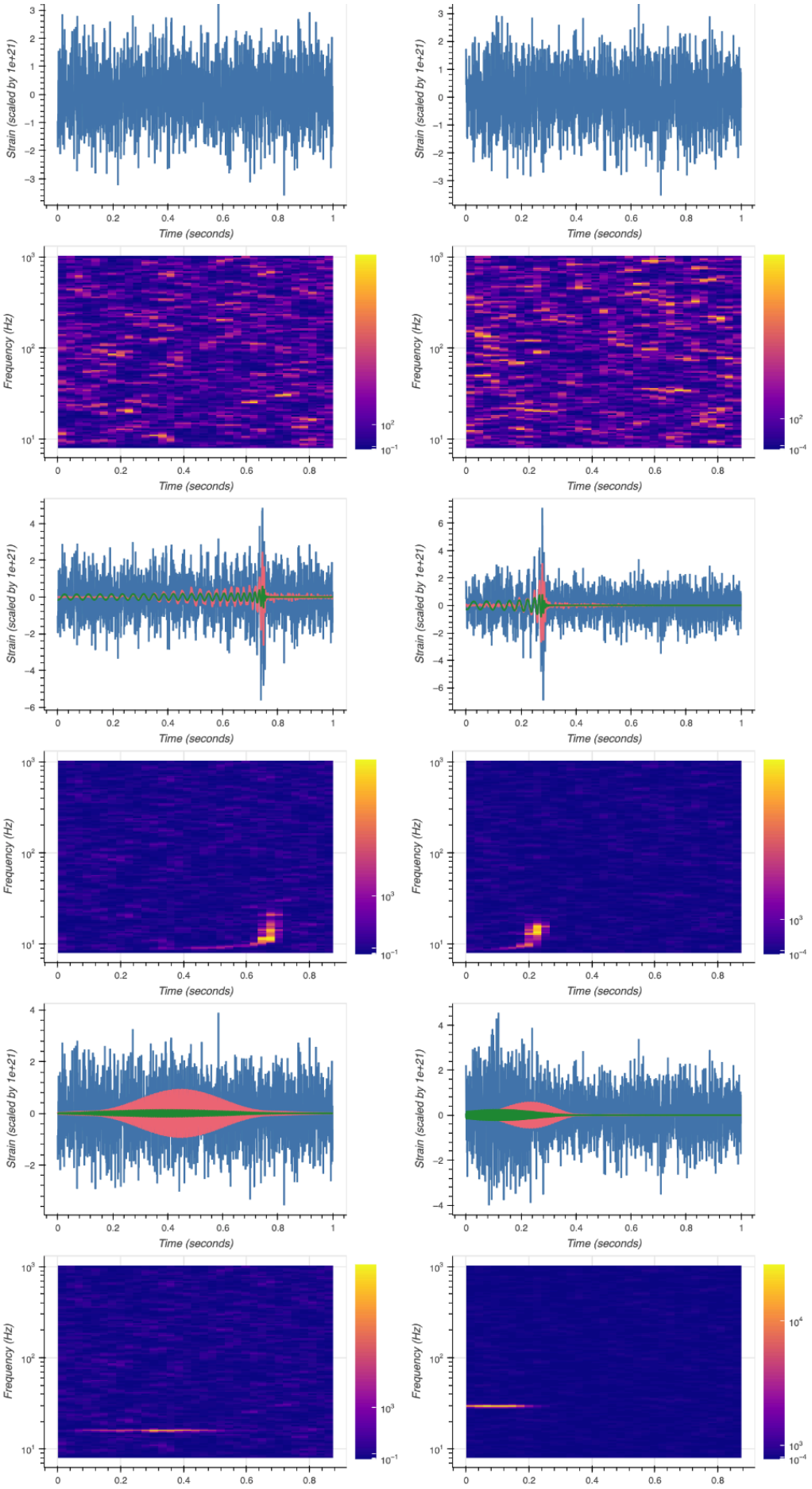

Strain (scaled by 1e+21)
Time (seconds)
Frequency (Hz)

Figure 29 | Six example noise segments and their corresponding spectrograms. In all cases, the noise is real interferometer data acquired from the LIGO Hanford detector during the 3rd observing run. It is whitened using the procedure described in Section 4.2.7.2. For the time series plots, the green series represents the original, unwhitened waveform before injection, the red series is the waveform with the same whitening transform applied to it as was applied to the on-source background plus injection, and the blue series is the whitened on-source background plus injection, except for the first two time series plots which contain no injection. The spectrograms were generated using the STFT described by Equation 70, converted into power with Equation 71, and finally transformed into a decibel logarithmic scale for plotting using Equation 73. The two uppermost plots and their respective spectrograms have no injections. The two middle plots and their respective spectrograms have IMRPhenomD [79] approximants created with cuPhenom injected into the noise, and the two lower plots and their respective spectrograms, have White Noise Burst (WNB) waveforms generated using the method described in Section 4.2.3, injected into the noise. In all cases, the injections were scaled to an optimal SNR randomly selected between 15 and 30; these are quite high values chosen to emphasise the features in the spectrograms. As can be seen, the whitened noise that contains injected features has spectrograms with highlighted frequency bins that have a magnitude much larger than the surrounding background noise; the different signal morphologies also create very different shapes in the spectrograms. This allows us to see the frequency components of the signal more easily, observe the presence of interesting features, and differentiate between the WNB and the CBC case.

4.2.7.7 Summary

There are multiple different possibilities for how to condition the data before it is fed into any potential machine learning model; see Table 3, and we have only covered some of the possibilities. Most methods come at the cost of removing at least some information from the original data. It remains to be seen, however, if this cost is worthwhile to ensure adequate model performance and feasible training durations.

Possible Model Inputs	Dimensionality of Output	Output Domain
Raw Onsource + Injection	1	Time
Whitened Onsource + Injection	1	Time
Pearsons Corrleation	1	Time
Fourier Transform (PSD)	1	Frequency
Power Spectral Density (PSD)	1	Frequnecy
Spectrogram	2	Time and Frequency

Table 3 | A non-exhaustive table of possible data conditioning modes. Feature engineering is often used in order to simplify a problem before it is presented to a machine learning model. There are many ways we could do this with gravitational-wave data. Presented are some of the most common. Each is described in more detail in Section 4.2.7.

4.2.8 Transient Glitch Simulation

As has previously been noted, as well as a quasi-stationary coloured Gaussian background, interferometer noise also contains transient detector glitches caused by a plethora of sources, both known and unknown. These glitches have a prominent effect on the upper-sensitivity bound of most types of search, so it may be important to represent features of this type in our training pipeline. Previous experiments performed during the development of the MLy pipeline [51] had shown that networks can often have greatly increased FARs when performing inference on data segments that contain transient glitches, even when those glitches were only present in the off-source segment used to generate the PSD used for data whitening. As such, a method to add glitches to the training distribution should be considered so that methods to deal with features of this type can hopefully be incorporated into the model's learned parameters during training.

There have been multiple attempts to classify and document the many transient glitches found in real interferometer data [84], [85], [86] , both through automated and manual means [87]. During operation within a standard observing run, there are both intensive manual procedures [88] to characterise the detector state and automated pipelines such as the iDQ pipeline [89]. There is also a large amount of work done offline to characterise the noise in a non-live environment [88]. These methods utilise correlation with auxiliary channels, frequency of triggers, and other information about the detector state to ascertain the likelihood that a given feature is a glitch or of astrophysical origin.

One of the most prominent attempts to classify transient glitches is the Gravity Spy project [86], which combines machine learning and citizen science to try and classify the many morphologies of transient glitches into distinct classes. Successful methods to classify glitches are highly useful since if a similar morphology appears again in the data it can be discounted as a probable glitch. Gravity Spy differentiates glitches into 19 classes plus one extra "no_glitch" class for noise segments that are proposed that do not contain a glitch. The other 19 classes are as follows: air_compressor, blip, chirp, extremely_loud, helix, koi_fish, light_modulation, low_frequency_burst, low_frequency_lines, none_of_the_above, paired_doves, power_line, repeating_blips, scattered_light, scratchy, tomte, violin_mode, wandering_line, and whistle. Some types, such as blips, are much more common than others.

There are two options we could use as example data in our dataset in order to familiarise the model with glitch cases. We could either use real glitches extracted from the interferometer data using the timestamps provided by the Gravity Spy catalog [86] or simulated glitches we generate ourselves. The forms of each would vary depending on whether it was a multi, or single-detector example and whether we are attempting to detect CBCs or bursts.

Real Glitches: The addition of real glitches to the training dataset is a fairly intuitive process, though there are still some parameters that have to be decided upon. By using timestamps from the Gravity Spy catalog [86], we can extract time segments of equal length to our example segments, which contain instances of different classes of glitches. We should process these identically to our regular examples with the same whitening procedure and off-source segments. Real glitches have the distinct advantage that any model will be able to use commonalities in their morphology to exclude future instances; this is also, however, their disadvantage. If you train a model on specific morphologies, then the introduction of new glitch types in future observing runs, which may well be possible given the constant upgrades and changes to detector technology, then it may be less capable of rejecting previously unseen glitch types [86]. However, it is still possible that these glitches will help the model to reject anything other than the true type of feature it has been trained to recognise by weaning it off simple excess power detection.

Simulated Glitches: The other option is to use simulated glitches. The form of these glitches depends highly on the nature of the search, primarily because you wish to avoid confusion between the mor-

phology of the feature you want your method to identify and your simulated glitches. For example, in a CBC search, you could use WNBs as simulated glitches, as their morphologies are entirely distinct, and there is no possibility of confusion. However, if we are using coherent WNBs across multiple detectors to train a model to look for coherence, then we must be careful that our glitch cases do not look indistinguishable from true positive cases, as this would poison the training pool by essentially mislabeling some examples. We could, in this case, use incoherent WNBs as simulated glitches as, ideally, we want our coherent search to disregard incoherent coincidences. This is the approach taken by the MLy pipeline [51], as a method to train the models to reject counterexamples of coherent features.

Other than the question of whether to use simulated or real glitches or maybe even both, a few questions remain: what is the ideal ratio between examples of glitches and non-glitched noise examples? Should the glitched background also be injected with waveforms at some rate? A real search would occasionally see glitches overlapping real signals, though this would occur in a relatively low number of cases, and including these types of signal-glitch overlaps could perhaps interfere with the training process whilst not adding a great deal of improvement to the true positive rate. Should glitches form their own class so that the model instead has to classify between signal, noise, or glitch rather than just signal or noise? These questions must be answered empirically.

For the multi-detector case, and thus also the burst detection case, we must decide how to align glitches across detectors. It seems safe to assume that adding coherent glitches across multiple detectors would be a bad idea in a purely coherence-based search pipeline — although perhaps if the model can learn to disregard certain morphologies based on prior experience, this would be a nice extension. For some simple glitch types, coincident and fairly coherent instances across detectors are not extremely unlikely. For example in the case of the most common glitch class identified by GravitySpy [86], blips, we often see coincident glitches in multiple detectors with a physically plausible arrival time difference, and because they are only glitches, their morphologies can often be similar.

We could also include cases of incoherent glitches across detectors but of the same class, incoherent glitches across detectors but of different classes, and any combination of glitches found in less than the full complement of detectors. Perhaps it would be the case that a good mix of all of these cases would better inoculate our model against glitches.

4.3 Perceptron Results

Now that we have finally assembled all the pieces required to generate training, testing, and validation datasets that can approximate real noise segments, we can finally repeat the experiments we performed on the MNIST data in Section 3.1.3, with both single and multi-layer perceptrons. The model architectures are similar, though the input vectors are now the size of our simulated interferometer output examples: `(NUM_EXAMPLES_PER_BATCH, NUM_SAMPLES)` in the case of the single detector CBC search and `(NUM_EXAMPLES_PER_BATCH, NUM_DETECTORS, NUM_SAMPLES)` in the multi-detector coherent burst search. We will use 32 training examples per batch, `NUM_EXAMPLES_PER_BATCH = 32`, as this is a standard power-of-two value used commonly across artificial neural network literature, and, in the multi-detector case, we will use only LIGO Hanford and LIGO Livingston, for now, excluding the Virgo detector, `NUM_DETECTORS = 2`. We have chosen to use only the two LIGO detectors as in many ways, this is the simplest possible multi-detector network case; signals projected onto these two detectors will have a greater similarity than signals projected onto either of these two detectors and the Virgo detector, both due to sensitivity and orientation and position differences. We have chosen to use a sample rate of 2048.0 Hz and an on-source duration of 1.0 s, allowing an additional crop region 0.5 s either side of the onsource segment to remove edge effects created when whitening with 16.0 s of off-source background. The reasoning for these choices has been described previously in this chapter. This means we will have 2048 samples per detector, `NUM_SAMPLES = 2048`, after it has been passed through the whitening layer. A flattening layer will only be required in the multi-detector case, in the

single-detector case, the input is already one-dimensional. The batch dimensions are not a dimension of the input data and simply allow for parallel processing and gradient descent; see Section 3.2.

The obfuscating noise consisted of real data taken from LIGO Hanford and LIGO Livingston for each respective detector. Locations of confirmed and candidate events are excluded from the data, but known glitch times have been included in the training, testing, and validation datasets.

For the single CBC case, cuPhenom waveforms with masses drawn from uniform distributions between 5.0 $M_\odot$ and 95.0 $M_\odot$ for the mass of both companions, and between −0.5 and 0.5 for the dimensionless spin component are injected into the noise and scaled with optimal SNR values taken from a uniform distribution of between 8.0 and 15.0 unless explicitly stated.

For the multi-detector Burst case, a coherent WNB was injected with durations between 0.1 s and 1.0 s, and the frequencies were limited to between 20.0 Hz and 500.0 Hz. The injected bursts were projected correctly onto the detectors using a physically realistic projection. The bursts were injected using the same scaling type and distribution as the CBC case, although notably, the network SNR was used rather than a single detector SNR.

During network training, the gradients were modified by batches consisting of 32 examples at a time, chosen as an industry standard batch size, and with a learning rate of 1.0×10^{-4}, and using the Adam optimiser, which again is a common standard across the industry. During training epochs, 10^5 examples were used before the model was evaluated against 10^4 example of the previously unseen test data. It should be noted that due to the nature of the generators used for the training, unlike in standard model training practices, no training examples were repeated across epochs, but the test dataset was kept the same for each epoch. After each epoch, if the validation loss for that epoch was the lowest yet recorded, the model was saved, replacing the existing lowest model. If no improvement in validation loss was seen in ten epochs (patience), the training was halted, and the best model was saved for further validation tests. Table 4 shows a large number of the training and dataset hyperparameters.

Hyperparameter	**Value**
Batch Size	32
Learning Rate	10^{-4}
Optimiser	Adam
Scaling Method	SNR
Minimum SNR	8.0
Maximum SNR	15.0
SNR Distribution	Uniform
Data Acquisition Batch Duration	2048.0 s
On-source Duration	1.0 s
Off-source Duration	16.0 s

Scale Factor	10^{21}

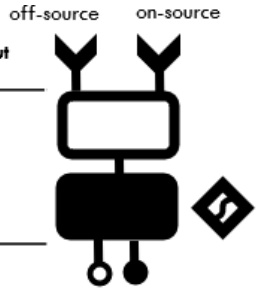
Table 4 | The common training and dataset hyperparameters shared by the CBC and Burst perceptron experiments. Note that the scale factor here refers to the factor used during the upscaling of the CBC waveforms and real interferometer noise from their extremely small natural dimensions to make them artificial neuron-friendly. This is done both to ensure that the input values work well with the network activation functions and learning rates, which are tuned around values near one, and to reduce precision errors in areas of the code that use 32-bit precision, employed to reduce memory overhead, computational cost and duration. Data acquisition batch duration is a parameter of the GWFlow data acquisition module. For speed, the GWFlow data acquisition system downloads data in larger segments than is required for each training batch, then randomly samples examples from this larger segment to assemble each training batch. The data acquisition batch duration determines how long this larger batch is. Smaller values will result in a more evenly mixed training data set and a lower overall GPU memory overhead but will be more time-consuming during the training process.

4.3.1.1 Architectures

We used architectures with four different layer counts: zero, one, two, and three hidden layers; see Figure 30. All models have a custom-implemented whitening layer, which takes in two vectors, the on-source and off-source segments, and performs a whitening operation as described in Section 4.2.7.2. They also all have a capping dense layer with a single output value that represents either the presence of a feature or the absence of one. The capping layer uses the Sigmoid activation function; see Equation 13, and the other hidden layers use ReLU activation functions, see Equation 10.

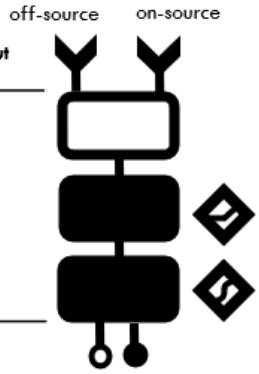
Layers are built with a number of neurons selected from this list $[64, 128, 256, 512]$, though fewer combinations were tested in architectures with a greater number of model layers. Models tested have these 14 configurations of neuron numbers per layer, specified as [num_hidden_layers:num_neurons_in_layer_1 … num_layers_in_layer_n]: ([0], [1:64], [1:128], [1:256], [1:512], [2:64,64], [2:128,64], [2:128,128], [2:256,64], [2:256,128], [2:256,256], [3:64,64,64], [3:128,128,128], [3:256,256,256]). These combinations were chosen to give a reasonable coverage of this section of the parameter space, though it is notably not an exhaustive hyperparameter search. From the performances demonstrated in this search compared to other network architectures, it was not deemed worthwhile to investigate further.

Type	Num Filters / Neurons	Filter / Pool Size	Stride	Activation Function	Dropout
Whitening	-	-	-	-	-
Dense	1	-	-	SoftMax	-



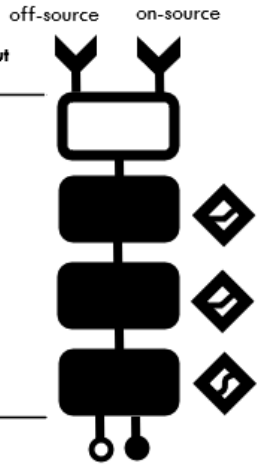
Binary Classification

Type	Num Filters / Neurons	Filter / Pool Size	Stride	Activation Function	Dropout
Whitening	-	-	-	-	-
Dense	64/128/256	-	-	ReLU	-
Dense	1	-	-	SoftMax	-



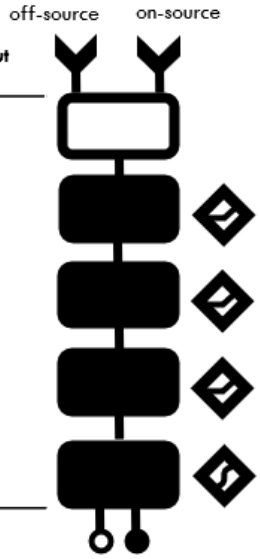
Binary Classification

Type	Num Filters / Neurons	Filter / Pool Size	Stride	Activation Function	Dropout
Whitening	-	-	-	-	-
Dense	64/128/256	-	-	ReLU	-
Dense	64/128/256	-	-	ReLU	-
Dense	1	-	-	SoftMax	-



Binary Classification

Type	Num Filters / Neurons	Filter / Pool Size	Stride	Activation Function	Dropout
Whitening	-	-	-	-	-
Dense	64/128/256	-	-	ReLU	-
Dense	64/128/256	-	-	ReLU	-
Dense	64/128/256	-	-	ReLU	-
Dense	1	-	-	SoftMax	-



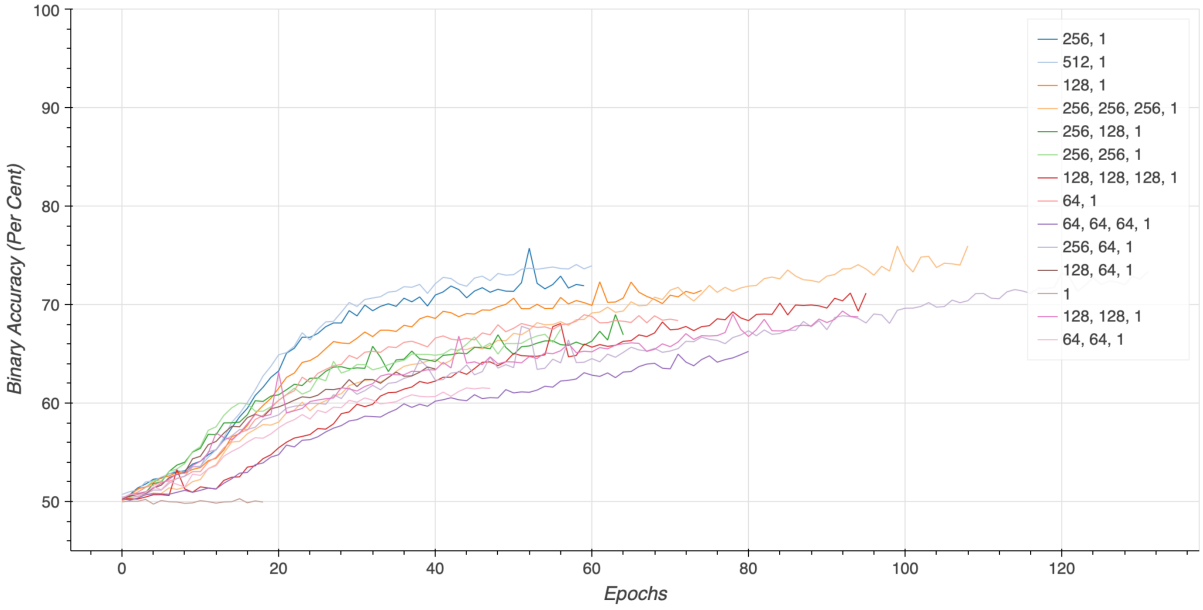
Binary Classification

Figure 30 | Perceptron diagrams. The four different architectures used to test the use of purely dense models for both the single-detector CBC detection case and the multi-detector burst detection problem. The only differences are that the input vector sizes were different between the cases: `(NUM_EXAMPLES_PER_BATCH, NUM_SAMPLES)` in the case of the single detector CBC search and `(NUM_EXAMPLES_PER_BATCH, NUM_DETECTORS, NUM_SAMPLES)` in the multi-detector coherent burst search. All models take in two input vectors into a custom-designed GWFlow whitening layer, the off-source and the on-source vectors; see Section 4.2.7.2 for more information about the whitening procedure, and all models are capped with a dense layer with a single output neuron that is used to feed the binary loss function, with a sigmoid activation function. Each hidden layer has been tested with 64, 128, and 256 neurons: *Top:* Zero-hidden layer model. *Second to top:* Two-hidden layer model. *Second to bottom:* Three-hidden layer model. *Bottom:* One hidden layer model.

