UNIVERSITY OF PUERTO RICO
RIO PIEDRAS CAMPUS
FACULTY OF NATURAL SCIENCES
DEPARTMENT OF MATHEMATICS

Deep Learning For Animal Audio Classification - A Foundational Approach

By

Rafael Meléndez Ríos

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE IN MATHEMATICS
AT THE UNIVERSITY OF PUERTO RICO

DATE: July 18, 2022

APPROVED BY THE MASTER OF SCIENCE ADVISORY COMMITTEE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE IN MATHEMATICS
AT THE UNIVERSITY OF PUERTO RICO

ADVISOR:

Carlos Corrada Bravo, Ph.D.
Professor
Department of Computer Science

READERS:

Mariano Marcano Velazquez, Ph.D.
Professor
Department of Computer Science

Remi Megret Laboye, Ph.D.
Associate Professor
Department of Computer Science

「花に鳴く鶯、水にすむ蛙の声を聞けば、生きとし生けるものいづれか歌を詠みざりける」-古今和歌集

”Listening to the bush warbler singing in the flower, and the voice of the frog that lives in water, which of all living things does not recite poetry?” - Kokinwakashu

Abstract

Despite impressive successes in computer vision, deep learning has made more limited progress for audio data, mostly in voice recognition and speech processing. Furthermore, most such work has depended on spectrograms and the subsequent adoption of computer vision standard practices, thus treating spectrograms as images and adopting model architectures and augmentation techniques that have been principally developed for image data. In our work we aim at laying the first stepping stones towards a foundation for deep learning for audio data, and in particular for animal sound classification under scarce data conditions. To do so, we handle four basic concerns, using raw input audio data and convolutional neural network models: 1) how to determine hyper-parameters such as filter size and sequence; 2) what data augmentation techniques work best; 3) how do models trained on raw input compare to spectrogram trained models; 4) how do model, augmentation, and input choices impact computer memory and time resource requirements. We corroborate common assumptions that more data and deeper models improve performance, validate the common practice of increasing filter size with depth. We also discover that increasing maxpool filter sizes boosts performance. Our augmentation experiments suggest that all audio data is invariant to time stretching, while only some data types are invariant to pitch shift, notably human speech. However, noise injection (random perturbation and Gaussian noise) did not improve performance. We conclude that models trained on raw inputs can achieve comparable or better results than spectrogram trained

models when data is structured and simple, while spectrogram models still outperform raw input models when data is complex or noisy/irregular. Additionally, using spectrograms allows for shallower models and shorter training time.

Acknowledgments

I would like to thank my professors for their guidance, and my family and friends for their support.

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep Learning has had tremendous successes in the areas of computer vision (CV), developing into its own subfield. In the realm of audio data, however, the story of deep learning is more uneven, with significant successes in voice recognition, but more varied performances across other types of audio data such as music, environmental noise, and animal sounds, as well as diverse tasks such as classification and generation. Additionally, much of the work done in audio, unlike vision, does not use as inputs the raw recorded values, but rather the result of one or more various feature extraction methods, such as Fourier transforms and mel-spectrograms. Spectrograms are particularly popular, since they allow practitioners to use image-like inputs and apply data augmentation (DA) techniques designed for images. More broadly, deep learning for audio seems to lack the same level of development as a subfield, including the lack of standard datasets for benchmarking models (comparable to MNIST or ImageNet), or best practices guidelines for architecture development and data augmentation choice, particularly for our case of classification for animal sounds and scarce data availability.

In our work we aim at laying the first stepping stones towards a foundation for deep learning for audio data, and in particular for animal sound classification. To do so, we explore four key concerns: (1) how to determine model hyper-parameters; (2) what data augmentation techniques to use; (3) how does the use of raw inputs compare with spectrogram inputs; and (4) how do model, augmentation, and input type choices impact resource requirements in terms of memory and train time.

(1) During a preliminary phase of experimentation, we trained various convolutional neural networks imitating the hyper-parameter settings found in successful models for CV (*e.g.*, filter number and sizes, layer sequences) for a dataset of local fauna in Puerto Rico. Results, however, were poor, and we thus determined to run a series of experiments exploring the impact of hyper-parameter settings for raw audio inputs. These experiments were done on a simpler dataset, AudioMNIST [5], containing recordings of spoken digits. It contains ample data that can be subsampled to simulate scarce data availability. Its labels represent very structured, regular signals, and is therefore of low complexity, making it an ideal stepping stone - any model that fails on this dataset should not be able to succeed on more complex ones. We then used the results of these experiments to design the models used in subsequent experiments. We note that no model is particularly fine-tuned for any dataset, since our goal is not to obtain the best result possible using all the tricks of the trade, but rather to uncover underlying principles that should guide the choice of neural network architectures and data augmentation techniques for animal sound classification using raw data.

(2) Our main set of experiments explores the impact of increasing model depth, using DA techniques (pitch shift, time stretch, random perturbation, Gaussian noise), and varying maxpool layer filter sizes, for three different datasets. DA is particularly important for small datasets, which is often the case with animal sound data. We confirm the conventional wisdom that deeper is better though up to a point, at least in the absence of any model- and dataset-specific fine-tuning. We identify time stretch as a useful DA technique across all datasets, and show varying or null results for the remaining techniques. Surprisingly, we show that increasing the filter size in maxpool layers can considerably improve model performance.

(3-4) In our final set of experiments, we train models using spectrograms and show that for simpler datasets, raw input models achieve comparable results, but that for more complex datasets spectrogram models still perform best. Finally, we consider model performance in light of memory and train time restrictions and conclude that spectrograms are best in the low memory and train time regime, but raw inputs (with or without DA) are preferred for higher memory and train time regimes for simpler datasets.

We use three different datasets that are sufficiently distinct and thus allow us to arrive at generalizations regarding performance behavior. The first one is the previously mentioned AudioMNIST, and is characterized by low data complexity due to the regular structure of the classes (reliably predictable sequence of phonemes that comprise each digit word) [5]. The second is XenoCanto, a crowdsourced dataset of bird calls [21]. It contains ample data but is quite noisy with recordings of variable lengths, containing environmental noise, and potentially unreliable labeling provided by a community of experts and amateurs alike. Lastly, the third is a dataset of local fauna in Puerto Rico, with recordings of various species including birds, frogs, and crickets [1]. Its data complexity is the highest as it contains animal calls produced by diverse kinds of biological hardware. It is a small dataset with reliable, expert-assigned labels for each five second segment of audio, but contains environmental noise.

In our work, we restrict ourselves to a small-data regime, since labeled datasets for animal sound classification are in general small, due to the labor-intensive process needed to generate them. This poses an extra challenge, since we cannot leverage big datasets, nor can we easily extract new instances for all species, even for the purpose of unlabeled pre-training. This makes identifying effective data augmentation methods key, as they provide the main ways to increase training set size when it is not feasible to obtain more data. This is thus the aspect that forms the center of our current research.

1.1 Previous Work

Data Augmentation encompasses a wide variety of techniques intended to exploit the presence of invariant features in data in order to increase sample size, particularly when data is scarce. It has been widely used in the area of Computer Vision and was part of the successful strategy that led AlexNet to make breakthrough progress in the ImageNet problem [24]. The authors in [37] provide a summary of data augmentation (DA) techniques for images, as well as a framework for categorizing them along two axes: data warping VS oversampling, and image manipulation VS deep learning. Data warping includes many common geometric techniques such as shifting, flipping, cropping, rotation, and translation, as well as a broader class of image manipulation techniques, including kernel filters and random erasing, alongside deep learning based techniques such as adversarial training and style transfer. Oversampling includes one image manipulation technique, image mixing, and the deep learning techniques of feature space augmentation and synthetic data generation with adversarial models. Although this survey is unrelated to audio data, it serves as a reference and lens through which to think about and classify DA techniques for audio, and consider which should be applicable and under which circumstances.

The most recent relevant work to our research is found in [30], which explores DA approaches for audio animal classification. However, they employ four "protocols" each composed of a different group of techniques, and then apply them to two single-animal datasets (*i.e.*, a cat dataset and a bird dataset). Furthermore, results were overall modest or comparable with improvement due to greater model complexity. The authors of [35] identify five techniques of DA and evaluate the performance of each one on a simple CNN model and then compare it with previous, standard models like SKM, using the UrbanSound8K dataset. The results showed modest improvements in accuracy, and, more importantly, showed how different DA methods affect different sound objects differently, suggesting possible benefits from class-dependent application. They provide a simple and clear methodology of evaluating the impact of DA methods centered on the difference in training metric with and without augmentation, and we will imitate their methodology in our experiments.