4.3.2 CBC Detection Dense Results

4.3.2.1 Training

First, we can examine the results of applying dense-layer perceptrons to the CBC single-detector morphology detection problem. Even during the training process, it is clear that, at least amongst the selected hyperparameters, these models are not going to be useful. See Figure 31 and Figure 32. None reach an accuracy of above 75% with a training patience of ten epochs. A training patience of ten means that if no improvement in the validation loss is seen within ten epochs, the training process is halted. Examining the plots; see Figure 31 and Figure 32, it seems possible that some of the perceptrons are on a very slow training trajectory and could have seen some marginal improvement if the training patience had been increased. It is also possible that other larger perceptron architectures may achieve greater success, as this was far from an exhaustive or even guided search of the perceptron hyperparameter space. However, as can be seen in Figure 31, the models take a significant number of epochs to reach the values they do, which is what we would expect from entirely dense models. As will be seen in later sections Section 4.5, other architectures can achieve much better results in fewer epochs. These results are here to act as an example of the difficulties of training dense networks for complex recognition tasks. For comparison with other methods, a more sophisticated analysis will be shown after the training history plots with SNR efficiency curves and FAR analysis.



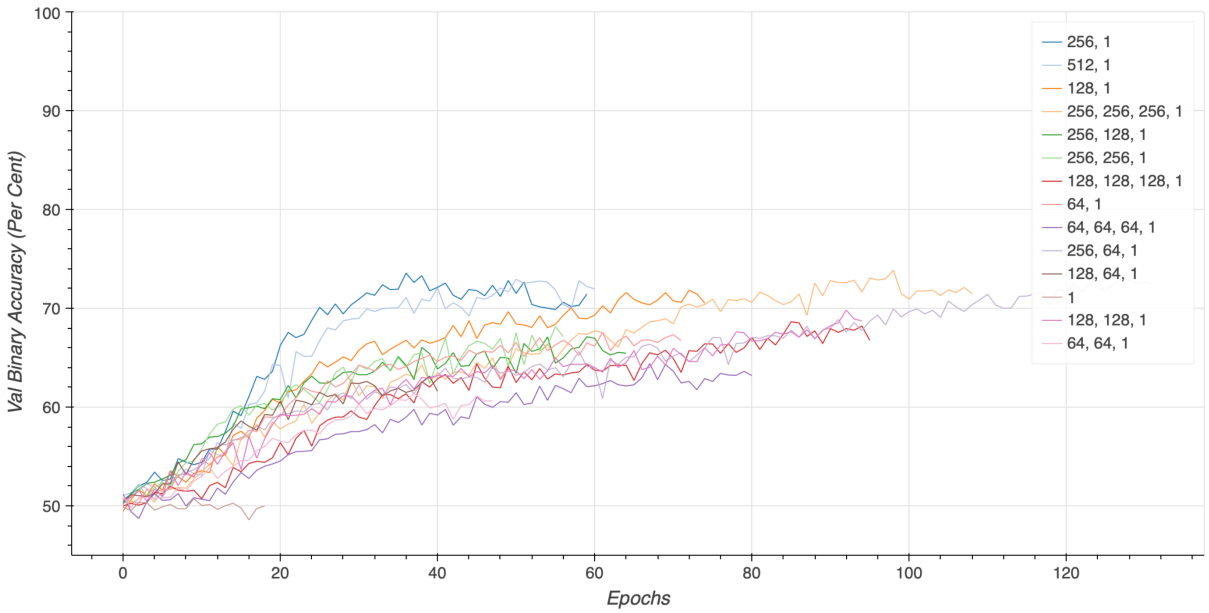


Figure 31 | Training loss history of perceptron models training to detect IMRPhenomD waveforms, generated using cuPhenom, obfuscated by real interferometer noise from the LIGO Livingston detector from the 3rd observing run. Visit https://rawcdn.githack.com/mrknorman/data_ad_infinitum/c619a91b36513a2244ccd610c709bb1643a64b4c/models/chapter_04_perceptrons_single/training_history.html for interactive plots, whilst they're still working. The optimal SNR of waveforms injected into the training and validation sets was uniformly distributed between 8 and 15. Input was from a single detector only. A rough search was performed over a relatively arbitrary selection of model architectures, which varied the number of layers and the number of perceptrons in each layer. The architectures of each model can be seen in the figure legends as a list of numbers where each number is the number of artificial neurons in that layer. All were trained with the same training parameters, details of which can be found at Table 4. Each epoch consisted of 10^5 training examples, and it should be noted that, unlike the regular training pipelines, each training epoch consisted of new unseen waveforms injected into unseen noise segments, though the validation examples were consistent. Training of each model was halted after ten consecutive epochs with no improvement to validation loss, the values of which are shown in Figure 32. Validation noise was drawn from a separate pool of GPS segments inaccessible to the training data loader. We can see that the accuracy only approaches 75%. Though this is from a pool of mixed SNRs and at an unrestrained false alarm rate (this accuracy uses a score threshold of 0.5 regardless of SNR), this is insufficient to be useful. *Upper:* Plot of model accuracies when mesured with training data (10^5 epoch unique examples). *Lower:* Plot of model accuracies when mesured with validation data (10^4 epoch-consistent examples).

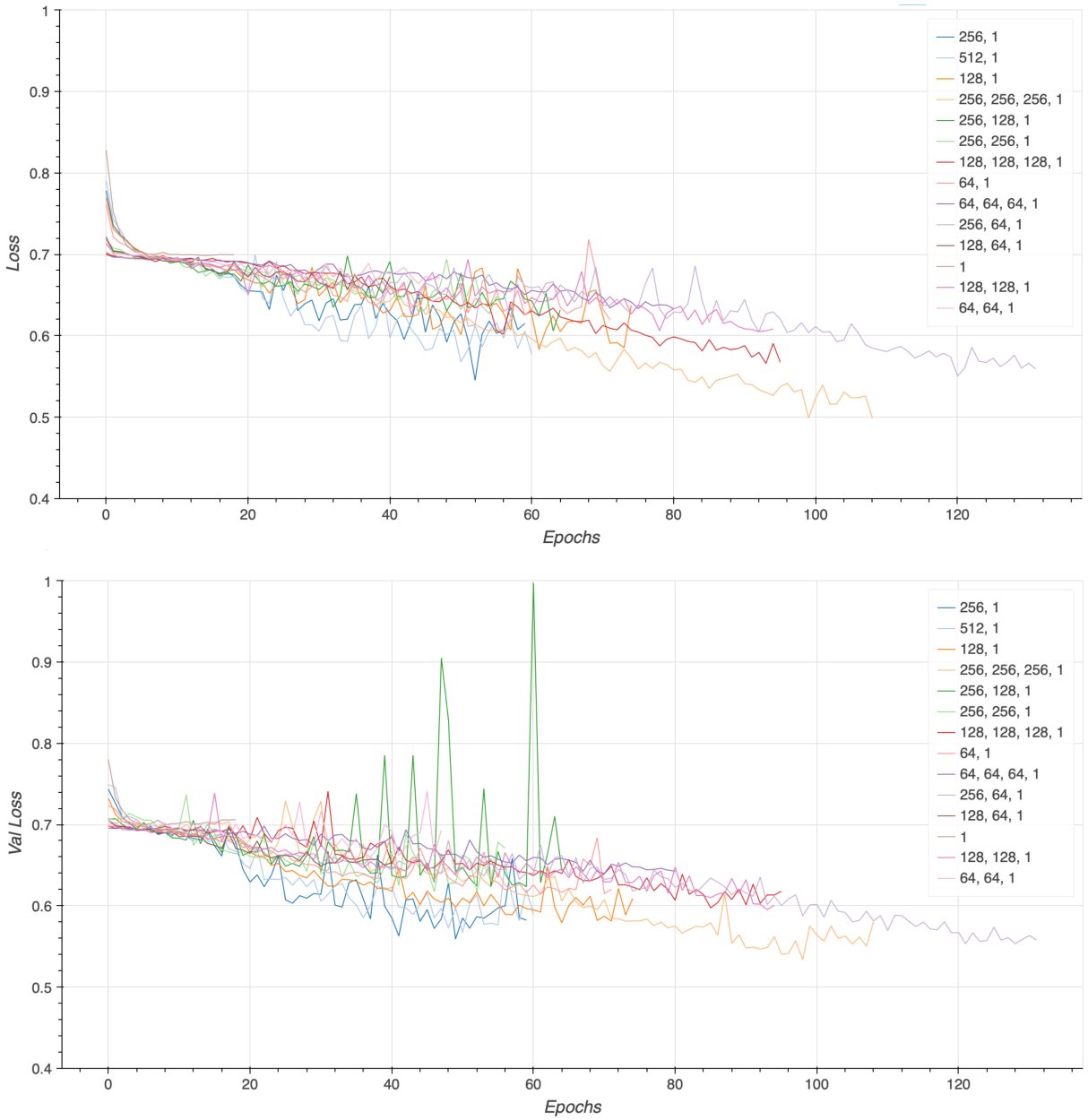


Figure 32 | Training LOSS history of perceptron models training to detect IMRPhenomD waveforms, generated using cuPhenom, obfuscated by real interferometer noise from the LIGO Livingston detector from the 3rd observing run. Visit https://rawcdn.githack.com/mrknorman/data_ad_infinitum/c619a91b36513a2244ccd610c709bb1643a64b4c/models/chapter_04_perceptrons_single/training_history.html for interactive plots, whilst they're still working. See Figure 31 for a more detailed description of the training data. *Upper:* Plot of model loss when measured with training data (10^5 epoch-unique examples). *Lower:* Plot of model loss when mesured with validation data (10^4 epoch-consistent examples).

4.3.2.2 Validation

Although the perceptron training performance was low, and probably sufficient to tell us that at least these configurations of perceptrons are not capable enough for CBC detection, a more complete validation was nonetheless performed on the trained models using the third as-yet-unseen validation data. This was both for comparison with later methods and to ensure our initial assessment of the results was correct. Although it is easy to draw quick conclusions from the training results, it is not an accu-

racy profile of the model performance, as the training validation results draw from a mixed pool of SNR values, do not consider the classes independently, and in the case of the accuracy result, use an uncalibrated detection threshold score of 0.5. This means that if a model outputs a score over 0.5 it is considered a detection, and a score lower than 0.5 is considered noise. By tuning this threshold, we can arrive at the desired False Alarm Rate (FAR), though this will have an inverse effect on the sensitivity, (the true positive rate) of the model.

Before we can apply this tuning we must evaluate our model's performance on a dataset consisting exclusively of noise examples. The perfect classifier would output zero for all examples in such a dataset. We are not dealing with perfect classifiers, so the model will output a score value for each pure noise example. If our classifier has good performance most of these scores will be low, preferably near zero, but some will inevitably rise above whatever detection threshold we set, dependant of course on the size of our FAR validation dataset, the larger the dataset the larger the expected value of our largest false positive score. The size of the dataset, required for threshold calibration will depend on the the value of FAR that is desired, with smaller false alarm rates requiring larger datasets. We will require a dataset in which the combined example durations sum to at least the duration of time wherein, given our FAR, we would expect one detection. However, since this is a statistical result, having only the exact duration required for our FAR would result in a great deal of error on that value. The larger the validation dataset, the more confident we can be in our calculation of the required FAR threshold. We will attempt to use a validation dataset around ten times larger than the minimum required, so we would expect 10 false alarms total from running the model on the dataset with the given threshold.

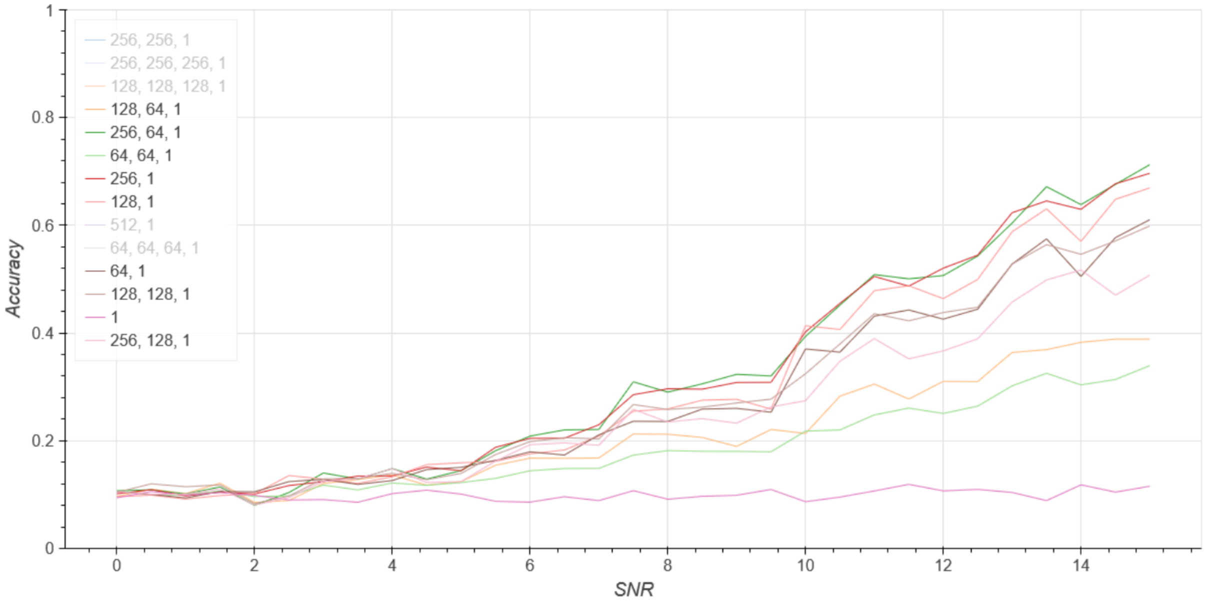
Of course, there is only so FAR we can tune the threshold value within the precision value available to us with 32-bit floats, and if the model gives scores to noise examples of exactly one, there is no way to differentiate them from other classifications. This means any model will have a maximum possible FAR beyond which it cannot distinguish positive results from negative ones.

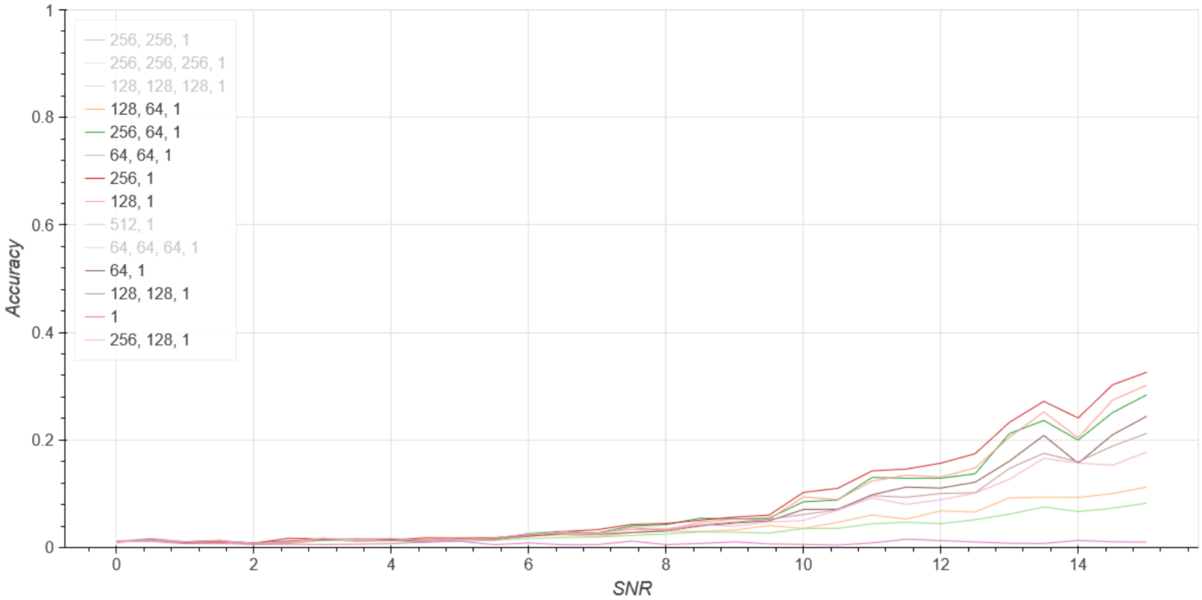
In order to determine the score threshold of a model for a given FAR, we can run that model over a sufficiently large FAR dataset, sort these scores from smallest to highest, and then assign each score an equivalent FAR. For example, if we sorted the scores from lowest to highest, and the first score was above the score threshold, then the FAR would be $\frac{1.0}{d_{\text{example}}}$ Hz, where d_{example} is the length of the input example in our case 0.1 s. If we set the threshold to be smaller than the smallest score this would mean that all the scores are above the threshold, therefore the model would produce a false alarm every time it ran. If the second sorted score was above the threshold but not the first, then all but one of the examples would be a false alarm, therefore the score would be $\frac{d_{\text{total}}-d_{\text{example}}}{d_{\text{total}}} \times \frac{1.0}{d_{\text{example}}}$ Hz, d_{total} is the total duration of examples in the validation set. This gives a general formula for the x-axis.

$$x = \frac{d_{\text{total}} - id_{\text{example}}}{d_{\text{total}}} \times \frac{1.0}{d_{\text{example}}} \text{ Hz} \tag{74}$$

,

where i is the x-axis index.


256, 256, 1
256, 256, 256, 1
128, 128, 128, 1
128, 64, 1
256, 64, 1
64, 64, 1
256, 1
128, 1
512, 1
64, 64, 64, 1
64, 1
128, 128, 1
1
256, 128, 1
Accuracy
SNR


256, 256, 1
256, 256, 256, 1
128, 128, 128, 1
128, 64, 1
256, 64, 1
64, 64, 1
256, 1
128, 1
512, 1
64, 64, 64, 1
64, 1
128, 128, 1
1
256, 128, 1
Accuracy
SNR

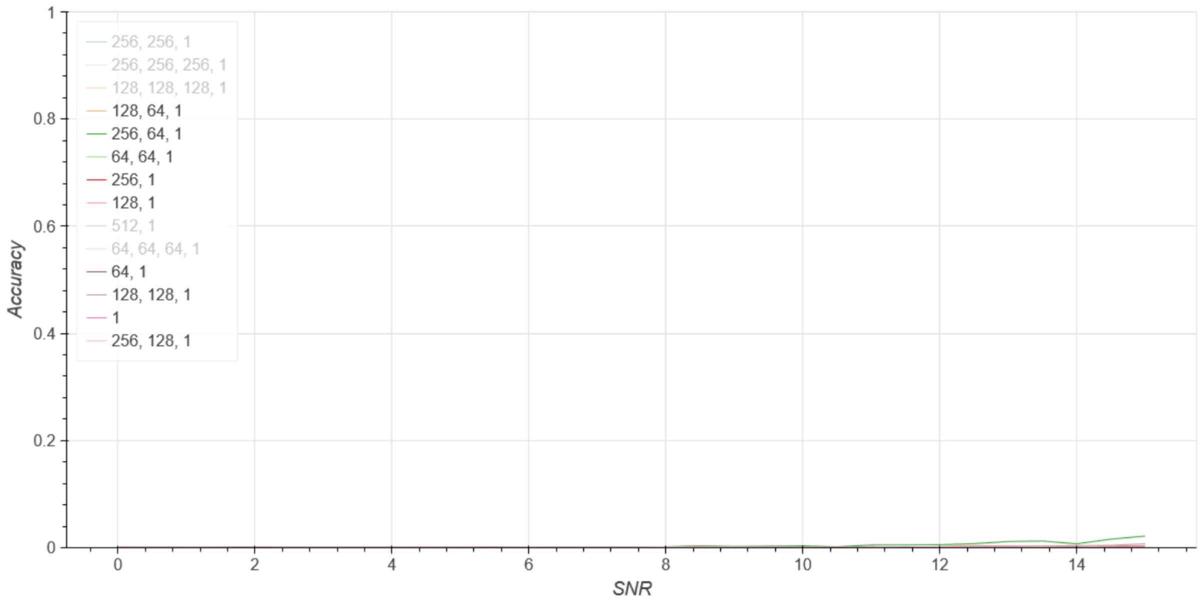

256, 256, 1
256, 256, 256, 1
128, 128, 128, 1
128, 64, 1
256, 64, 1
64, 64, 1
256, 1
128, 1
512, 1
64, 64, 64, 1
64, 1
128, 128, 1
1
256, 128, 1
Accuracy
SNR

Figure 33 | Perceptron efficiency curves. *Upper:* Efficiency curve at a False Alarm Rate (FAR) of 0.1 Hz. *Middle:* Efficiency curve at 0.01 Hz. *Lower:* Efficiency curve at 0.001 Hz.