Additionally, the authors of [41] distinguish DA techniques that can be applied to time-domain (T) (*e.g.*, Gaussian noise injection, window cropping), frequency-domain (F) (*e.g.*, amplitude and phase perturbations), and T-F domain (*e.g.*, local averaging), many of which have been mentioned in previous papers. They also mention signal decomposition-based methods that can be used to exploit seasonality (*i.e.*, periodic trends) in the data, which can be helpful in identifying the calls of certain animals. The authors of [23] compare Vocal Tract Length Perturbation (VTLP), introduced in [19] as a DA technique for warping voice audio, with their proposed speed-perturbation with resampling technique for Speech Recognition and claim that the latter is an efficient approximation of VTLP with time-warping. The authors of [32] propose a simple augmentation protocol for Automatic Speech Recognition, combining time warping, frequency masking, and time masking. Of particular interest is the fact that their analysis shows how DA helps to tackle the problem of overfitting the data, moving in the direction of underfitting - it increases validation accuracy at the expense of train accuracy. Finally, the authors of [8] use Reinforcement Learning to propose an automated learning algorithm called AutoAugment, which optimizes the data augmentation applied by randomly selecting for each sample a data augmentation technique from a fixed set and a corresponding magnitude, with the aim to learn the optimal policy of technique + magnitude parameter for a given dataset. Such an approach would be useful to determine class-specific DA techniques.

We will not concern ourselves much with deep learning approaches to augmentation, but it is worth noting the following. The authors of [9] propose DA for learned feature spaces, first learning a feature representation of data by training an autoencoder, and then adding noise to the encoded representations and decoding to generate new synthetic samples. However, the authors of [37] suggest that, at least for images, augmentations at the raw input are preferable to those at the feature space, though the technique may be useful if coupled with disentanglement [18]. The authors of [2] propose DAGAN, or Data Augmentation Generative Adversarial Network, which leverages the generative model learned by a trained GAN to generate new synthetic samples similar to those produced by the real generating distribution. This is particularly helpful when training on a related but different dataset whose samples contain many or all of the same underlying (invariant) features.

Although the experiments in [23, 30, 32, 35] showed improvements with DA, they are mostly modest. Thus, after an initial review of the literature, it seems an open question whether DA techniques help performance in a significant way for audio data, and how such improvements compare with obtaining more data. Most successful applications of DA have been observed in computer vision problems, such as object classification in AlexNet [24]. Furthermore the experiments of [23, 30, 32, 35] all use spectrograms as input data, emulating the Computer Vision problems. Although raw sound data has been used as inputs for other tasks such as music generation (*e.g.*, WaveNet [31]), as far as we are aware, we are the first to compare DA methods for raw data inputs for the task of animal audio classification and other similar applications.

Chapter 2

Theoretical Background

2.1 Machine Learning - an Introduction

A canonical definition for machine learning (hereafter ML) comes from Tom Mitchell where it is described as the process in which a computer program learns, and thus improves, its performance on tasks T by leveraging its experience E on T, as quantified by some performance measure P [26]. Those well versed in ML will recognize how fitting this definition is, but also perceive a strong influence from the field of artificial intelligence and much affinity with the ML approach of reinforcement learning. It serves as a good abstraction analogous to the human experience of learning, which helps the practitioner think clearly when defining a problem to solve, yet reveals little about what ML involves in practice.

For a description of the practice of ML, we turn to Murphy [28] and Goodfellow et al [14]. Murphy, focusing on the algorithmic, describes it as a set of pattern detection methods that help us extract patterns useful for such tasks as future data prediction or decision making under uncertainty [28]; while Goodfellow, focusing on the data, describes it as a "form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions and a decreased emphasis on proving confidence intervals around these functions" [14]. Thus ML, in its current form, centers on automated processes, wherein the computer program itself uses an algorithm to determine the rules it uses to complete some task. And these algorithms exploit statistical properties of data, such as correlations, in order to detect patterns that are useful to the program in its attempt to understand and encode the relevant information of the data generating distribution. We follow Murphy and Goodfellow for our treatment of ML below.

2.1.1 Key Concepts and Definitions

Though the number of tasks T for which ML can be used is infinite, they are each a particular example of some broader class of problems. In ML, we typically classify problems as falling into one of two categories: supervised learning or unsupervised learning. In supervised learning, we concern ourselves with predictions: we are given a dataset of inputs X, known as features or covariates, with corresponding outputs Y, known as the response variable, that we want to predict. It is called supervised because programs, after making predictions, are provided with the correct answers or feedback, *i.e.* Y, so that they can assess their own performance and make any necessary corrections - similar to students supervised by an instructor. Formally, it is a problem of function approximation, in which we assume that there is some unknown function mapping inputs to outputs, $f(x) = y$. Our goal is to find a suitable approximation to f, namely $\hat{f}(x) = \hat{y}$, such that $\hat{y}$ is equal to or close enough to y.