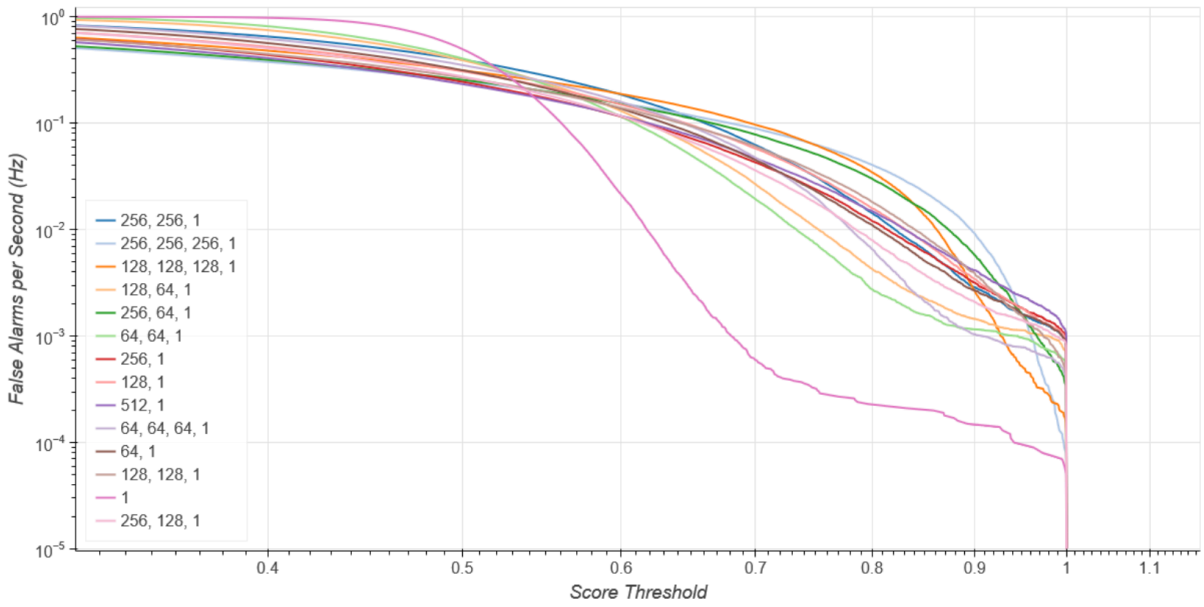


Figure 34 | Perceptron False Alarm Rate (FAR) curves.

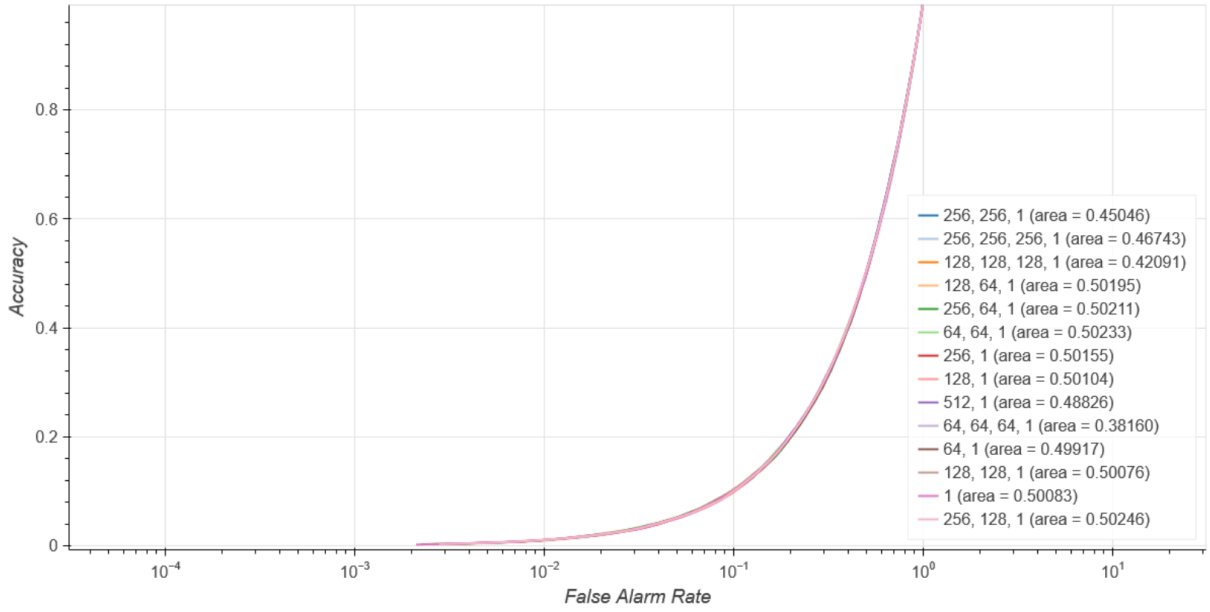


Figure 35 | Reciever Operator Curve (ROC) Curve at $\rho_{\text{opt}} = 8$.

4.3.3 Burst Detection Dense Results

Talk about the average SNR of the background at some point and how even Gaussian noise will generate a certain number of false alarms.

Show that dense layers cant do it

4.4 Introducing Convolutional Neural Networks (CNNs)

As we have seen, simple dense-layer perceptrons can not adequately perform detection tasks on gravitational-wave data. This was anticipated, given the complexity of the distribution. Perceptrons have not been at the forefront of artificial neural network science for some time. We must turn toward other architectures. Although, in some ways, specialising the network will limit the capacity of our model to act as a universal function approximator, in practice, this is not a concern, as we have at least some idea of the process that will be involved in completing the task at hand, in this case, image, or more correctly time-series recognition.

The Convolutional Neural Network (CNN) is currently one of the most commonly used model archetypes. In many ways, the development of this architecture was what kickstarted the current era of artificial neural network development. On 30$^{\text{th}}$ December 2012, the AlexNet CNN [14] achieved performance in the ImageNet multi-class image recognition competition, far superior to any of its competitors. This success showed the world the enormous potential of artificial neural networks for achieving success in previously difficult domains.

CNNs are named for their similarity in operation to the mathematical convolution, although it is more closely analogous to a discrete cross-correlation wherein two series are compared to each other by taking the dot product at different displacements. Unless you are intuitively familiar with mathematical correlations, I do not think this is a useful point of reference for understanding CNNs. So, I will not continue to refer to convolutions in the mathematical sense past this paragraph.

CNNs are primarily employed for the task of image and time-series recognition. Their fundamental structure is similar to dense-layer networks on a small scale. They are comprised of artificial neurons that take in several inputs and output a singular output value after processing their inputs in conjunction with that neuron's learned parameters; see Section 3.1.1. Typical CNNs ingest an input vector, have a single output layer that returns the network results, and contain a variable number of hidden layers. However, the structure and inter-neural connections inside and between the layers of a CNN are fundamentally different.

Unlike perceptrons, layers inside CNNs are, by definition, not all dense, fully-connected layers. CNNs introduce the concept of different types of sparsely-connected computational layers. The classical CNN comprises a variable number, C, of convolutional layers stacked upon the input vector, followed by a tail of D, dense layers, which output the result of the network. This gives a total of $N = C + D$ layers, omitting any infrastructure layers that may also be present, such as a flattening layer, which is often employed between the last convolutional layer and the first dense layer, because convolutional layers inherently have multidimensional outputs and dense layers do not. Purely convolutional networks, which consist only of convolutional layers, are possible, but these are a more unusual configuration, especially for classification tasks. Purely convolutional networks appear more often as autoencoders and in situations where you want to lessen the black-box effects of dense layers. Convolutional layers are often more interpretable than pure dense layers as they produce feature maps that retain the input vector's dimensionality.

Convolutional layers can and often do appear as layers in more complex model architectures, which are not necessarily always feed-forward models. They can appear in autoencoders, generative adversarial networks, recurrent neural networks, and as part of attention-based architectures such as transformers and generative diffusion models. We will, for now, consider only the classical design: several convolutional layers capped by several dense ones.

As discussed, CNNs have a more specialised architecture than dense layers. This architecture is designed to help the network perform in a specific domain of tasks by adding a priori information defin-

ing information flow inside the network. This can help reduce overfitting in some cases, as it means a smaller network with fewer parameters can achieve the same task as a more extensive dense network. Fewer parameters mean less total information can be stored in the network, so it is less likely that a model can memorise specific information about the noise present in training examples. A CNN encodes information about the dimensionality of the input image; the location of features within the input image is conserved as it moves through layers. It also utilises the fact that within some forms of data, the same feature is likely to appear at different locations within the input vector; therefore, parameters trained to recognise features can be reused across neurons. For example, if detecting images of cats, cat's ears are not always going to be in the same location within the image. However, the same pattern of parameters would be equally helpful for detecting ears wherever it is in the network.

The following subsections describe different aspects of CNNs, including a description of pooling layers, which are companion layers often employed within convolutional networks.

4.4.1 Convolutional Layers

CNNs take inspiration from the biological visual cortex. In animal vision systems, each cortical neuron is not connected to every photoreceptor in the eye; instead, they are connected to a subset of receptors clustered near each other on the 2D surface of the retina. This connection area is known as the **receptive field**, a piece of terminology often borrowed when discussing CNNs.

Convolutional Layers behave in a similar manner. Instead of each neuron in every layer being connected to every neuron in the previous layer, they are only connected to a subset, and the parameters of each neuron are repeated across the image, significantly reducing the number of model parameters and allowing for translation equivariant feature detection. It is a common misnomer that convolutional layers are translation invariant; this is untrue, as features can and usually do move by values which are not whole pixel widths, meaning that even if the filters are the same, the pixel values can be different and give different results. One common problem with CNNs is that very small changes in input pixel values can lead to wildly different results, so this effect should be mitigated if possible. If they do not involve subsampling, however, CNNs are sometimes equivariant. This means that independent of starting location, ignoring edge effects, if you shift the feature by the same value, the output map will be the same — this can be true for some configurations of CNN, but is also broken by most common architectures.

This input element subset is nominally clustered spacially, usually into squares of input pixels. This means that unlike with dense input layers, wherein 2D and greater images must first be flattened before being ingested, the dimensionality of the input is inherently present in the layer output. In a dense layer, each input is equally important to each neuron. There is no distinguishing between inputs far away from that neuron and inputs closer to that neuron (other than distinctions that the network may learn during the training process). This is not the case inside convolutional layers, as a neuron on a subsequent layer only sees inputs inside its receptive field.

As the proximity of inputs to a neuron can be described in multiple dimensions equal to that of the input dimensionality, the network, therefore, has inherent dimensionality baked into its architecture — which is one example of how the CNN is specialised for image recognition. In the case of a 2D image classification problem, we now treat the input vector as 2D, with the receptive field of each neuron occupying some shape, most simply a square or other rectangle, on the 2D vector's surface.

The term receptive field is usually reserved to describe how much of the input image can influence the output of a particular neuron in the network. The set of tunable parameters which define the computation of a neuron in a convolutional layer when fed with a subset of neuron outputs or input vector values from the previous layer is called a **kernel**. Each kernel looks at a subset of the previous layers' output and produces an output value dependent on the learned kernel parameters. A kernel with pa-

rameters tuned by model training is sometimes called a **filter**, as, in theory, it filters the input for a specific translation-invariant feature (although, as we have said, this is only partially true). The filter produces a strong output if it detects that feature and a weak output in its absence. Identical copies of this kernel will be tiled across the previous layer to create a new image with the same dimensionality as the input vector, i.e. kernels in a time-series classifier will each produce their own 1D time-series feature map, and kernels fed a 2D image will each produce a 2D image feature map. In this way, each kernel produces its own feature map where highly scoring pixels indicate the presence of whatever feature they have been trained to identify, and low-scoring ones indicate a lack thereof. Because the network only needs to learn parameters for this single kernel, which can be much smaller than the whole image and only the size of the feature it recognises, the number of trainable parameters required can be significantly reduced, decreasing training time, memory consumption, and overfitting risk. For a single kernel with no stride or dilation, see Section 4.4.2, applied to an input vector with no depth dimension, the number of trainable parameters is given by

$$\text{len}(\theta_{\text{kernel}}) = \left(\prod_{i}^{N} S_i \right) + 1 \tag{75}$$

where $\text{len}(\theta_{\text{kernel}})$ is the number of trainable parameters in the kernel, N is the number of dimensions in the input vector, and S_i it the configurable hyperparameter, kernel size in the i^{th} dimension. The extra plus one results from the bias of the convolutional kernel.

For example, a 1D kernel of size 3, would have $3 + 1 = 4$ total parameters, independent of the size of the input vector, and a 2D kernel of size 3×3 would have $3 \times 3 + 1 = 10$ total parameters, again independent of the size of the 2D input vector in either dimension. See Figure 36 for an illustration of the structure of a convolutional kernel.

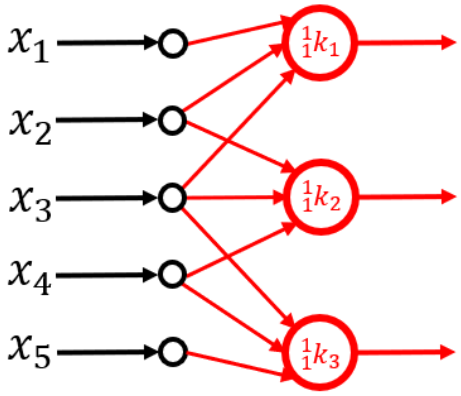


Figure 36 | Diagram of a single kernel, ^1_1k, in a single convolutional layer. In this example, a 1D vector is being input; therefore, the single kernel's output is also 1D. This kernel has a kernel size of three, meaning that each neuron receives three input values from the layer's input vector, $\vec{x}$, which in this case is length five. This means there is room for three repeats of the kernel. Its parameters are identical for each iteration of ^1_1k at a different position. This means that if a pattern of inputs recognised by the kernel at position 1, $^1_1k_1$ is translated two elements down the input vector, it will be recognised similarly by the kernel at $^1_1k_3$. Although this translational invariance is only strict if the translation is a whole pixel multiple and no subsampling (pooling, stride, or dilation) is used in your network, this pseudo-translational invariance can be useful, as often, in images and time series data, similar features can appear at different spatial or temporal locations within the data. For example, in a speech classification model, a word said at the start of the time series can be recognised just as easily by the same pattern of parameters if that word is said at the end of the time series (supposing it lies on the

sample pixel multiple). Thus, the same kernel parameters and the same filter can be repeated across the time series, reducing the number of parameters needed to train the model. This particular kernel would have $3 + 1 = 4$ total parameters, as it applied to a 1D input vector, and has a kernel size of three, with an additional parameter for the neuron bias. With only a single kernel, only one feature can be learned, which would not be useful in all but the most simple cases. Thus, multiple kernels are often used, each of which can learn its own filter.

When first reading about convolutional layers, it can be confusing to understand how they each "choose" which features to recognise. What should be understood is that this is not a manual process; there is no user input on which kernels filter which features; instead, this is all tuned by your chosen optimiser during the training process. Even the idea that each kernel will cleanly learn one feature type is an idealised simplification of what can happen during training. Gradient descent has no elegant ideas of how it should and should not use the architectures presented to it and will invariably follow the path of least resistance, which can sometimes result in strange and unorthodox uses of neural structures. The more complex and non-linear the recognition task, the more often this will occour.

Although we do not specify exactly which features each kernel should learn, there are several hyper-parameters that we must fix for each convolutional layer before the start of training. We must set a kernel (or filter) size for each dimension of the input vector. For a 1D input vector, we will set one kernel size per kernel; for a 2D input vector, we must set two, and so on. These kernel dimensions dictate the number of input values read by each kernel in the layer and are nominally consistent across all kernels in that layer; see Figure 37 for an illustration of how different kernel sizes tile across a 2D input.

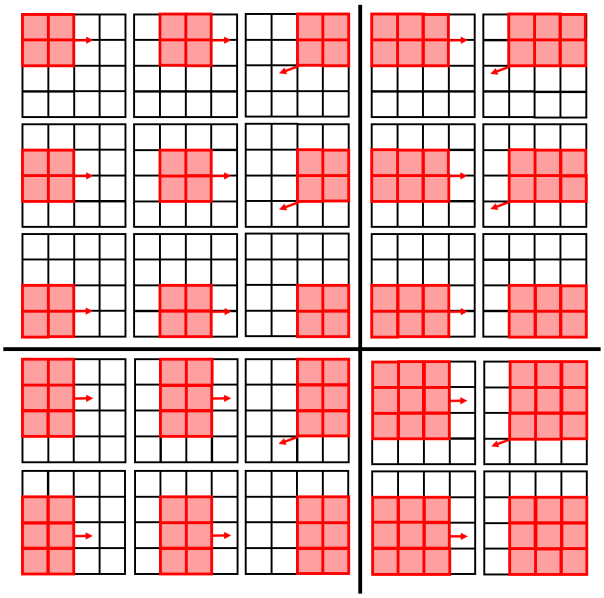


Figure 37 | Illustration of how different values of kernel size would be laid out on a 4×4 input image. In each case, unused input image values are shown as empty black squares on the grid, and input values read by the kernel are filled red. The grids show the input combinations that a single kernel would ingest if it has a given size, assuming a stride value of one and zero dilation. The kernel sizes are as follows: *Upper left:* 2×2. *Upper right:* 3×2. *Lower left:* 2×3. *Lower right:* 3×3. One pixel in the output map is produced for each kernel position. As can be seen, the size of the output map produced by the kernel depends both on the input size and the kernel size; smaller kernels produce a larger output vector.

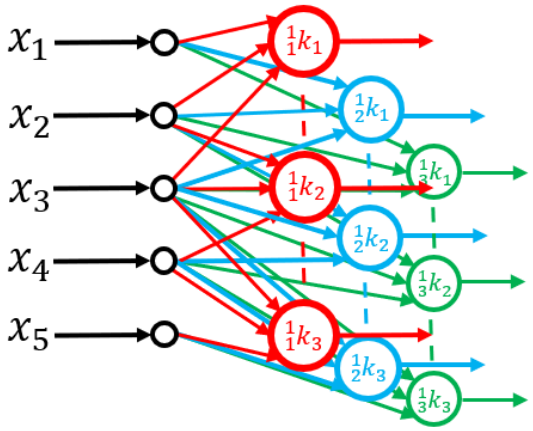
The other hyperparameters that must be set are the number of different kernels and the choice of activation function used by the kernel's neurons. These hyperparameters can sometimes be manually tuned using information about the dataset, i.e. the average size of the features for kernel size and the number of features for the number of kernels, but these can also be optimised by hyperparameter optimisation methods, which might be preferable as is often difficult to gauge which values will work optimally for a particular problem.

Multiple kernels can exist up to an arbitrary amount inside a single convolutional layer. The intuition behind this multitude is simply that input data can contain multiple different types of features, which can each need a different filter to recognise; each kernel produces its own feature map as it is tiled across its input, and these feature maps are concatenated along an extra **depth** dimension on top of the dimensionality of the input vector. A 1D input vector will have 2D convolutional layer outputs, and a 2D input vector will result in 3D convolutional outputs. The original dimensions of the input vector remain intact, whilst the extra discrete depth dimension represents different features of the image; see Figure 38.

In the case of a colour picture, this depth dimension could be the red, green, and blue channels, meaning this dimension is already present in the input vector. The number of trainable parameters of a single convolutional layer is given by

$$\text{len}\left(\theta_{\text{conv\_layer}}\right) = K \times \left(\left(D \times \prod_{i}^{N} S_i\right) + 1\right) \tag{76}$$

where $\text{len}\left(\theta_{\text{conv\_layer}}\right)$ is the total number of parameters in a convolutional layer, K is the number of convolutional kernels in that layer, a tunable hyperparameter, and D is the additional feature depth dimension of the layer input vector, which is determined either by the number pre-existing feature channels in the input vector, i.e. the colour channels in a full-colour image or, if the layer input is a previous convolutional layer, the number of feature maps output by that previous layer, which is equivalent to the number of kernels in the previous layer. For example, a 1D convolutional layer with three kernels, each with size three, ingesting a 1D input with only a singleton depth dimension would have $3 \times ((1 \times (3)) + 1) = 12$ total trainable parameters, whereas a 2D convolutional layer with three kernels of size 3×3 looking at a colour RGB input image would have $3 \times (3 \times (3 \times 3) + 1) = 84$ total trainable parameters.



Type	Num Filters / Neurons	Filter / Pool Size	Stride	Activation Function	Dropout
Convolutional	3	3	1	ReLU	-

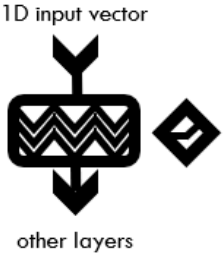
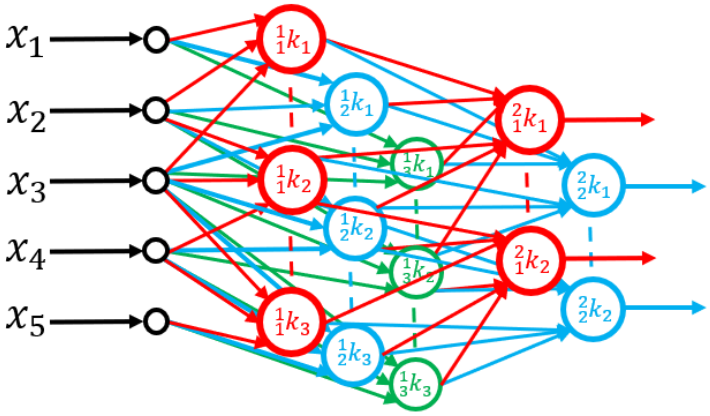


Figure 38 | *Upper:* Diagram of three convolutional kernels, $\left[{}^{1}_{1}k, {}^{1}_{2}k, {}^{1}_{3}k\right]$, in a single convolutional layer. Each kernel is coloured differently, in red, green, and blue. Artificial neurons of the same colour will share the same learned parameters. Again, a 1D vector is being input; therefore, the output of each of the kernels is 1D, and the output of the kernels stack to form a 2D output vector, with one spatial dimension retained from the input vector and an extra discrete depth dimension representing the different features learned by each of the kernels. Again, each kernel has a kernel size of three. Multiple kernels allow the layer to learn multiple features, each of which can be translated across the input vector, as with the single kernel. Using Equation 76, this layer would have $3 \times ((1 \times 3) + 1) = 12$ trainable parameters. It should be noted that this is a very small example simplified for visual clarity; real convolutional networks can have inputs many hundreds or thousands of elements long and thus will have many more iterations of each kernel, as well as many more kernels sometimes of a much larger size. *Lower:* Abstracted diagram of the same layer with included hyperparameter information.

As with dense layers, multiple convolutional layers can be stacked to increase the possible range of computation available; see Figure 39. The first convolutional layer in a network will ingest the input vector, but subsequent layers can ingest the output of previous convolutional layers, with kernels slicing through and ingesting the entirety of the depth dimension. In theory, this stacking allows the convolutional layers to combine multiple more straightforward features in order to recognise more complex, higher-level features of the input data — although, as usual, things are not always quite so straightforward in practice. When calculating the number of trainable parameters in multiple convolutional layers, we can use Equation 76 for each layer and sum the result.



Type	Num Filters / Neurons	Filter / Pool Size	Stride	Activation Function	Dropout
Convolutional	3	3	1	ReLU	-
Convolutional	2	2	1	ReLU	-

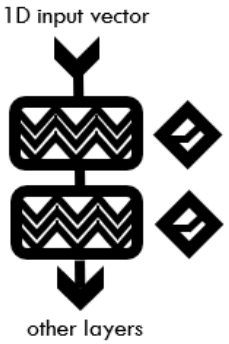
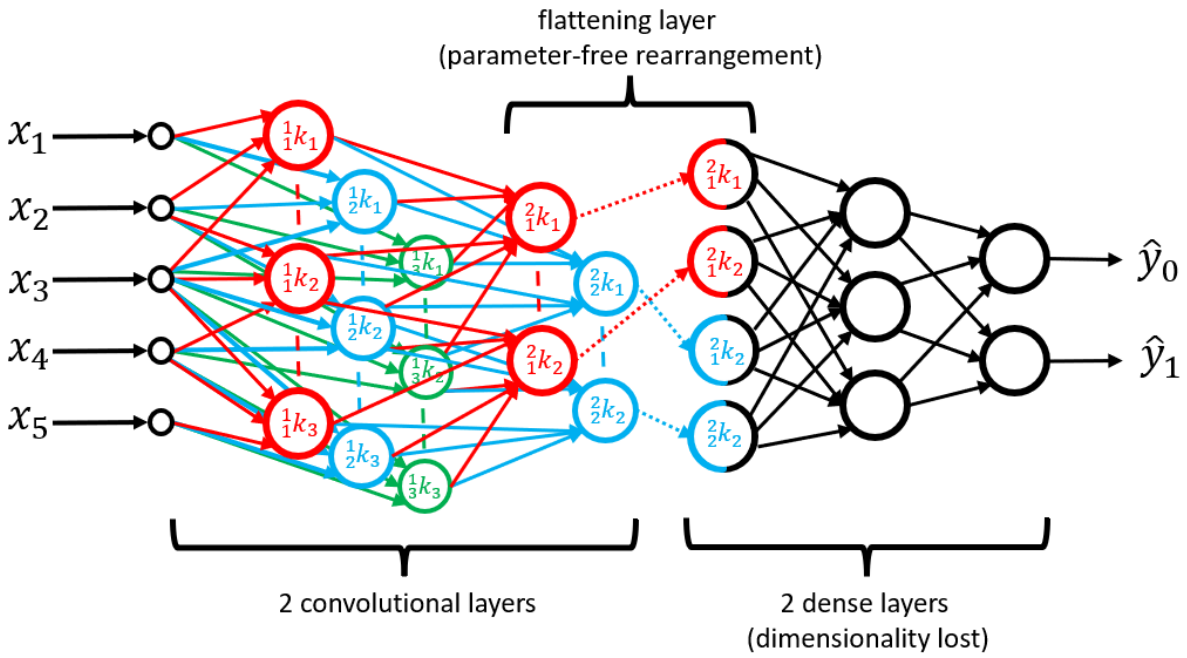


Figure 39 | *Upper:* Diagram of two convolutional layers, each with independent kernels. The first layer has three kernels, each with a size of three. The second layer has two kernels, both with a size of two. Again, this is a much-simplified example which would probably not have much practical use. Different kernels are coloured differently, in red, green, and blue. Although it should be noted that similar colours across layers should not be taken as any relationship between kernels in different layers, they are each tuned independently and subject to the whims of the gradient descent process. This example shows how the kernels in the second layer take inputs across the entire depth of the first layer but behave similarly along the original dimension of the input vector. In theory, the deeper layer can learn to recognise composite features made from combinations of features previously recognised by the layers below and visible in the output feature maps of the different kernels. This multi-layer network slice would have $(3 \times ((1 \times 3) + 1)) + (2 \times ((3 \times 2) + 1)) = 26$ total trainable parameters. This was calculated by applying Equation 76 to each layer. *Lower:* Abstracted diagram of the same layers with included hyperparameter information.

The result of using one or more convolutional layers on an input vector is an output vector with an extra discrete depth dimension, with each layer in the stack representing feature maps. Whilst often considerably more interpretable than maps of the parameters in dense layers, these maps are often not very useful alone. However, a flattened version of this vector is now, hopefully, much easier for dense layers to classify than the original image. As such, CNNs used for classification are almost always capped by one or more dense layers in order to produce the final classification result; see Figure 40 for a toy example of a CNN used for binary classification.



Type	Num Filters / Neurons	Filter / Pool Size	Stride	Activation Function	Dropout
Convolutional	3	3	1	ReLU	-
Convolutional	2	2	1	ReLU	-
Dense	3	-	-	ReLU	-
Dense	2	-	-	SoftMax	-

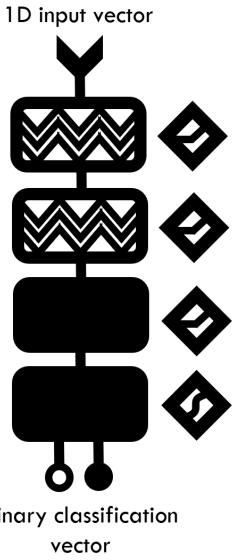


Figure 40 | *Upper:* Diagram of a very simple convolutional neural network binary classifier consisting of four layers with tunable parameters plus one infrastructure layer without parameters. Two subsequent convolutional layers ingest the five-element input vector, $\vec{x}$. The 2D output of the latter of the two layers is flattened into a 1D vector by a flattening layer. This flattened vector is then ingested by two dense layers, the latter of which outputs the final classification score. The first convolutional layer has three convolutional kernels, each with a size of three, and the second convolutional layer has two kernels, both with a size of two. The first dense layer has three artificial neurons, and the final output dense layer has a number of neurons dictated by the required size of the output vector. In the case of binary classification, this is either one or two. Different kernels within a layer are differentiated by colour, in this case, red, green, or blue, but a similar colour between layers does not indicate any relationship. Dimensionless neurons are shown in black; it should be noted that after flattening, dimensional information is no longer necessarily maintained by the network structure. Of course, no information is necessarily lost either, as the neuron index itself contains information about where it originated, so, during training, this information can still be used by the dense layers;

it is just not necessarily maintained as it is in convolutional layers. This network will have in total $26 + (3 \times 4 + 4) + (2 \times 3 + 2) = 50$ trainable parameters. This network is very simple and would probably not have much practical use in real-world problems other than straightforward tasks that would probably not necessitate using neural networks. *Lower:* Abstracted diagram of the same model with included hyperparameter information.

4.4.2 Stride, Dilation, and Padding

Stride is a user-defined hyperparameter of convolutional layers that must be defined before training. By default, this will be set to one and often remains this way. Like kernel size, it is a multidimensional parameter with a value for each input vector dimension. A convolutional layer's stride describes the distance the kernel moves between instances. For example, if the stride is one, then a kernel is tiled with a separation of one input value from its last location. Stride, S, is always greater than zero, $S > 0$. The kernels will overlap in the i^{th} dimension if $S_i < k_i$. If $S_i = k_i$, there will be no overlap and no missed input vector values. If $S_i > k_i$, some input vector values will be skipped; this is not usually used. Along with kernel size, stride determines the output size of the layer. A larger stride will result in fewer kernels and, thus, a smaller output size; see Figure 41 below for an illustration of different kernels strides.

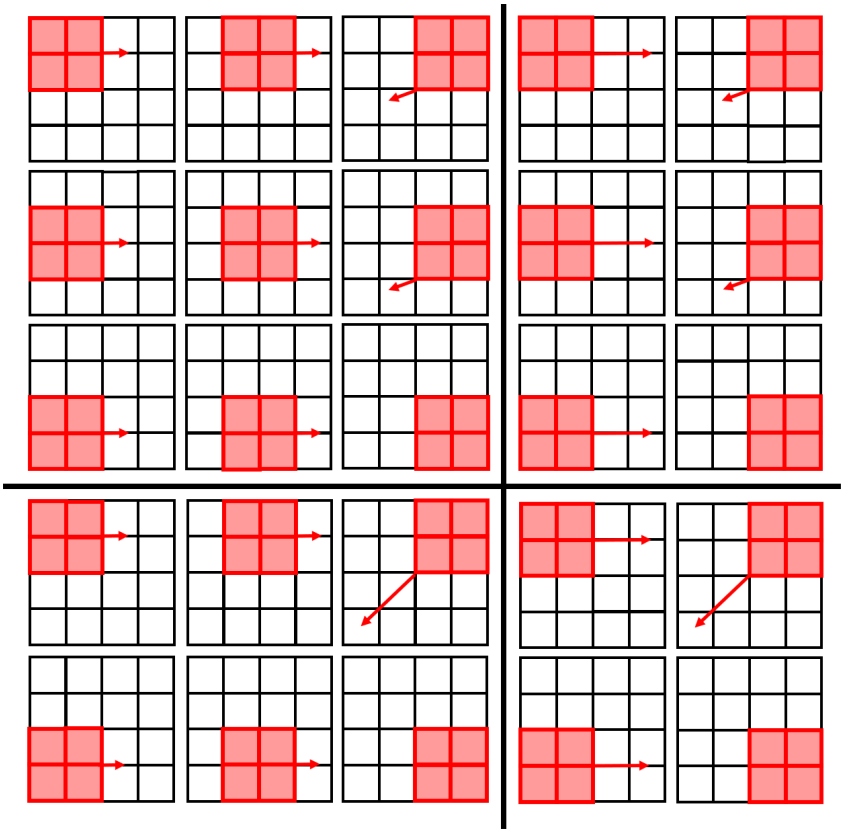


Figure 41 | Illustration of how different values of kernel stride would be laid out on a 4×4 input image. In each case, unused input image values are shown as empty black squares on the grid, and input values read by the kernel are filled in red. Similar to kernel size, different values of stride result in a different output vector size. The strides shown are as follows: *Upper left:* $2, 2$. *Upper right:* $3, 2$. *Lower left:* $2, 3$. *Lower right:* $3, 3$.

Introducing kernel stride primarily serves to reduce the overall size of your network by reducing the output vector without adding additional parameters; in fact, the number of parameters is independent

of stride. Reducing the size of your network might be a desirable outcome as it can help reduce computational time and memory overhead. It can also help to increase the receptive field of neurons in subsequent layers as it condenses the distance between spatially separated points, so if you're playing with the resolution of feature maps in your model to balance the identification of smaller and larger scale features, it could potentially be a useful dial to tune. In most cases, however, it's left at its default value of one, with the job of reducing the network size falling to pooling layers; see Section 4.4.3.

One interesting and potentially unwanted effect of introducing stride into our network is that it removes the complete translation equivariance of the layer by subsampling; instead, translations are only equivariant if they match the stride size, i.e. if a kernel has a stride of two features are invariant if they move exactly two pixels, which is not a common occurrence.

Dilation is a further hyperparameter that can be adjusted prior to network training; by default, this value would be set to zero, and no dilation would be present. Dilation introduces a spacing inside the kernel so that each input value examined by the kernel is no longer directly adjacent to another kernel input value, but instead, there is a gap wherein the kernel ignores that element. This directly increases the receptive field of that kernel without introducing additional parameters, which can be used to help the filters take more global features into account. It can also be used in the network to try and combat scale differences in features; if multiple kernels with different dilations are used in parallel on different model branches, the model can learn to recognise features at the same scale but with different dilations; see Figure 42.

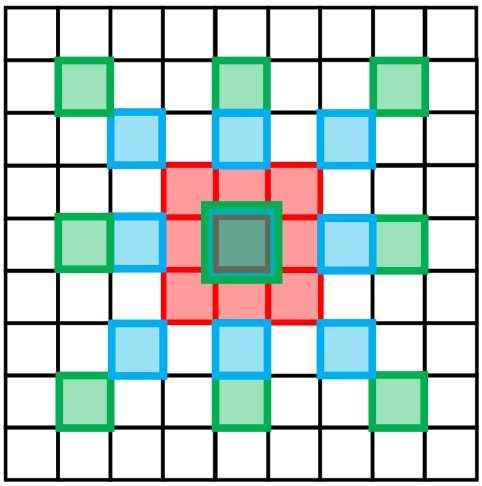


Figure 42 | Diagram illustrating how different values of kernel dilation affect the arrangement of the kernel input pixels. In this example, the receptive field of a single 3×3 kernel at three different dilation levels is displayed; differing colours represent the input elements at each dilation level. The shaded red kernel illustrates dilation level zero; the shaded blue region is a kernel with dilation of one, and the green kernel has a kernel dilation of two.

Particular stride, dilation, and size combinations will sometimes produce kernel positions that push them off the edge of the boundaries of the input vector. These kernel positions can be ignored, or the input vector can be padded with zeros or repeats of the nearest input value; see Figure 43.

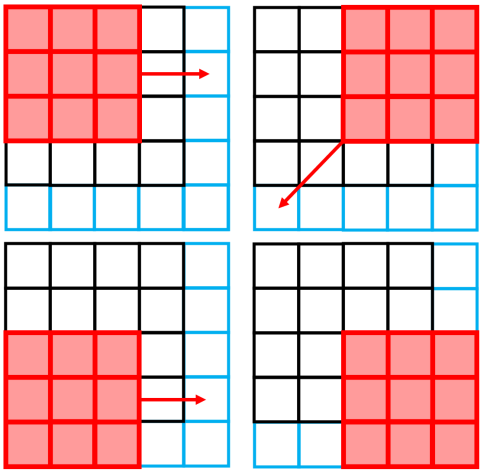


Figure 43 | Diagram illustrating how padding can be added to the edge of an input vector in order to allow for otherwise impossible combinations of kernel, stride, size, and dilation. In each case, unused input image values are shown as empty black squares on the grid, input values read by the kernel are shaded red, and empty blue squares are unused values added to the original input vector, containing either zeros or repeats of the closest data values. In this example, the kernel size is 3×3, and the kernel stride is $2, 2$.

4.4.3 Pooling

Pooling layers, or simply pooling, is a method used to restrict the number of data channels flowing through the network. They see widespread application across the literature and have multiple valuable properties. They can reduce the size of your network and thus the computation and memory overhead, and they can also make your network more robust to small translational, scale, and rotational differences in your network. Convolutional layers record the position of the feature they recognise but can sometimes be overly sensitive to tiny shifts in the values of input pixels. Small changes in a feature's scale, rotation or position within the input can lead to a very different output, which is evidently not often desirable behaviour.

Pooling layers do not have any trainable parameters, and their operation is dictated entirely by the user-selected hyperparameters chosen before the commencement of model training. Instead, they act to group together pixels via subsampling throwing away excess information by combining their values into a single output. In this way, they are similar to convolutional kernels, however. instead of operating with trained parameters, they use simple operations. The two most common types of pooling layers are **max pooling** and **average pooling**; max pooling keeps only the maximum value within each of its input bins, discarding the other values; intuitively, we can think of this as finding the strongest evidence for the presence of the feature within the pooling bin and discarding the rest. Average pooling averages the value across the elements inside each pooling bin, which has the advantage that it uses some information from all the elements.

As can be imagined, the size of CNNs can increase rapidly as more layers and large numbers of kernels are used, with each kernel producing a feature map nearly as large as its input vector. Although the number of parameters is minimised, the number of operations increases with increasing input size. Pooling layers are helpful to reduce redundant information and drastically reduce network size whilst also making the network more robust to small changes in the input values.

Along with the choice of operational mode, i.e. average or maximum, pooling layers have some of the same hyperparameters as convolutional kernels, size and stride. Unlike convolutional layers, the pooling stride is usually set to the same value as the pooling size. Meaning that there will be no overlap

between pooling bins but also no gaps. This is due to the purpose of pooling layers, which attempt to reduce redundant information; if stride were set to smaller values, there would be little reduction and little point to the layer.

Because pooling with stride is a form of subsampling, it does not maintain strict translational equivariance unless the pool stride is one, which, as stated, is uncommon. Thus, as most CNN models use pooling, most CNNs are neither strictly translationally invariant nor equivariant.

4.5 Results from the Literature

4.5.1 CBC Review

CNNs are introduced here as a thorough literature review of other attempts to solve the classification problem would be prudent before we continue. Almost all attempts from the literature involve models which are at least as complex as CNNs. Hence, an understanding of their operation is crucial to gaining an effective overview of the current state of the field. This is not intended to be an exhaustive catalogue, although efforts have been made to be as complete as possible.

Both gravitational-wave astrophysics and deep learning methods have been through rapid advancement in the previous decade, so it is perhaps unsurprising that there has also developed a significant intersection between the two fields. Multiple artificial network architectures, CNNs, autoencoders, generative adversarial networks, recurrent neural networks, and attention-based networks like transformers and generative diffusion models have been applied to numerous gravitational-wave data analysis problems. This review will focus on efforts to apply CNN classifiers to detect features hidden within interferometer data. First, we will look at attempts to detect Compact Binary Coalescences (CBCs), followed by a look at unmodeled Bursts detection attempts. More complex network architectures will be reviewed later when we examine attention layers in closer detail; see Section 6.

The earliest attempts at CBC classification using artificial neural networks were a pair of independently published papers by George *et al.* [90] and Gabbard *et al.* [91]. George *et al.* [90] applied CNNs to the binary classification problem and basic parameter estimation. They used CNNs with two outputs to extract parameter estimates for the two companion masses of the binary system. They used the whitened outputs of single interferometers as inputs and utilised CNNs of a standard form consisting of convolutional, dense, and pooling layers. They evaluated two models, one smaller and one larger. In their first paper, they used only simulated noise, but they produced a follow-up paper showing the result of the model's application to real interferometer noise [92]. Gabbard *et al.* [91] used an alternate CNN design with a different combination of layers. They only used a single network architecture, and no attempt at parameter estimation was made. A differentiating feature of their paper was the training of individual network instances to recognise different SNRs. Both George *et al.* [90] and Gabbard *et al.* [91] achieved efficiency curves that closely resembled that of matched filtering; of note, however, both were validated at a considerably higher FAR than is usually allowed in a production search, this will be a consistent theme throughout the literature and is one of the greatest blockers to using CNNs in an official search pipeline.

There have been many papers that follow up on these two initial attempts. Several papers with mixed results are hampered by inconsistencies and unclear methodology. Luo *et al.* [93] attempted to improve the model described by Gabbard *et al.* They have presented their results using a non-standard "Gaussian noise amplitude parameter" and in the form of Receiver Operator Curves (ROC) on a linear rather than a log scale. Whilst within their own comparisons, they seem to have improved network operation over the original design, at least at higher FARs, it is difficult to make a comparison against other papers because of the unorthodox presentation. Schmitt *et al.* [94] attempted to compare the performance of one of the models presented in George *et al.* [90] with three different model architectures, Temporal Convolutional Networks (TCNs), Gated Recurrent Units (GRUs), and Long Short-Term Memory

(LSTMs). However, the paper has a strange use of SNR, which is not described in detail; they used negative SNRs, which would be non-physical if it were describing optimal SNRs along with SNRs of zero, which should only create entirely zero-valued waveforms. They seem to show that the other model architectures can achieve higher performance than CNNs, but without a known waveform scaling, it is hard to compare to other results.

A more interesting follow-up by Fan *et al.* [95] took the smaller of the two models introduced in George *et al.* [90] and extended it to use multiple detectors as inputs rather than the previously mentioned studies, which looked at only single detectors. They do this for both detection and parameter estimation and appear to show improved accuracy results over the original paper [90], although they do not address the confounding factor of having to deal with real noise. Krastev *et al.* tested the use of the other larger model introduced by George *et al.* [90]. They tested its use on Binary Neuron Star (BNS) signals, as well as reaffirming its ability to detect BBH signals. They used significantly longer input windows to account for the longer detectable duration of BNS signals. They found BNS detection to be possible, although it proved a significantly harder problem.

Using a different style of architecture, Gebhard *et al.* [96] argued that convolution-only structures are more robust and less prone to error, as they remove much of the black-box effect produced by dense layers and allow for multiple independently operating (though with overlapping input regions) networks, creating an ensemble which generates a predictive score for the presence of a signal at multiple time positions. This results in a time-series output rather than a single value, which allows the model to be agnostic to signal length. Their determination of the presence of a signal can thus rely on the overall output time series rather than just a single classification score. Similarly to Fan *et al.* [95], they used multiple detector inputs. Whilst this is interesting work, they give their results at different values of False Positive Rate (FPR), the probability that a given positive result is erroneous, rather than False Alarm Rate (FAR), the probability that a given duration of noise with no signal will produce a positive result, so again it is hard to make a direct comparison to other studies.

There have been at least two papers which utilise ensemble approaches to the problem. Ensembles consist of multiple independently trained models in the hopes that the strengths of another will counteract the weaknesses of one under the assumption that it is less likely for them both to be weak in the same area. A joint decision is then taken through some mechanism that takes the result of all models into consideration, often waiting for certain models' votes under certain criteria. Huerta *et al.* [97] used an approach consisting of four independently trained models, each of which has two separate CNN branches for the LIGO Hanford and LIGO Livingston detectors, which are then merged by two further CNN layers. Their efficiency results are presented on a logarithmic ROC, which gives more clarity than a linear one, but their results are still clustered tightly in one corner, making them hard to parse. Still, they have efficiency results down to a lower FAR than any paper reviewed so far, at 1×10^{-5}, which is impressive, although the efficiency scores at these FARs are low ($< 1\%$). Overall, the paper is more focused on the software infrastructure for deploying neural network models. Ma *et al.* [98] used an ensemble network that employ one of the architectures described by Gabbard *et al.* [91]. They utilise two "subensembles" in an arrangement in which each detector has its own ensemble composed of networks which vote on a false/positive determination; the results of both of the two subensembles are then combined for a final output score. They do not give efficiency scores at set SNRs, so again, it is difficult to compare against other results.

There have also been some interesting studies which use feature engineering to extract features from the input data before those features are fed into the CNN models, see Section 4.2.7. Wang *et al.* [99] use a sparse matched filter search, where template banks of only tens of features, rather than the usual hundreds of thousands or millions, were used. The output of this sparce matched filter was then ingested by a small CNN, which attempted to classify the inputs. Notably, they use real noise from

the 1st LVK joint observing run and multi-detector inputs. Though an interesting method, their results appear uncompetitive with other approaches. Reza [100] *et al.* used a similar approach but split the input into patches before applying the matched filter. However, results are not presented in an easily comparable fashion. Bresten *et al.* [101] adapts one of the architectures from George *et al.* [90] but applies a feature extraction step that uses a topological method known as persistent homology before the data is ingested by the network. It is an interesting approach, but their results are unconvincing. They limited their validation data to 1500 waveforms at only 100 specific SNR values in what they term their "hardest case". They showed poor results compared to other methods, suggesting their method is undeveloped and heavily SNR-tailored.

There have been at least three spectrogram-based attempts to solve the CBC detection problem: Yu *et al.* [102] and Aveiro *et al.* [103]. Yu *et al.* used single detector spectrograms, which are first analysed in strips using multiple 1D CNNs before being fed into a 2D CNN for final classification; they achieve middle-of-the-range efficiency results. Aveiro *et al.* [103] focused on BNS detection and used an out-of-the-box object detection network to try and detect patterns in spectrograms. They do not state efficiencies for SNRs less than ten. Finally, there was also a search paper [104], which searched through the second observing run using spectrograms-based CNNs; they detected nothing of significance.

There has also been an attempt to use wavelet decomposition for the problem. Lin *et al.* [105] focused on the detection of BNS signals by wavelet decomposition with some very promising results shown to outperform matched filtering; a subsequent follow-up paper [106] showed that the same method could be applied to BBH signals with equal promise. They achieve an efficiency of 94% when detecting waveforms with an SNR of 2 at a FAR of 1×10^{-3}, which undercuts the competition by considerable margins. Their method is certainly worth investigation but was unfortunately missed until this thesis was in the latter stages of construction, so no wavelet decomposition methods have been attempted.

There have also been a number of papers utilising CNNs for specialised detection cases, such as mass asymmetric CBCs [107] by Andrés-Carcasona *et al.*, who employ spectrogram-based CNNs to run a search over O3, and eccentric CBCs by Wei *et al.* [108], the latter of which also focuses on early detection along with a few other papers [109], [110], [111] which attempt to detect CBCs signals before the inspiral proper. There have also been a number of papers which discuss the use of CNNs for the analysis of data from future space-based detectors [112], [113] . For brevity, and as they are less relevant to our problems, these special cases will not be discussed here.

As can be seen, it is very difficult to compare the performance of many of the architectures and methods presented in the literature. The results are presented at wildly different FARs and SNR ranges, often using different incomparable metrics and with varying levels of rigour. There is a tendency to apply new tools and ideas to the problem without careful thought about how the results can be standardised. Table 5 displays results from some of the papers which were found to have at least somewhat comparable metrics.

Name	Conv	Pool	Dense	Real Noise?	Detectors	Target	Feature	SNR Tailored	FAR	Acc 8	Acc 6	Acc 4
George *et al.* [90]	0, 2, 4	1, 3, 5	6, 7	No	Single	BBH	No	No	5×10^{-2}	0.98	0.70	0.16
-	0, 2, 4, 6	1, 3, 5, 7	8, 9, 10	-	-	-	-	-	-	0.99	0.80	0.21
George *et al.* [92]	-	-	-	Yes	-	-	-	-	-	0.98	0.77	0.18
Gabbard *et al.* [91]	0, 1, 3, 4, 6, 7	2, 5, 8	9, 10, 11	No	Single	BBH	No	Yes	1×10^{-1}	1.0	0.88	0.44
-	-	-	-	-	-	-	-	-	1×10^{-2}	0.99	0.69	0.10
-	-	-	-	-	-	-	-	-	1×10^{-3}	0.98	0.49	0.02
Fan *et al.* [95]	0, 2, 4	1, 3, 5	6, 7	No	Three	BBH	No	No	4×10^{-2}	0.99	0.84	0.32
Krastev *et al.* [114]	0, 2, 4, 6	1, 3, 5, 7	8, 9, 10	No	Single	BNS	No	No	1×10^{-1}	0.71	0.42	0.20
•	-	-	-	-	-	-	-	-	1×10^{-2}	0.32	0.10	0.02

•	-	-	-	-	-	-	-	-	1×10^{-3}	0.11	0.00	0.00
Gebhard *et al.* [96]	**Conv layers $\times$ 12**	None	None	**Yes**	**Two**	BBH	No	No	800 (IFPR)	0.83	0.35	Not Given
-	-	-	-	-	-	-	-	-	450 (IFPR)	0.80	0.32	Not Given
-	-	-	-	-	-	-	-	-	100 (IFPR)	0.67	0.23	Not Given
Wang *et al.* [99]	0, 2	1, 3	4, 5	**Yes**	**Two**	BBH	**Matched Filter**	No	1×10^{-1}	0.60	0.24	0.12
-	-	-	-	-	-	-	-	-	1×10^{-2}	0.30	0.05	0.00
-	-	-	-	-	-	-	-	-	1×10^{-3}	0.08	0.00	0.00
Huerta *et al.* [97]	**Ensemble $\times$ 4**	None	-	Yes	Two	BBH	No	No	**5×10^{-4}**	0.20	0.15	Not Given
-	-	-	-	-	-	-	-	-	**5×10^{-5}**	0.01	0.001	Not Given
Yu *et al.* [102]	$2 \times 1D$ CNN $\rightarrow 2D$ CNN	-	-	Yes	Single	BBH	**Spectrogram**	No	6×10^{-2}	0.89	0.67	0.20

Table 5 | Note: Some accuracy values are extracted from plots by eye, so substantive error will have been introduced.

4.5.2 Burst Review

The literature surrounding burst detections with CNNs is, fortunately for the sake of the word count of this document, considerably more limited. In all of the previously mentioned deep-learning studies, the training of the network has relied on accurate models of CBC waveforms. As has been noted, the availability of reliable waveforms for other potential gravitational-wave sources, i.e. bursts, is considerably narrower due to unknown physical processes, large numbers of free parameters, and computational intractability, making it nearly impossible to have a sufficiently sampled template bank.

Despite this, there have been some attempts, most notably using simulated supernovae waveforms, as these are the most likely candidates for initial burst detection. There have been at least five attempts to classify supernovae with this method. Iess *et al.* [115] used a CNN mode with two separate inputs; a 1D time series and a 2D spectrogram were fed into different input branches of the model. They used supernova signals taken from simulation catalogues along with a simple phenomenological model for two transient glitches classes in order to train the CNN to distinguish between the glitches and supernovae in the hopes that if a supernova signal were to appear in future interferometer data, it could be identified as such, rather than being ruled out as a glitch. Perhaps unsurprisingly, due to the complexity of the signal compared to CBCs, they require a significantly higher SNR in order to achieve similar accuracy results as the CBC case, although they still achieve some efficiency at lower SNRs. Chan *et al.*

[116] trained a CNN using simulated core-collapse supernovae signals drawn from several catalogues covering both magnetorotational-driven and neutrino-driven supernovae. They measured the ability to detect the signal and correctly classify which of the two types it fell into. They used moderately deep CNNs and emphasised the importance of multi-detector inputs for the task. They found it possible to detect magnetorotational-driven events at considerably greater distances.

Lopez *et al.* [117], [118] forgoes the use of simulated template backs in their training for a phenomenological approach in an attempt to try and avoid the problem of small template banks. They used an intricate model architecture comprised of mini-inception-resents to detect supernova signals in time-frequency images of LIGO-Virgo data. Mini-inception resnets consist of multiple network branches of different lengths, which run in parallel before combining to produce a final classification score. Having some paths through the network that are shorter than others can be beneficial to avoid the vanishing gradient problem, wherein gradients fall off to zero within the network; having shortcuts allows the network to maintain a clearer view of the inputs even when other paths have become deep, Blocks of layers within networks that have **skip connections** periodically like this are known as residual blocks, and allow much deeper architectures than would otherwise be possible. Networks that employ skip connections are known as **residual networks** or **resnets**. Inception designs have multiple different network branches, all consisting of residual blocks, so there are many paths through the network from input vector to output.

Sasaoka *et al.* [119], [120] use gradient-weighted feature maps to train CNNs to recognise supernovae spectrograms. They utilised core-collapse supernovae waveforms from a number of catalogues. However, they only achieved good classification performances at 1 kpc. They attributed some of their difficulties to features lost to the lower resolution of their time-frequency maps and recommended trying a different algorithm for their generation.

There have also been a few attempts to apply CNNs to the problem of unmodelled signal detection, looking for generic signals using methods which do not require a precisely tailored training set. As has been discussed, we do not yet know how well our simulations will align with real supernovae' gravitational emissions, and it is hard to tell whether the differences between our training datasets and the real signals will significantly hinder our model's ability to detect real signals. Such difficulty could certainly be a possibility; often, deep learning modules can be very sensitive to changes in their

distribution and can lose significant efficacy when applied to out-of-distribution examples. If a sensitive enough generic model could be trained, this would alleviate this problem.

Marianer *et al.* [121] attempt a generic detection method via anomaly recognition. This is not a novel idea in the field of machine learning. However, its application to a generic burst search is intriguing. They apply their model to spectrograms of known transient noise glitches and use a mini-inception resnet to classify the input spectrograms into glitch classes. While examining the feature space of the classifier as it examines data, i.e. the feature maps of the neurons in internal convolutional layers, they utilise two anomaly detection methods to identify when a particular feature space does not look like it belongs to any of the known classes. This means they do not rely directly on the model to output a "novel" class. The latter poses a difficult problem as it is unclear how to ensure the training set is well sampled over every possible counterexample.

The other and most relevant work to this thesis on unmodeled glitch detection is MLy [51]. MLy is a deep learning pipeline that relies on CCN models, which are trained to directly identify coherence between multiple detectors rather than using any pattern recognition or anomaly rejection techniques. This makes it somewhat unique amongst the methods presented. Rather than using a dataset consisting of particular morphologies of signal, MLy utilises distributions of generic white noise burst signals that, in their entirety, will cover all possible burst morphologies with a certain frequency range and duration. One would note that these distributions would also cover all possible glitch morphologies within that parameter space. Therefore, MLy is trained not only to notice the presence of a signal but also the coherence of that signal across detectors. In that sense, it is similar to the operation of many of the preexisting burst pipelines, though it is the only purely machine-learning pipeline to attempt to do this.

MLy achieves this goal by utilising two independent CNN models, one of which looks simply for excess power in both detectors, the coincidence model, and one of which attempts to determine coherence between detections; the second model is fed feature-engineered data in the form of the rolling Pearson correlation between detectors with a number of factor-of-sample-interval timeshifts equivalent to the maximum arrival time difference between the two detectors in question. It does this for the two LIGO detectors and the Virgo detector. It is trained on four types of example: pure noise, noise with a simulated transient (WNB) in one detector only, noise with a simulated transient in all three detectors but with enforced incoherence, and coherent WNBs projected into the three detectors in a physically realistic manner. Using this method, the coherence network can learn to differentiate between coincident glitches and coherent signals.

As baseline models to compare with the results of our improvement attempts, in the next section, we will train five model architectures using the GWFlow data acquisition and training pipeline. Since they are the basis of many of the subsequent attempts at CBC detection, we will train both the models presented in George *et al.* [90] along with the model presented in [91] and for the coherence case, the two models of the MLy pipeline [51].

4.5.3 CBC Detection recreation

Type	Num Filters / Neurons	Filter / Pool Size	Stride	Dilation	Activation Function	Dropout
Convolutional	16	16	1	0	ReLU	-
MaxPooling	-	4	4	0	-	-
Convolutional	8	32	1	4	ReLU	-
MaxPooling	-	4	4	0	-	-
Convolutional	8	64	1	4	ReLU	-
MaxPooling	-	4	4	0	-	-
Dense	64	-	-	-	ReLU	-
Dense	2	-	-	-	SoftMax	-

Type	Num Filters / Neurons	Filter / Pool Size	Stride	Dilation	Activation Function	Dropout
Convolutional	64	16	1	1	ReLU	-
MaxPooling	-	4	4	0	-	-
Convolutional	128	16	1	2	ReLU	-
MaxPooling	-	4	4	0	-	-
Convolutional	256	16	1	2	ReLU	-
MaxPooling	-	4	4	0	-	-
Convolutional	512	32	1	2	ReLU	-
MaxPooling	-	4	4	0	-	-
Dense	128	-	-	-	ReLU	-
Dense	64	-	-	-	ReLU	-
Dense	2	-	-	-	SoftMax	-

Figure 44 | CNN architectures from George *et al.* [90]. *Upper:* Smaller model. *Lower:* Larger model.

Type	Num Filters / Neurons	Filter / Pool Size	Stride	Dilation	Activation Function	Dropout
Convolutional	64	8	1	0	ELU	-
MaxPooling	-	8	8	0	-	-
Convolutional	32	8	1	0	ELU	-
Convolutional	32	16	1	0	ELU	-
MaxPooling	-	6	6	0	-	-
Convolutional	16	16	1	0	ELU	-
Convolutional	16	32	1	0	ELU	-
MaxPooling	-	4	4	0	-	-
Convolutional	16	32	1	0	ELU	-
Dense	64	-	-	-	ELU	0.5
Dense	64	-	-	-	ELU	0.5
Dense	2	-	-	-	SoftMax	-

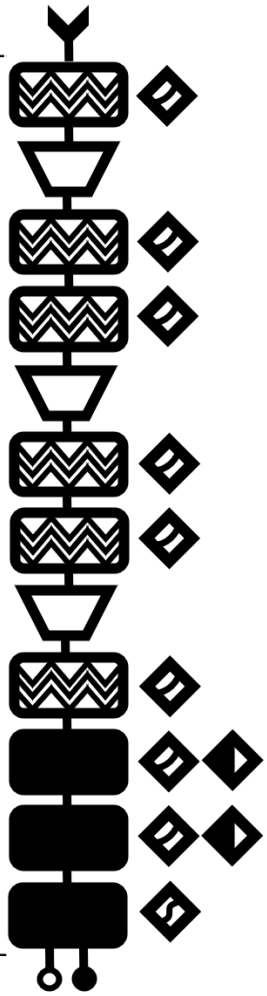


Figure 45 | CNN architecture from Gabbard *et al.* [91].

5 Demystifying Artificial Neural Networks for Gravitational-Wave Analysis: Exploring Hyperparameter Space with Genetic Algorithms

5.1 The Problem with Parameters

An applicable machine-learning approach can be found for almost every problem in gravitational-wave data science. That does not mean that machine learning should be applied to every problem in gravitational-wave data science. We must be careful with a liberal application of machine learning approaches and always keep the goal in mind: what exactly are we trying to achieve by applying this particular method? As described in the "No free lunch theorem", [https://ieeexplore.ieee.org/document/585893] for every possible algorithm, there are advantages and disadvantages and no algorithm that completely supersedes another in all classes. This means a rigorous and systematic method for comparing different techniques is required. This problem is confounded with deep learning techniques, as the number of free parameters when designing and optimising your neural network is vast - technically infinite in the non-real case that network size is not a constraint.

There are a huge number of regularisations that can be applied to a network, and the number of developed layer types and general architectures is considerable and increasing in an almost logarithmic fashion year on year. Even ignoring the number of different types of regularisation, most regularisations have multiple associated parameters, including parameters specifying the design of individual network layers. We label any parameter to do with the model design (aka its regularisation), or that is not optimised during the model training process as a "hyperparameter". Hyperparameters include values to do with the greater structure of the network, such as the type and number of layers in a network, the configuration of the layers themselves, the number of neurons in a dense layer, or the number of filters in the convolutional layer; the training of the network, the learning rate, the number of epochs, and the optimiser, as well as all the parameters associated with the training dataset. Essentially hyperparameters encompass all parameters that must be determined before the initialisation of model training.

This chapter will first give a brief overview of available hyperparameter optimisation methods, then discuss why evolutionary population-based methods were chosen as the hyperparameter optimisation technique for the remainder of this thesis, followed by a demonstration of the effectiveness of hyperparameter optimisation to improve model performance on various gravitational-wave data analysis problems. We will conclude by discussing how this work has been pivotal in developing MLy, a machine learning pipeline currently in review for live deployment in the latter half of the fourth joint observing run.

The goal of any given hyperparameter optimisation process is to maximise the model's performance given a specific objective function. This objective function could be as simple as minimising the model loss, but other performance metrics might also be important to us, such as model inference time or memory usage - or, as is the case for gravitational wave transient detection, minimising values that it would not necessarily make sense to have as part of the loss function, like false alarm rate. If we naively gave models a loss function which only allows a once-in-one-hundred-years false alarm rate, they might never produce a positive result at all. It would be hard to balance such a low false alarm rate requirement with other terms in the loss function, and balancing loss function terms is always a difficult challenge that leads to unstable training. [CITATION AND REPHRASING ARE PROBABLY NEEDED HERE.]

If one is to compare two regularisation techniques, for example, comparing fully connected networks to networks with some convolutional layers, a method must be used to determine all of these hyperparameters. Many, if not most, of these hyperparameters, will have some effect, somewhere between significant and small, on the model's overall performance. Thus the vector space defined by these parameters comprises regions of different model performances, and indeed model performance can be measured in multiple ways. Presumably, given the task at hand, there will be some region within this parameter space that maximises desired performance goals. In the optimal scenario, the comparison of two sets of architectures and hyperparameters will occur between these regions. Thus, a method must find approximate values for these optimal hyperparameters. The reader might now see the recursion that has started here. We are applying an optimisation method to an optimisation that will introduce its own set of hyperparameters. Such hyperparameters will, in turn, need to be at least selected if not optimised. However, it can be shown that the selection of network hyperparameters can make a profound impact [cite] on the performance of the model, and it is hoped that with each optimisation layer, the effects are considerably diminished, meaning that roughly tuned hyperparameters for the hyperparameter optimiser are sufficient for to find comparably optimised solutions.

We can use a similar example parameter space that we generated for Section 3.2, except this time it is being used to represent hyperparameter space against the model-objective function, rather than parameter space against the loss.

Fig: An example parameter space, here represented in 2D. In actuality, the space is likely to have a much larger dimensionality.

Perhaps unsurprisingly, hyperparameter optimisation is an area of considerable investigation and research in machine learning. However, similar to the rest of the field, I think it would be incorrect to call it well-understood. There are several effective methods for hyperparameter optimisation, and there is no universally accepted set of criteria for which method to use for which problems. What follows is a brief and undoubtedly non-comprehensive review of currently available hyperparameter optimisation techniques.

5.1.1 Human-guided trial and error

The most straightforward and obvious method to find effective model hyperparameters relies on human-guided trial and error. This method, as might be expected, involves a human using their prior assumptions about the nature of the problem, the dataset, and the model structure, to roughly guide them towards an acceptable solution, using multiple trials to rule out ineffective combinations and compare the results to the human's hypothesised intuitions. Evidently, whilst this technique is simple to implement and can be time efficient, it suffers from several deficiencies. The results of this method can vary in effectiveness depending on the previous experience of the guiding human; if they have a lot of experience with prior optimisation tasks, they are likely to have more effectively tuned priors. It is also possible that an experienced optimiser might have overly tuned priors, and that bias might cause them to miss possible new solutions that were either previously overlooked or are only relevant to the particular problem being analysed. The results of this method also suffer from a lack of consistency; even the most experienced human optimiser is unlikely to perform precisely the same optimisation technique across multiple problems. Despite these weaknesses, this method is commonly used throughout gravitational wave machine-learning papers and can still be an effective solution for isolated optimisation.

5.1.2 Grid Search

A more methodical approach is to perform a grid search across the entirety or a specified subsection of the available parameter space. In this method, a grid of evenly spaced points is distributed across the selected parameter space - a trial is performed at each grid point, and the performance results of those

trials are then evaluated. Depending on the computing power and time available, this process can be recursed between high-performing points. This method has the advantage of performing a much more rigorous search over the entirety of the parameter space. However, it can be highly computationally expensive if your parameter space has large dimensionality, which is often the case. A grid search can also be ineffective at finding an optimal solution if the objective function is non-linear and highly variable with minor changes or evidently if its solution lies outside of the range of initial boundaries.

Fig: An example of the samples a grid search might use to find an optimal hyperparameter solution.

5.1.3 Random Search

Random search is a very similar method to a grid search; however, instead of selecting grid points evenly spaced across the parameter space, it randomly selects points from the entirety of the parameter space. It has similar advantages and disadvantages to grid search, and with infinite computing resources, both would converge on the ground truth value for the objective function. However, random search has some benefits over grid search that allow it to more efficiently search the parameter space with fewer evaluations. When performing a grid search, the separation of grid points is a user-defined parameter, which both introduces a free parameter and creates possible dimensional bias. A grid search will also search the same value for any given hyperparameter many times, as along the grid axis, it will appear many times, whereas a random search should rarely repeat samples on any hyperparameter. It should also be noted that some statistical uncertainty will be introduced, which would not be present in the case of a grid search and might limit the comparability of different approaches. Both the random and grid search techniques have the disadvantage that all samples are independently drawn, and unless the processes are recursed, no information from the performance results can influence the selection of new points.

Fig: An example of the samples a random search might use to find an optimal hyperparameter solution.

5.1.4 Bayesian Optimisation

A Bayesian optimisation approach makes use of our initial beliefs, priors, about the structure of the objective function. For example, you might expect the objective function to be continuous and that closer points in the parameter space might have similar performance. The objective function is estimated probabilistically across the parameter space. It is updated as more information is gathered by new samples, which can be gathered either in batches or one at a time. The information obtained by these new samples is incorporated into the estimated objective function in an effort to move it closer to the ground truth objective function.

The placement of samples is determined by a combination of the updated belief and a defined acquisition function, which determines the trade-off between exploration and exploitation. The acquisition function assigns each point in the parameter space a score based on its expected contribution to the optimisation goal, effectively directing the search process. A standard method for modelling the objective function in Bayesian optimisation is Gaussian Processes, but other techniques are available, such as Random Forests and Bayesian Neural Networks, among others. This optimisation technique is often employed when evaluating the objective function is expensive or time-consuming, as it aims to find the optimal solution with as few evaluations as possible.

Fig: An example of the samples a Bayesian optimisation might use to find an optimal hyperparameter solution.

5.1.5 Gradient-Based Optimisation

Genetic Algorithms are optimisation methods which can be used to find a set of input parameters which maximise a given fitness function. Often, this fitness function measures the performance of a

certain process. In our case the process being measured is the training and testing of a given set of ANN hyperparameters - the hyperparameters then, are the input parameters which are being optimised.

	“Hyperparameters (genes)”, “”, “”	“Base Genes (1 each per genome)”, “”, “”
“Name”, “Min”, “Max”	“Structural”, “”, “”	“Num Layers (int)”, “”, “”
“Input Alignment Type (enum)”, “”, “”	“Training”, “”, “”	“Loss Type (enum)”, “”, “”
“Optimiser Type (enum)”, “”, “”	“Learning Rate (double)”, “”, “”	“Batch Size (int)”, “”, “”
“Num Epocs (int)”, “”, “”	“Num Semesters (int)”, “”, “”	“Dataset”, “”, “”
“Num Training Examples (int)”, “”, “”	“Layer Genes (1 each per layer per genome)”, “”, “”	“Name”, “Min”, “Max”
“Layer Type (enum)”, “”, “”	“Dense”, “”, “”	“Number of Dense ANs (int)”, “”, “”
“Convoloutional”, “”, “”	“Number of Kernels (int)”, “”, “”	“Kernel Size (int)”, “”, “”
“Kernel Stride (int)”, “”, “”	“Kernel Dilation (int)”, “”, “”	“Pooling”, “”, “”
“Pooling Present (bool)”, “”, “”	“Pooling Type (enum)”, “”, “”	“Pooling Size (int)”, “”, “”
“Pooling Stride (int)”, “”, “”	“Batch Norm”, “”, “”	“Batch Norm Present (bool)”, “”, “”
“Dropout”, “”, “”	“DropOut Used (bool)”, “”, “”	“DropOut Value (double)”, “”, “”
“Activation”, “”, “”	“Activation Present (bool)”, “”, “”	“Activation Function (enum)”, “”, “”

Optimised parameters are called genes (g), and each set of genes are called a genome textbf{genomes} (G). $G = \left[g_1, g_i ... g_{\{x\}}\right]$, where x is the number of input parameters. Each genome should map to a single fitness score (F) via the fitness function.

Genetic algorithms operate under the following steps, note that this describes the procedure as performed in this paper, slight variations on the method are common:

begin{enumerate} item textbf {Generation:} First, an initial population of genomes, P is generated. $P = [G_1, G_i, ... G_N]$, where N is the number of genomes in the population. Each genome is randomised, with each gene limited within a search space defined by $g_{\{i\}}\{\min\}$ and $g_{\{i\}}\{\max\}$. item textbf{Evaluation:} Next, each genome is evaluated by the fitness function to produce an initial fitness score. In our case this means that each genome is used to construct a CNN model which is trained and tested. The result of each test is used to generate a fitness score for that genome. item textbf{Selection:} These fitness scores are used to select which genomes will continue to the next generation. There are a few methods for doing this, however since we do not expect to need any special functionality in this area we have used the most common selection function - “the Roulette Wheel” method. In this method the fitness scores are normalised so that the sum of the scores is unity. Then the fitness scores are stacked into bins with each bin width determined by that genomes fitness score. N random numbers between 0 and 1 are generated, each genome is selected by the number of random numbers that fall into its bin. Any given genome can be selected multiple or zero times. item textbf{Crossover and mutations:} The genomes that have been selected are then acted upon by two genetic operators, crossover and mutation. Firstly, genomes are randomly paired into groups of two, then two new genomes are created by randomly selecting genes from each parent. A bit-wise mutation is then performed on each of the new genomes with a certain mutation probability M. Mutation and Crossover creates genomes which are similar to both parents but with enough differences to continue exploring the domain space.

item textbf{Termination:} If the desired number of generations has been reached the process ends and the highest performing solution is returned. Else-wise the process loops back to step 2 and the newly created genomes are evaluated. end{enumerate}

subsection{Example Data}

Three sets of data were independently generated using identical parameters but differing random seeds - training, testing, and validation datasets. The training and testing datasets were used during the training of each model during each generation. The same datasets were used for each genome - however each was independently shuffled with a different seed for every case.

The datasets parameters were chosen to match as closely as possible, the following paper by cite{Gabbard2018}.

SNR - discussion - range vs single value, high snr vs low snr

5.1.6 Population-Based methods

Population

5.1.7 Why genetic algorithms?

5.2 Layer Configuration Tests

Testing which combinations of layers are most effective.

5.2.1 Dense Layers

5.2.2 Convolutional Layers

5.2.3 Regularisation

5.2.4 Custom Layer Exploration

5.3 Input Configuration Tests

Testing which input method is most effective. I.e. number of detectors and widthwise, lengthwise, or depthwise.

One detector vs multiple.

SNR cutoff point.

5.3.1 Noise Type Tests

Also, noise type.

And feature engineering.

5.4 Output Configuration Tests

Baysian tests

5.5 Label Configuration Tests

Testing which configuration of the label is the most effective combination of noise, glitch, etc.

5.6 Branched Exploration

5.7 All together

5.8 Deployment in MLy

6 Skywarp: An Attention-Based model for the Detection of Gravitational-Wave Compact Binary Coalescences

6.1 Attention!

The global information provided by an individual element within a sequence is often greater than the local information contained within the isolated datum. This extra information is stored contextually within the relationship between the given element and the other elements in the sequence, both within the information stored locally by the other elements and by the relative and absolute positions of the other elements.

The set of possible combinations of elements is large, even within relatively small sequences. Therefore, in order to regularise a machine learning model to extract contextual information efficiently, a method must be implemented to determine which elements contribute the most contextual information to a given datum. This method is attention. Attention determines which elements in the sequence contribute highly to the global information of a given element. Once attention has been determined, global contextual information can be embedded within each element's local information. Ideally, this process makes the output elements, now with contextual information embedded locally, easier for other machine-learning methods to interpret.

A transformer model is a machine learning algorithm which implements this method to localise global information using attention. The output to a transformer block has the same dimensionality as the block's input, as it retains the same number of elements. Ideally, each element has been transformed to contain a proportion of the global information stored within the input sequence.

The models we describe in this section are novel in that they utilise attention mechanisms (Bahdanau et al. 2014, Luong et al. 2015), a type of differentiable memory in which a global context vector is learnt over an input sequence $x_{\{i\}}$.

The aim of the attention mechanism is to embed global context locally; in order to do this, a comparison must be made between each element of the sequence and (in the case of self-attention) each other element of the same sequence. It is trivial to see that not every element in every sequence will be equally contextual to each other and that the contextual dependence will depend on the information being extracted.

This corresponds to learning the following relation:

$$y_{\{j\}} = \alpha^{\{i\}}_{\{j\}} x_{\{i\}}$$

begin{equation}

Where $\alpha^{\{i\}}_{\{\{i\}j\}} \in \mathfrak{R}$ are the scalar attention scores measuring the relative correlation between $x_{\{i\}}$ and $x_{\{j\}}$. In this way, one learns intra-sequence relations; long-term dependencies are captured because the entire input sequence is used to compute a single element of the output sequence.

In order to calculate these attention scores, we generate three vectors, namely a query $q_{\{i\}}$, key $k_{\{i\}}$, and value $v_{\{i\}}$ vector, for each sequence element $x_{\{i\}}$, forming three matrices for the sequence as a whole: Q, K, and V. These matrices are created by applying projection matrices: $W_{\{Q\}}$, $W_{\{K\}}$ and $W_{\{V\}}$ to the input sequence, the values of these weights matrices are learned via backpropagation during the model training process.

The query, key, and value matrices are used to calculate the attention scores for each element. The query value for each sequence element is matched against the key value of each other element, the alignment of the key and query determines a weighting for the value vector, a distilled representation

of the information contained within that element. The weighted value vectors are then summed to produce the new, contextually embedded, sequence.

The two most commonly used attention functions are dot-product (Luong et al. 2015) and additive attention (Bahdanau et al. 2014), our models utilise the former and so we restrict our discussion to the work of (Luong et al. 2015) and extensions. In either case, the function α maps a set of query q, key k and value v vectors to a weighted sum of the values.

$$\alpha(q, K, V) = \sum_{\{i\}} a(q, k_i) v_i$$

Where $a(.,.)$ is called the alignment function and measures the similarity between the queries and keys. In the case of dot-product attention proposed by (Luong et al. 2015) :

$$a(q, k) = \sigma\big(q^{\{T\}} k\big)$$

Where σ is the Softmax function (). This calculation is performed on each element of the sequence to produce a new sequence of equal length, hopefully with some contextual context embedded. Generalising the attention function we get:

$$\alpha(Q, K, V) = \sigma\big(Q K^{\{T\}}\big) V$$

Where again, σ is the Softmax function.

6.2 Transformers

Since their introduction, attention mechanisms have been utilised in a number of different neural network architectures, including transformers and stable diffusion models. Transformers were first proposed by (Vaswani et al. 2017) to solve natural-language processing tasks, showing significant improvement over previous recurrent and convolutional architectures. For these reasons, we decided to investigate a fully attention-based model, inspired by a Transformer encoder.

The transformer model uses the attention mechanism described earlier in section Section 6.1 within discrete blocks called multi-attention heads. Multi-attention heads have N multiples of the weights matrices used to generate query, key, and value matrices from input sequences. These multiple heads can be thought of in an analogous manner to different convolutional filters inside a CNN layer; each head can focus on extracting a different type of contextual information from the sequence. This is necessary as the useful contextual information embedded within a sequence can be more complex than it is possible to extract with a single attention head. The output sequences of all N heads are merged after the block to ensure that the output and input sizes are the same.

Often, as is the case for our models, the multi-attention heads are followed by normalisation and one or more dense layers. These blocks of layers can then be stacked to form components of a transformer.

%embedding %positional encoding,

6.3 Literature

Chatterjee et al offer perhaps the most relevant work, they utilize Long Short Term Memory (LSTM) networks, a form of Recurrent Neural Network, for both the problems of signal detection and reconstruction. Recurrent neural networks have an internal state determined by previous inferences, and thus, they have the ability to retain some information about all previous data. In many ways, RNNs were the predecessor to Transformer models, largely because they are able, in some way, to make inferences from global information rather than being limited by the receptive fields of convolutional filters.

There have also been some attempts to detect other proposed signal types, known as gravitational-wave bursts, with machine learning methods, most prominently core-collapse supernovae. Although there has yet to be a real detection of any such signals, these types of signals are in many ways a more interesting candidate for machine learning methods due to the much higher degree of uncertainty in waveform modelling. The physics behind burst signals is often much less understood, as well as possessing a much larger number of degrees of freedom. There have also been a few papers which have attempted to use machine learning methods as a coherence detection technique, therefore eschewing any requirement for accurate signals. Such detection problems could also benefit from the application of transformer models and will be an area of future inquiry.

As demonstrated, there has been a considerable investigation into the use of machine learning techniques for gravitational wave detection. However, there has not been a significant investigation into the use of transformers for such a purpose, with only this paper by Zhao et al known at this time, which focuses on the problem of space-based detection. https://arxiv.org/pdf/2207.07414.pdf

subsection{Model}

We have investigated a number of attention-based models and compared their performance to convolutional models as well as other traditional analysis techniques. We have investigated a fully attention-based model, as well as combined convolutional-attention model.

For the CNN model, we adapted a model from the literature. [] Architecture is as described in figure ref{fig:cnn}.

Type	Num Filters / Neurons	Filter / Pool Size	Stride	Activation Function	Dropout
Convolutional	64	8	1	ReLU	-
MaxPooling	-	8	8	-	-
Convolutional	32	8	1	ReLU	-
Convolutional	32	16	1	ReLU	-
MaxPooling	-	6	6	-	-
Convolutional	16	16	1	ReLU	-
Convolutional	16	32	1	ReLU	-
MaxPooling	-	4	4	-	-
Convolutional	16	32	1	ReLU	-
Dense	64	-	-	ReLU	0.5
Dense	64	-	-	ReLU	0.5
Dense	2	-	-	SoftMax	-

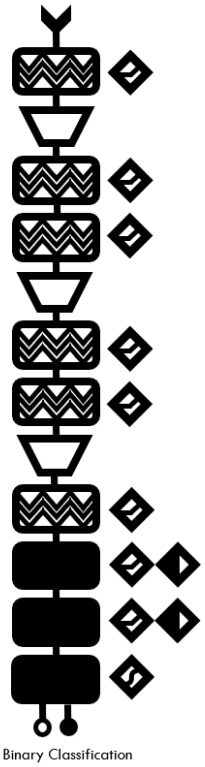


Figure 46 |

%This needs major edits. When transformers are utilised for Natural Language Processing (NLP) tasks, the input strings of natural language are first tokenised into discrete tokens before those tokens are fed into an embedding layer to convert the discrete tokens into continuous vectors that the network can ingest. When adapting the architecture for use on time series data, there are some design decisions that

must be taken. Tokenization, although still possible, is not longer required as the input data is initially in a continuous form. However, when deciding how to feed the series into the transformer, there are several options. Although it is possible to feed an attention block with the length one vector from the input time series, it was found that this naive approach eliminated much of the transformer's potential for element-wise comparison. To resolve this, the method used by the vision transformer can be used; the input data can be segmented into N segments, and then fed into the network. In addition or in place of such a segmentation, an embedding layer can also be employed to increase the dimensionality of the segments.

In the pure attention model, we reshaped the input time series (1s at 8192Hz) into 512 segments of size 16, these segments were then encoded into larger vectors of length 128 by a single convolutional layer with a filter size of 1. This embedding was performed to allow sufficient size for the positional encoding to be added to each vector. This solution was found after trialling several variations. See figure ref{fig: transformer}, for more detailed information on the network.

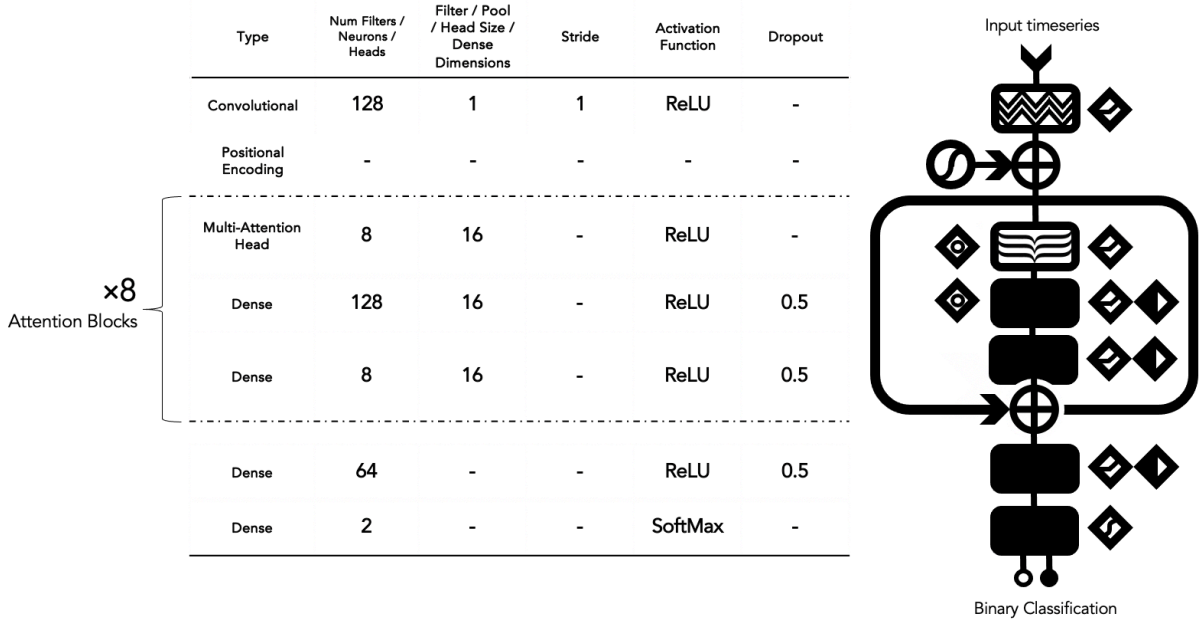


Figure 47 |

On testing, we found that the pure attention model did not perform as well as the CNN model. It was found that the transformer model could much more easily overfit to the training data, even with large training datasets. In order to combat this - a combination convolutional-attention model was introduced. This model, described in figure ref{fig:conv_transformer}, feeds the output of the convolutional layers from the CNN described by figure ref{fig: transformer} into the attention blocks described in figure ref{fig:cnn}, in attempts to gain the benefits of both methods.

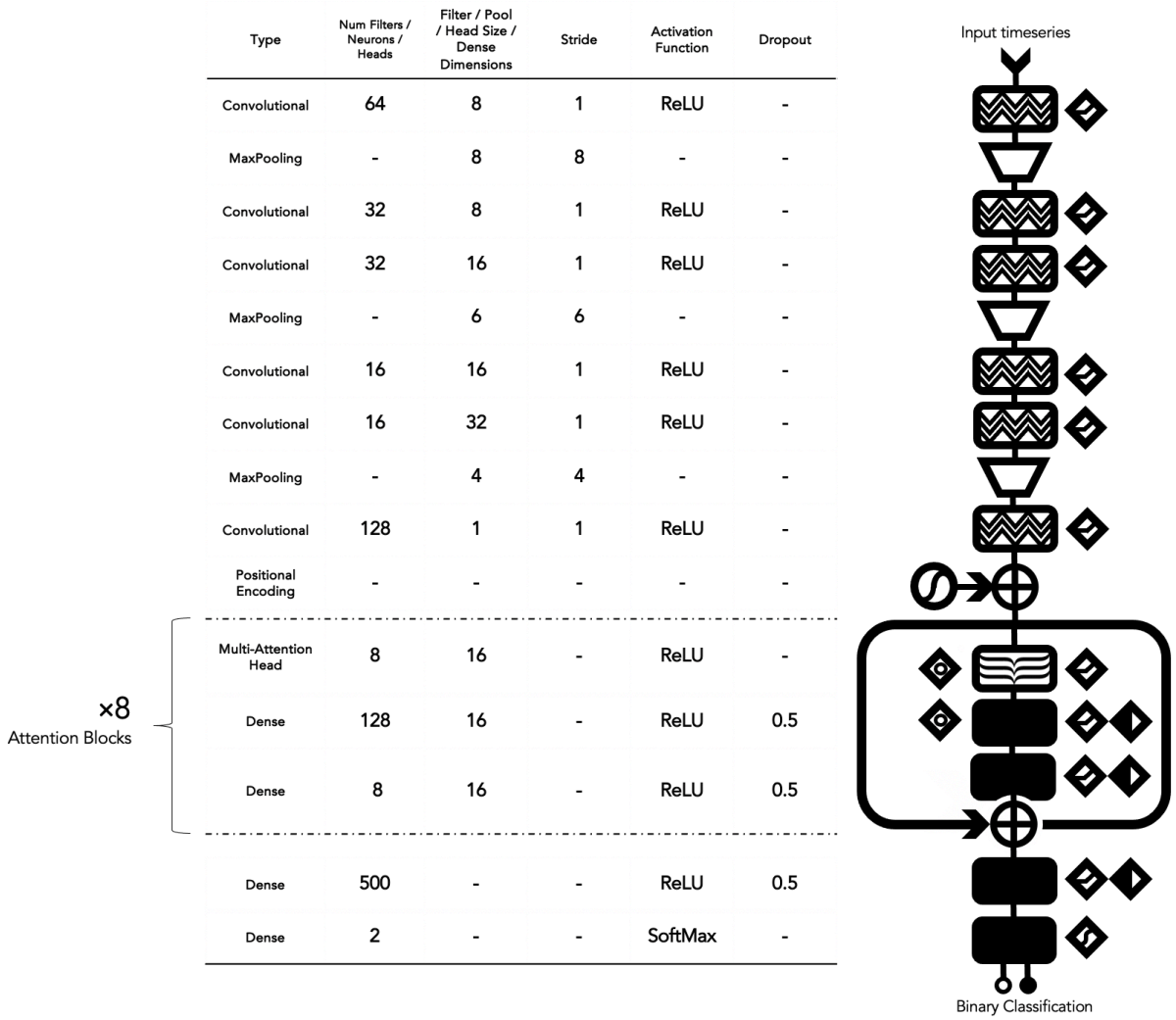


Figure 48 |

subsection{Training, Testing, and Validation Data}

subsection{Training Procedure}

section{Results label{sec:results}}

To profile the performance of Skywarp we compare it against three alternative techniques: the standard matched filtering approach which is the current method used to confirm the detection of CBC signals; a Convolutional Neural Network (CNN) with architecture taken from this early paper by George et Al, as this was the first type of neural network architecture applied to CBC detection; and a Recurrent Neural Network, as another neural network architecture which is commonly used for time series analysis.

begin{table}[] begin{tabular}{|l||l|l|l|l|} hline SNR & Matched Filtering & CNN & RNN & Skywarp \ hline 4 & & & & \ hline 6 & & & & \ hline 8 & & & & \ hline 10 & & & & \ hline end{tabular} end{table}

section{label{sec:discussion} Discussion}

textbf{}

7 CrossWave: Detection and Parameterisation of Overlapping Compact Binary Coalescence Signals

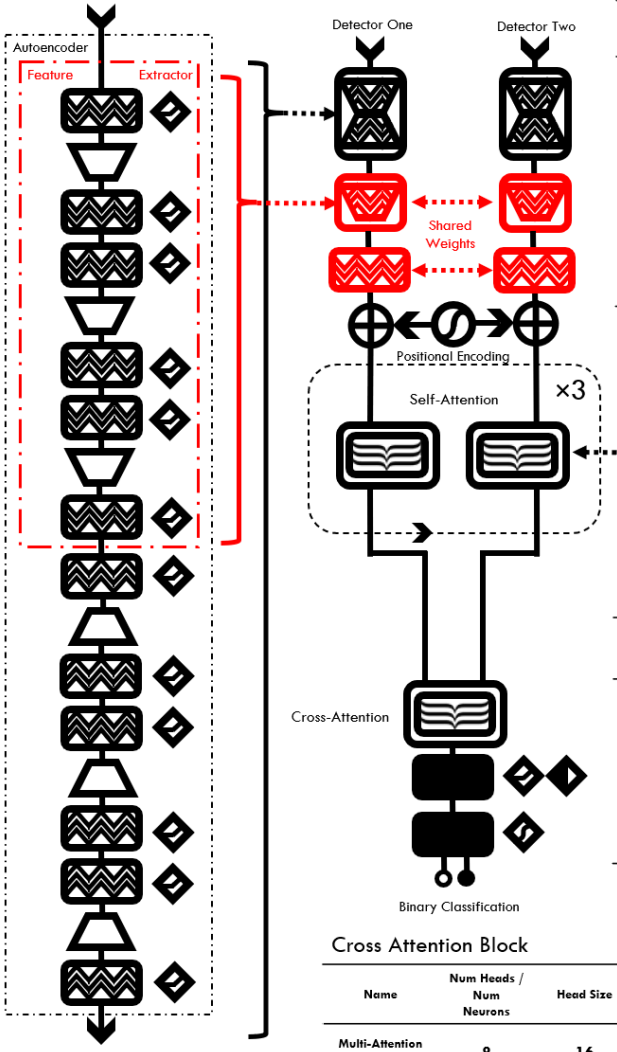
We introduce CrossWave, a new attention-based neural network model for the identification and parameter estimation of overlapping CBC signals. CrossWave can with efficiencies matching that of more conventional matched filtering techniques, separate the case of overlapping mergers from lone mergers, but with considerably lower inference times and computational cost. We suggest CrossWave or a similar architecture may be used to augment existing CBC detection and parameter estimation infrastructure, either as a complementary confirmation of the presence of overlap or to extract the merger times of each signal in order to use other parameter estimation techniques on the separated parts of the signals.

Significant improvements to our gravitational wave detection capability are anticipate,d within the next decade, with improvements to existing detectors [cite], as well as future 3rd and 4th generation space and ground-based detectors such as the Einstein Telescope [cite] and Cosmic Explorer. Whilst the current rate of Compact Binary Coalescence (CBC) detection is too low (estimate) for any real concern about the possibility of overlapping detections, estimated rates for future networks (estimate) will render such events a significant percentage of detections.

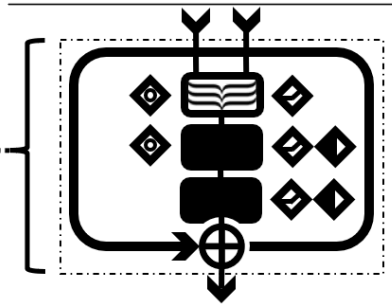
Contemporary detection and parameter pipelines do not currently have any capabilities to deal with overlapping signals - and although, in many cases, detection would still be achieved [cite], it is likely that parameter estimation would be compromised by the presence of the overlap, especially if more detailed information about higher modes and spins [cite] are science goals.

We introduce CrossWave, two attention-based neural network models for the identification and parameter estimation of overlapping CBC signals. CrossWave consists of two complementary models, one for the separation of the overlapping case from the non-overlapping case and the second as a parameter estimation follow-up to extract the merger times of the overlapping signals in order to allow other parameter estimation methods to be performed.

Type	Num Filters	Filter / Pool Size	Stride	Activation Function
Convolutional	64	8	1	ReLU
MaxPooling	-	8	8	-
Convolutional	32	8	1	ReLU
Convolutional	32	16	1	ReLU
MaxPooling	-	6	6	-
Convolutional	16	16	1	ReLU
Convolutional	16	32	1	ReLU
MaxPooling	-	4	4	-
Convolutional	16	32	1	ReLU
Transpose Convolutional	16	32	1	ReLU
Upscaling	-	4	4	-
Transpose Convolutional	16	32	1	ReLU
Transpose Convolutional	16	16	1	ReLU
Upscaling	-	6	6	-
Transpose Convolutional	32	16	1	ReLU
Transpose Convolutional	32	8	1	ReLU
MaxPooling	-	8	8	-
Transpose Convolutional	64	8	1	ReLU



Name	Number Filters	Filter Size	Activation Function	Dropout
Denoising Autoencoder (Independent Weights)	-	-	-	-
Feature Extractor (Weights Shared)	-	-	-	-
CNN Embedding (Weights Shared)	128	1	ReLU	-

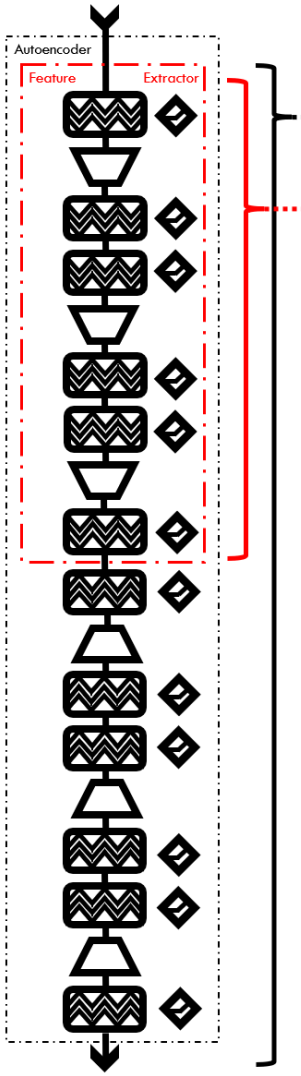


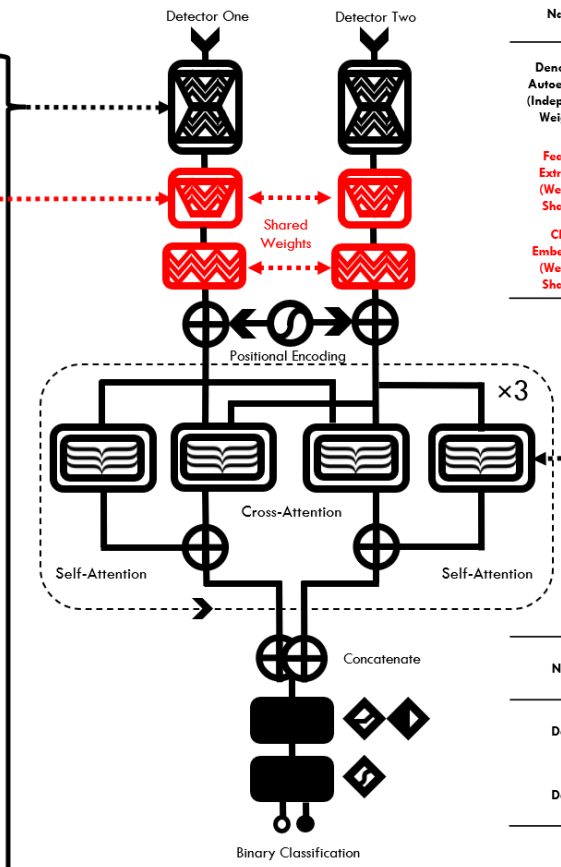
Name	Number of Neurons	Activation Function	Dropout
Attention Block	-	-	-
Dense	500	ReLU	0.5
Dense	2	SoftMax	-

Cross Attention Block

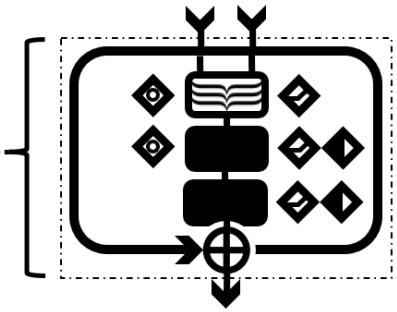
Name	Num Heads / Num Neurons	Head Size	Activation Functions	Dropout
Multi-Attention Head	8	16	ReLU	-
Dense	128	16	ReLU	0.5
Dense	8	16	ReLU	0.5

Type	Num Filters	Filter / Pool Size	Stride	Activation Function
Convolutional	64	8	1	ReLU
MaxPooling	-	8	8	-
Convolutional	32	8	1	ReLU
Convolutional	32	16	1	ReLU
MaxPooling	-	6	6	-
Convolutional	16	16	1	ReLU
Convolutional	16	32	1	ReLU
MaxPooling	-	4	4	-
Convolutional	16	32	1	ReLU
Transpose Convolutional	16	32	1	ReLU
Upscaling	-	4	4	-
Transpose Convolutional	16	32	1	ReLU
Transpose Convolutional	16	16	1	ReLU
Upscaling	-	6	6	-
Transpose Convolutional	32	16	1	ReLU
Transpose Convolutional	32	8	1	ReLU
MaxPooling	-	8	8	-
Transpose Convolutional	64	8	1	ReLU





Name	Number Filters	Filter Size	Activation Function	Dropout
Denoising Autoencoder (Independent Weights)	-	-	-	-
Feature Extractor (Weights Shared)	-	-	-	-
CNN Embedding (Weights Shared)	128	1	ReLU	-



Name	Number of Neurons	Activation Function	Dropout
Dense	500	ReLU	0.5
Dense	2	SoftMax	-

Cross Attention Block

Name	Num Heads / Num Neurons	Head Size	Activation Functions	Dropout
Multi-Attention Head	8	16	ReLU	-
Dense	128	16	ReLU	0.5
Dense	8	16	ReLU	0.5

8 Software Development

8.1 cuPhenom

8.1.1 Contributions

9 Related Work

51112

Bibliography

[1] B. P. Abbott *et al.*, "Observation of Gravitational Waves from a Binary Black Hole Merger", *Physical Review Letters*, no. 6, p. 61102, Feb. 2016, Available: https://doi.org/10.1103/PhysRevLett.116.061102

[2] Sir Issac Newton, *Newton's Principia: the mathematical principles of natural philosophy*. 1846. Available: https://archive.org/details/newtonspmathema00newtrich/

[3] Stuart J. Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021, pp. 1–5.

[4] Murray Campbell, A. Joseph Hoane, and Feng-hsiung Hsu, "Deep Blue", *Artificial Intelligence*, pp. 57–83, 2002, Available: https://doi.org/10.1016/S0004-3702(01)00129-1

[5] Edward Moore Geist, "It's already too late to stop the AI arms race—We must manage it instead", *Bulletin of the Atomic Scientists*, pp. 318–321, 2016, Available: https://doi.org/10.1080/00963402.2016.1216672

[6] Brady D. Lund, Ting Wang, Nishith Reddy Mannuru, Bing Nie, Somipam Shimray, and Ziang Wang, "ChatGPT and a new academic reality: Artificial Intelligence-written research papers and the ethics of the large language models in scholarly publishing", *periodical of the Association for Information Science and Technology*, pp. 570–581, 2023, Available: https://doi.org/10.1002/asi.24750

[7] Pat Langley, "The changing science of machine learning", *Machine Learning*, pp. 275–279, Feb. 2011, Available: https://doi.org/10.1007/s10994-011-5242-y

[8] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers", *IBM periodical of Research and Development*, pp. 210–229, 1959, Available: https://doi.org/10.1147/rd.33.0210

[9] Christopher M. Bishop, *Pattern Recognition and Machine Learning*. Springer New York, NY, 2006, p. 3.

[10] M. Hady, A. Faeouk, and F. Schwenker, "Semi-supervised Learning", in *Handbook on Neural Information Processing*, Springer Berlin Heidelberg, 2013, pp. 215–239. Available: https://doi.org/10.1007/978-3-642-36657-4_7

[11] J. Morimoto, G. Cheng, C. Atkeson, and G. Zeglin, "A simple reinforcement learning algorithm for biped walking", 2004, pp. 3030–3035. Available: https://doi.org/10.1109/ROBOT.2004.1307522

[12] J. Koutník, C. Giuseppe, S. Jürgen, and G. Faustino, "Evolving large-scale neural networks for vision-based reinforcement learning", 2013, pp. 1061–1068. Available: https://doi.org/10.1145/2463372.2463509

[13] M. Zahangir Alom *et al.*, "A State-of-the-Art Survey on Deep Learning Theory and Architectures", *Electronics*, 2019, Available: https://doi.org/10.3390/electronics8030292

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", May 2017, pp. 84–90. Available: https://doi.org/10.1145/3065386

[15] K. J. Piczak, "Environmental sound classification with convolutional neural networks", 2015, pp. 1–6. Available: https://doi.org/10.1109/MLSP.2015.7324337

[16] Y. Kim, "Convolutional Neural Networks for Sentence Classification", Association for Computational Linguistics, 2014, pp. 1746–1751. Available: https://doi.org/10.3115/v1/D14-1181

[17] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and Tell: A Neural Image Caption Generator", 2015. Available: https://doi.org/10.1109/CVPR.2015.7298935

[18] S. Ikawa and K. Kashino, "Neural Audio Captioning Based on Conditional Sequence-to-Sequence Model", Oct. 2019, pp. 99–103. Available: https://doi.org/10.33682/7bay-bj41

[19] Alexander M. Rush, Sumit Chopra, and Jason Weston, "A Neural Attention Model for Abstractive Sentence Summarization", Sep. 2015, pp. 379–389. Available: https://doi.org/10.18653/v1/D15-1044

[20] I. Goodfellow *et al.*, "Generative Adversarial Nets", Curran Associates, Inc., Dec. 2014, pp. 2672–2680. Available: https://doi.org/10.5555/2969033.2969125

[21] Aäron van den Oord *et al.*, "WaveNet: A Generative Model for Raw Audio". 2016. Available: https://www.deepmind.com/blog/wavenet-a-generative-model-for-raw-audio

[22] I. Sutskever, James Martens, and Geoffrey E. Hinton, "Generating text with recurrent neural networks", 2011, pp. 1017–1024.

[23] David Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search", *Nature*, pp. 484–489, 2016, Available: https://doi.org/10.1038/nature16961

[24] David Silver *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play", *Science*, pp. 1140–1144, 2018, Available: https://doi.org/10.1126/science.aar6404

[25] Oriol Vinyals *et al.*, "Grandmaster level in StarCraft II using multi-agent reinforcement learning", pp. 350–354, 2019, Available: https://doi.org/10.1038/s41586-019-1724-z

[26] A. Lohokare, A. Shah, and M. Zyda, "Deep Learning Bot for League of Legends", 2020, pp. 322–324. Available: https://doi.org/10.1609/aiide.v16i1.7449

[27] M.N.Q. Macedo, J.J.M. Galo, L.A.L. de Almeida, and A.C. de C. Lima, "Demand side management using artificial neural networks in a smart grid environment", *Renewable and Sustainable Energy Reviews*, pp. 128–133, 2015, Available: https://doi.org/10.1016/j.rser.2014.08.035

[28] S. T. A. Niaki and M. Davoodi, "Designing a multivariate–multistage quality control system using artificial neural networks", *International periodical of Production Research*, Jan. 2009, Available: https://doi.org/10.1080/00207540701504348

[29] Claudine Badue *et al.*, "Self-driving cars: A survey", *Expert Systems with Applications*, p. 113816, 2021, Available: https://doi.org/10.1016/j.eswa.2020.113816

[30] J. Jumper *et al.*, "Highly accurate protein structure prediction with AlphaFold". in Nature, no. 596. pp. 583–589, Aug. 01, 2021. Available: https://doi.org/10.1038/s41586-021-03819-2

[31] S. C. Kleene, "Representation of Events in Nerve Nets and Finite Automata", in *Automata Studies. (AM-34)*, 1956, pp. 3–42. Available: https://doi.org/10.1515/9781400882618-002

[32] Python Software Foundation, “Python Programming Language”. Available: https://www.python.org/

[33] Charles R. Harris *et al.*, “Array programming with NumPy”, *Nature*, pp. 357–362, Sep. 2020, Available: https://doi.org/10.1038/s41586-020-2649-2

[34] Martin Abadi *et al.*, “Large-Scale Machine Learning on Heterogeneous Systems”. 2015. Available: https://doi.org/10.5281/zenodo.4724125

[35] L. Deng, “The mnist database of handwritten digit images for machine learning research”, *IEEE Signal Processing Magazine*, 2012, Available: https://doi.org/10.1109/MSP.2012.2211477

[36] Bokeh Development Team, “Bokeh: Python library for interactive visualization”. 2018. Available: https://bokeh.pydata.org/en/latest/

[37] Noah Golowich, Alexander Rakhlin, and S. Ohad, “Size-Independent Sample Complexity of Neural Networks”, Jul. 2018, pp. 297–299. Available: https://proceedings.mlr.press/v75/golowich18a.html

[38] Xia Hu, Jian Pei, Lingyang Chu, Weiqing Liu, and Jiang Bian, “Model complexity of deep learning: a survey”, *Knowledge and Information Systems*, Aug. 2021, Available: https://doi.org/10.1007/s10115-021-01605-0

[39] B. Haskell and M. Bejger, “Astrophysics with continuous gravitational waves”, *Nature Astronomy*, Sep. 2023, Available: https://doi.org/10.1038/s41550-023-02059-w

[40] N. J. Cornish and K. Shuman, “Black hole hunting with LISA”, *Physical Review D*, no. 12, Available: https://doi.org/10.1103/PhysRevD.101.124008

[41] D Davis *et al.*, “LIGO detector characterization in the second and third observing runs”, *Classical and Quantum Gravity*, p. 135014, Jun. 2021, Available: https://doi.org/10.1088/1361-6382/abfd85

[42] B. P. Abbott *et al.*, “All-sky search for short gravitational-wave bursts in the first Advanced LIGO run”, *Physical Review D*, no. 4, p. 42003, Feb. 2016, Available: https://doi.org/10.1103/PhysRevD.95.042003

[43] B. P. Abbott *et al.*, “All-sky search for short gravitational-wave bursts in the second Advanced LIGO and Advanced Virgo run”, *Physical Review D*, no. 2, p. 24017, Jul. 2019, Available: https://doi.org/10.1103/PhysRevD.100.024017

[44] R. Abbott *et al.*, “All-sky search for short gravitational-wave bursts in the third Advanced LIGO and Advanced Virgo run”, *Physical Review D*, no. 12, p. 122004, Dec. 2021, Available: https://doi.org/10.1103/PhysRevD.104.122004

[45] David Radice, Viktoriya Morozova, Adam Burrows, David Vartanyan, and Hiroki Nagakura, “Characterizing the Gravitational Wave Signal from Core-collapse Supernovae”, *The Astrophysical periodical Letters*, Apr. 2019, Available: https://doi.org/10.3847/2041-8213/ab191a

[46] Antonis Mytidis, Michael Coughlin, and Benrand Whiting, “Constraining the R-Mode Saturation Amplitude from a Hypothetical Detection of R-Mode Gravitational waves from a newborn neutron star”, *The Astrophysical periodical*, Aug. 2015, Available: https://doi.org/10.1088/0004-637X/810/1/27

[47] Patrick J Sutton *et al.*, “X-Pipeline: an analysis package for autonomous gravitational-wave burst searches”, *New periodical of Physics*, p. 53034, May 2010, Available: https://doi.org/10.1088/1367-2630/12/5/053034

[48] R. Lynch, S. Vitale, R. Essick, E. Katsavounidis, and F. Robinet, "Information-theoretic approach to the gravitational-wave burst detection problem", *Physical Review D*, no. 10, May 2017, Available: https://doi.org/10.1103/PhysRevD.95.104046

[49] Marco Drago *et al.*, "coherent WaveBurst, a pipeline for unmodeled gravitational-wave data analysis", *SoftwareX*, p. 100678, 2021, Available: https://doi.org/10.1016/j.softx.2021.100678

[50] Neil J Cornish and Tyson B Littenberg, "Bayeswave: Bayesian inference for gravitational wave bursts and instrument glitches", *Classical and Quantum Gravity*, p. 135012, Jun. 2015, Available: https://doi.org/10.1088/0264-9381/32/13/135012

[51] Vasileios Skliris, Michael R. K. Norman, and Patrick J. Sutton, "Real-Time Detection of Unmodelled Gravitational-Wave Transients Using Convolutional Neural Networks", Arxiv. [Online]. Available: https://doi.org/10.48550/arXiv.2009.14611

[52] B. P. Abbott *et al.*, "Multi-messenger Observations of a Binary Neutron Star Merger*", *The Astrophysical Journal Letters*, Oct. 2017, Available: https://doi.org/10.3847/2041-8213/aa91c9

[53] R. Abbott *et al.*, "Observation of Gravitational Waves from Two Neutron Star–Black Hole Coalescences", *The Astrophysical Journal Letters*, Jun. 2021, Available: https://doi.org/10.3847/2041-8213/ac082e

[54] B. P. Abbott *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*, no. 3, p. 31040, Sep. 2019, Available: https://doi.org/10.1103/PhysRevX.9.031040

[55] R. Abbott *et al.*, "GWTC-2: Compact Binary Coalescences Observed by LIGO and Virgo during the First Half of the Third Observing Run", *Physical Review X*, no. 2, p. 21053, Jun. 2021, Available: https://doi.org/10.1103/PhysRevX.11.021053

[56] R. Abbott *et al.*, "GWTC-3: Compact Binary Coalescences Observed by LIGO and Virgo During the Second Part of the Third Observing Run", Arxiv. [Online]. Available: https://doi.org/10.48550/arXiv.2111.03606

[57] J. Antoniadis *et al.*, "The second data release from the European Pulsar Timing Array - III. Search for gravitational wave signals EPTA Collaboration and InPTA Collaboration", *Astronomy & Astrophysics*, 2023, Available: https://doi.org/10.1051/0004-6361/202346844

[58] Warren R. Brown, Mukremin Kilic, Carlos Allende Prieto, A. Gianninas, and Scott J. Kenyon, "The ELM Survey. V. Merging Massive White Dwarf Binaries", *The Astrophysical Journal*, p. 66, May 2013, Available: https://doi.org/10.1088/0004-637X/769/1/66

[59] A. Lamberts, S. Blunt, T. B. Littenberg, S. Garrison-Kimmel, T. Kupfer, and R. E. Sanderson, "Predicting the LISA white dwarf binary population in the Milky Way with cosmological simulations", *JF Monthly Notices of the Royal Astronomical Society*, no. 4, 2019, Available: https://doi.org/10.1093/mnras/stz2834

[60] P. A. R. Ade *et al.*, "Planck 2015 results: XIII. Cosmological parameters", *Astronomy & Astrophysics*, Sep. 2016, Available: https://doi.org/10.1051/0004-6361/201525830

[61] Chiara Caprini and Daniel G Figueroa, "Cosmological backgrounds of gravitational waves", *Classical and Quantum Gravity*, p. 163001, Jul. 2018, Available: https://doi.org/10.1088/1361-6382/aac608

[62] B. Link, L. M. Franco, and R. I. Epstein, "Starquake-induced Magnetic Field and Torque Evolution in Neutron Stars", *Astrophysical Journal*, Dec. 1998, Available: https://doi.org/10.1086/306457

[63] E. Giliberti and G. Cambiotti, “Starquakes in millisecond pulsars and gravitational waves emission”, *Monthly Notices of the Royal Astronomical Society*, no. 3, 2022, Available: https://doi.org/10.1093/mnras/stac245

[64] M. Modjaz, C. P. Gutiérrez, and I. Arcavi, “New regimes in the observation of core-collapse supernovae”, *Nature Astronomy*, no. 8, pp. 717–724, Aug. 2019, Available: https://doi.org/10.1038/s41550-019-0856-2

[65] S. E. Gossan, P. Sutton, A. Stuver, M. Zanolin, K. Gill, and C. D. Ott, “Observing gravitational waves from core-collapse supernovae in the advanced detector era”, *Physical Review D*, no. 4, p. 42002, Feb. 2016, Available: https://doi.org/10.1103/PhysRevD.93.042002

[66] N. Andersson *et al.*, “Gravitational waves from neutron stars: promises and challenges”, *General Relativity and Gravitation*, no. 2, pp. 409–436, Feb. 2011, Available: https://doi.org/10.1007/s10714-010-1059-4

[67] M. Abramowicz, M. Bejger, É. Gourgoulhon, and O. Straub, “A Galactic centre gravitational-wave Messenger”, *Scientific Reports*, no. 1, Apr. 2020, Available: https://doi.org/10.1038/s41598-020-63206-1

[68] S. Olmez, V. Mandic, and X. Siemens, “Gravitational-wave stochastic background from kinks and cusps on cosmic strings”, *Physical Review D*, no. 10, May 2010, Available: https://doi/10.1103/PhysRevD.81.104028

[69] M. H. P. M. van Putten, “Proposed Source of Gravitational Radiation from a Torus around a Black Hole”, *Physical Review Letters*, no. 9, p. 91101, Aug. 2001, Available: https://doi.org/10.1103/PhysRevLett.87.091101

[70] Takashi Hiramatsu, Masahiro Kawasaki, and Ken'ichi Saikawa, “On the estimation of gravitational wave spectrum from cosmic domain walls”, *Journal of Cosmology and Astroparticle Physics*, Feb. 2014, Available: https://doi.org/10.1088/1475-7516/2014/02/031

[71] Marc Favata, “The gravitational-wave memory effect”, *Classical and Quantum Gravity*, p. 84036, Apr. 2010, Available: https://doi.org/10.1088/0264-9381/27/8/084036

[72] R. Abbott *et al.*, “Search for Gravitational Waves Associated with Fast Radio Bursts Detected by CHIME/FRB During the LIGO--Virgo Observing Run O3a”, ArXiv. [Online]. Available: https://doi.org/10.48550/arXiv.2203.12038

[73] A. Macquet *et al.*, “Search for Long-duration Gravitational-wave Signals Associated with Magnetar Giant Flares”, *The Astrophysical Journal*, Sep. 2021, Available: https://doi.org/10.3847/1538-4357/ac0efd

[74] R. Abbott *et al.*, “Search for Gravitational Waves Associated with Gamma-Ray Bursts Detected by Fermi and Swift during the LIGO–Virgo Run O3b”, *The Astrophysical Journal*, Apr. 2022, Available: https://doi.org/10.3847/1538-4357/ac532b

[75] H. Grote and Y. V. Stadnik, “Novel signatures of dark matter in laser-interferometric gravitational-wave detectors”, *Physical Review Research*, no. 3, p. 33187, Dec. 2019, Available: https://doi.org/10.1103/PhysRevResearch.1.033187

[76] P. Virtanen *et al.*, “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”, *Nature Methods*, pp. 261–272, 2020, Available: https://doi.org/10.1038/s41592-019-0686-2

[77] D. Macleod, J. Areeda, S. Coughlin, T. Massinger, and A. Urban, “GWpy: A Python package for gravitational-wave astrophysics”, *SoftwareX*, p. 100657, Available: https://doi.org/10.1016/j.softx.2021.100657

[78] Kent Blackburn, Benoit Mours, Stuart Anderson, Albert Lazzarini, John Zweizig, and Ed Maros, "Specification of a Common Data Frame Format for Interferometric Gravitational Wave Detectors (IGWD)". Available: https://dcc.ligo.org/LIGO-T970130-v3/public

[79] M. Purrer, S. Khan, F. Ohme, O. Birnholtz, and L. London, "IMRPhenomD: Phenomenological waveform model". Jun. 2023. Available: https://ui.adsabs.harvard.edu/abs/2023ascl.soft07019P/abstract

[80] Christian D Ott, "The gravitational-wave signature of core-collapse supernovae", *Classical and Quantum Gravity*, Feb. 2009, Available: https://doi.org/10.1088/0264-9381/26/6/063001

[81] Christian D. Ott, Adam Burrows, Luc Dessart, and Eli Livne, "Characterizing the Gravitational Wave Signal from Core-collapse Supernovae", *Physical Review Letters*, May 2006, Available: https://doi.org/10.1103/PhysRevLett.96.201102

[82] C. M. Biwer *et al.*, "PyCBC Inference: A Python-based Parameter Estimation Toolkit for Compact Binary Coalescence Signals", *Publications of the Astronomical Society of the Pacific*, Jan. 2019, Available: https://doi.org/10.1088/1538-3873/aaef0b

[83] Piotr Jaranowski and Andrzej Kr´olak, "A. Gravitational-Wave Data Analysis. Formalism and Sample Applications: The Gaussian Case.", *Living Reveiw in Relativity*, 2005, Available: https://doi.org/10.12942/lrr-2005-3

[84] E. Cuoco, A. Iess, F. Morawski, and M. Razzano, "Machine Learning for the Characterization of Gravitational Wave Data", in *Handbook of Gravitational Wave Astronomy*, Springer Singapore, 2020, pp. 1–23.

[85] Miquel Llorens-Monteagudo, Alejandro Torres-Forné, and Antonio Marquina, "Classification of gravitational-wave glitches via dictionary learning", *Classical and Quantum Gravity*, Mar. 2019, Available: https://doi.org/10.1088/1361-6382/ab0657

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

[87] Jade Powell *et al.*, "Classification methods for noise transients in advanced gravitational-wave detectors II: performance tests on Advanced LIGO data", *Classical and Quantum Gravity*, Jan. 2017, Available: https://doi.org/10.1088/1361-6382/34/3/034002

[88] D Davis *et al.*, "LIGO detector characterization in the second and third observing runs", *Classical and Quantum Gravity*, Jun. 2021, Available: https://doi.org/10.1088/1361-6382/abfd85

[89] Reed Essick, Patrick Godwin, Chad Hanna, Lindy Blackburn, and Erik Katsavounidis, "iDQ: Statistical inference of non-gaussian noise with auxiliary degrees of freedom in gravitational-wave detectors", *Machine Learning: Science and Technology*, Dec. 2020, Available: https://doi.org/10.1088/2632-2153/abab5f

[90] Daniel George and E. A. Huerta, "Deep neural networks to enable real-time multimessenger astrophysics", *Physical Review D*, Mar. 2018, Available: https://doi.org/10.1103%2Fphysrevd.97.044039

[91] H. Gabbard, M. Williams, F. Hayes, and C. Messenger, "Matching Matched Filtering with Deep Networks for Gravitational-Wave Astronomy", *Physical Review Letters*, no. 14, Apr. 2018, Available: https://doi.org/10.1103/PhysRevLett.120.141103

[92] Daniel George and E.A. Huerta, "Deep Learning for real-time gravitational wave detection and parameter estimation: Results with Advanced LIGO data", *Physics Letters B*, pp. 64–70, 2018, Available: https://doi.org/10.1016/j.physletb.2017.12.053

[93] H.-M. Luo, W. Lin, Z.-C. Chen, and Q.-G. Huang, "Extraction of gravitational wave signals with optimized convolutional neural network", *Frontiers of Physics*, no. 1, Nov. 2019, Available: https://doi.org/10.1007/s11467-019-0936-x

[94] A. Schmitt, K. Fu, S. Fan, and Y. Luo, "Investigating Deep Neural Networks for Gravitational Wave Detection in Advanced LIGO Data", Association for Computing Machinery, 2019, pp. 73–78. Available: https://doi.org/10.1145/3339363.3339377

[95] X. Fan, J. Li, X. Li, Y. Zhong, and J. Cao, no. 6, Feb. 2019, Available: https://doi.org/10.1007/s11433-018-9321-7

[96] T. D. Gebhard, N. Kilbertus, I. Harry, and B. Scholkopf, "Convolutional neural networks: A magic bullet for gravitational-wave detection?", *Physical Review D*, no. 6, Available: https://doi.org/10.1103/PhysRevD.100.063015

[97] E. A. Huerta *et al.*, "Accelerated, scalable and reproducible AI-driven gravitational wave detection", *Nature Astronomy*, no. 10, Oct. 2021, Available: https://doi.org/10.1038/s41550-021-01405-0

[98] C. Ma, W. Wang, H. Wang, and Z. Cao, "Ensemble of deep convolutional neural networks for real-time gravitational wave signal recognition", *Physical Review D*, no. 8, Apr. 2022, Available: https://doi/10.1103/PhysRevD.105.083013

[99] H. Wang, S. Wu, Z. Cao, X. Liu, and J.-Y. Zhu, "Gravitational-wave signal recognition of LIGO data by deep learning", *Physical Review D*, no. 10, May 2020, Available: https://doi.org/10.1103/PhysRevD.101.104003

[100] Chetan Verma, Amit Reza, Dilip Krishnaswamy, Sarah Caudill, and Gurudatt Gaur, "Employing Deep Learning for Detection of Gravitational Waves from Compact Binary Coalescences", ArXiv. [Online]. Available: https://doi.org/10.48550/arXiv.2110.01883

[101] Christopher Bresten and Jae-Hun Jung, "Detection of gravitational waves using topological data analysis and convolutional neural network: An improved approach", ArXiv. [Online]. Available: https://doi.org/10.48550/arXiv.1910.08245

[102] S. Fan, Y. Wang, Y. Luo, A. Schmitt, and S. Yu, "Improving Gravitational Wave Detection with 2D Convolutional Neural Networks", 2021, pp. 7103–7110. Available: https://doi.org/10.1109/ICPR48806.2021.9412180

[103] J. Aveiro *et al.*, "Identification of binary neutron star mergers in gravitational-wave data using object-detection machine learning models", *Physical Review D*, no. 8, Oct. 2022, Available: https://doi.org/10.1103/PhysRevD.106.084059

[104] A. Menendez-Vazquez, M. Kolstein, M. Martinez, and L. M. Mir, "Searches for compact binary coalescence events using neural networks in the LIGO/Virgo second observation period author", *Physical Review D*, no. 6, May 2021, Available: https://doi.org/10.1103/PhysRevD.103.062004

[105] B.-J. Lin, X.-R. Li, and W.-L. Yu, "Binary neutron stars gravitational wave detection based on wavelet packet analysis and convolutional neural networks", *Frontiers of Physics*, no. 5, Nov. 2019, Available: https://doi.org/10.1007/s11467-019-0935-y

[106] X.-R. Li, W.-L. Yu, X.-L. Fan, and G. J. Babu, "Some optimizations on detecting gravitational wave using convolutional neural network", *Frontiers of Physics*, no. 5, Jun. 2020, Available: https://doi.org/10.1007/s11467-020-0966-4

[107] M. Andres-Carcasona, A. Menendez-Vazquez, M. Martinez, and L. M. Mir, "Searches for mass-asymmetric compact binary coalescence events using neural networks in the LIGO/Virgo third

observation period", *Physical Review D*, no. 8, Apr. 2023, Available: https://doi.org/10.1103/PhysRevD.107.082003

[108] Wei Wei *et al.*, "Deep Learning with Quantized Neural Networks for Gravitational-wave Forecasting of Eccentric Compact Binary Coalescence", *The Astrophysical Journal*, Sep. 2021, Available: https://doi.org/10.3847/1538-4357/ac1121

[109] G. Baltus, J. Janquart, M. Lopez, A. Reza, S. Caudill, and J. R. Cudell, "Convolutional neural networks for the detection of the early inspiral of a gravitational-wave signal", *Physical Review D*, no. 10, May 2021, Available: https://doi.org/10.1103/PhysRevD.103.102003

[110] G. Baltus, J. Janquart, M. Lopez, H. Narola, and J.-R. Cudell, "Convolutional neural network for gravitational-wave early alert: Going down in frequency", *Physical Review D*, no. 4, Aug. 2022, Available: https://doi.org/10.1103/PhysRevD.106.042002

[111] G. Baltus, J.-R. Cudell, J. Janquart, M. Lopez, S. Caudill, and A. Reza, "Detecting the early inspiral of a gravitational-wave signal with convolutional neural networks", 2021. Available: https://doi.org/10.1103/10.1109/CBMI50038.2021.9461919

[112] X. Fan, J. Li, X. Li, Y. Zhong, and J. Cao, "Applying deep neural networks to the detection and space parameter estimation of compact binary coalescence with a network of gravitational wave detectors", *Science China Physics, Mechanics & Astronomy*, no. 6, Feb. 2019, Available: https://doi.org/10.1007/s11433-018-9321-7

[113] W.-H. Ruan, H. Wang, C. Liu, and Z.-K. Guo, "Rapid search for massive black hole binary coalescences using deep learning", *Physics Letters B*, Jun. 2023, Available: https://doi.org/10.1016/j.physletb.2023.137904

[114] P. G. Krastev, "Real-time detection of gravitational waves from binary neutron stars using artificial neural networks", *Physics Letters B*, Apr. 2020, Available: https://doi.org/10.1016/j.physletb.2020.135330

[115] Alberto Iess, Elena Cuoco, Filip Morawski, and Jade Powell, "Core-Collapse supernova gravitational-wave search and deep learning classification", *Machine Learning: Science and Technology*, May 2020, Available: https://doi.org/10.1088/2632-2153/ab7d31

[116] M. L. Chan, I. S. Heng, and C. Messenger, "Detection and classification of supernova gravitational wave signals: A deep learning approach", *Physical Review D*, no. 4, Aug. 2020, Available: https://doi.org/10.1103/PhysRevD.102.043022

[117] M. López, M. Drago, I. Di Palma, F. Ricci, and P. Cerdá-Durán, "Deep learning algorithms for gravitational waves core-collapse supernova detection", 2021, pp. 1–6. Available: https://doi.org/10.1109/CBMI50038.2021.9461885

[118] M. Lopez, I. Di Palma, M. Drago, P. Cerda-Duran, and F. Ricci, "Deep learning for core-collapse supernova detection", *Physical Review D*, no. 6, Mar. 2021, Available: https://doi.org/10.1103/PhysRevD.103.063011

[119] Seiya Sasaoka *et al.*, "Deep Learning Application for Detecting Gravitational Waves from Core-Collapse Supernovae", 2023. Available: https://doi.org/10.1103/10.22323/1.444.1499

[120] S. Sasaoka *et al.*, "Deep Learning for Detecting Gravitational Waves from Compact Binary Coalescences and Its Visualization by Grad-CAM", 2023. Available: https://doi.org/10.22323/1.444.1498

[121] T. Marianer, D. Poznanski, and J. X. Prochaska, “A semisupervised machine learning search for never-seen gravitational-wave sources”, *Monthly Notices of the Royal Astronomical Society*, no. 4, Feb. 2021, Available: https://doi.org/10.1093/mnras/staa3550