Supervised learning comes in two general flavors. If the response variable is a real number or a value within a continuous range of ordered values, then it is a regression problem. If the response variable is categorical, i.e. a value from a fixed set of finite unordered values, then it is a classification problem. The latter is often subdivided based on whether inputs map to one of either two classes (binary classification) or more than two classes (multiclass classification), or if inputs can belong to more than one class (multi-label classification). More broadly, though, problems with structured outputs (available at train time) will usually fall under the umbrella of supervised learning.

In contrast, there are only inputs in unsupervised learning and no response variable or feedback. Here, the concern is descriptive: we want the computer program to discover patterns, or hidden structures, in the data that may be useful for some other task to be performed later. It is thus also referred to as knowledge discovery. Formally, it is a problem of density estimation, in which we assume there is some data generating distribution defined by an unknown density $p(x|\theta)$ with unknown parameters θ. Our goal is to capture the structure inherent in the unknown density $p(x|\theta)$, either directly by estimating the density with $\hat{p}(x|\hat{\theta})$, or indirectly through heuristics or approximation methods.

Two canonical types of unsupervised learning problems are clustering and dimension reduction. Clustering aims to uncover groups or clusters of data points which are more similar to each other within a cluster than to data points from other clusters. It involves the dual goals of determining how many clusters there could be in the data, *i.e.*, estimating the density $p(K|D)$ for the number of cluster K given the data D, as well as what the cluster label for each data point x_i in D could be, i.e. $p(z_i = k|x_i, D)$ for label $k \in K$. This approach is commonly used for building customer type profiles, where companies want to identify types or clusters of customers in order to better target ads or products. Dimension reduction aims at projecting the data points, each assumed to be a multi-dimensional vector, into a lower dimensional subspace that best captures the defining characteristics of the data. The intuition here is that "there may only be a small number of degrees of variability, corresponding to latent factors" that actually determine the shape and nature of the data [28]. An example of this approach is the use of latent semantic analysis (LSA), which takes a dataset of documents and leverages the correlation statistics of words within documents to extract a set of defining topics within the dataset or to calculate document similarity measures [28].

Besides differences in goal, unsupervised and supervised learning can be distinguished by the type of experience E they are exposed to. Both approaches expose the algorithms to the entire dataset, but the dataset for the former is composed solely of features $X_i, D = X_{i}{}_{i=1}^{n}$, while for the latter it is composed of features X_i paired with some value or vector $Y_i, D = (X_i, Y_i)_{i=1}^{n}$. In probabilistic terms, unsupervised learning attempts to learn the distribution $p(X)$ or interesting properties of said distribution, while supervised learning attempts to estimate $p(Y|X)$. That said, this distinction is in fact not formally fixed, and probability can, in most cases, be used to exchange between the two, as explained by [14]. For unsupervised learning, we can use the chain rule of probability to decompose $p(X) = \prod_{j=1}^{n} p(x_j|x_1, \ldots, x_{j-1})$, where each x_j is an entry in the vector X, thus transforming an unsupervised problem into n supervised learning problems (*i.e.*, predicting x_j given the previous $(x_1, \ldots, x_{j-1})$ features). Likewise, for supervised learning, we can first use an unsupervised method to analyze (X, Y), treating Y as just another feature(s), and then using the definition of conditional probability to infer $p(Y|X)$. Thus, theoretically, we can navigate between the approaches.

There is a third approach in ML, known as reinforcement learning (RL), particularly popular in the field of AI. We will not concern ourselves with RL in this work, but a description is included for the sake of completion. This approach involves an agent/program that participates in a reward system: it observes an environment, takes an action, and receives feedback from the environment (known as reward). The goal of the agent is to receive as inputs the data generated by this decision process and to learn a policy or strategy on how to take actions, in order to maximize its reward. RL has achieved tremendous successes in recent years, as seen in DeepMind's AlphaGo and other similar projects.

Machine learning methods can also be divided with regards to their use of parameters: parametric versus non-parametric. Parametric models, including the ones used in our research, make the assumption that the information necessary to perform a task can be captured in a fixed number of parameters. This assumption about the nature of the training data is called the model's inductive bias [28]. The parameters define the model and are learned during training time. This makes the models faster to use, but slower to train. Non-parametric models, on the other hand, do not have a fixed number of parameters. Rather, the number of parameters grows with the amount of training data available, as we can see, for example, in the K-nearest neighbors (KNN) algorithm. These models are more flexible and require little or no training time, but are often too costly to compute for large datasets [28].

2.1.2 Evaluation

When evaluating ML models on tasks T, there are various ways to measure the performance P, depending on the nature of the problem and one's goals. For unsupervised learning, there is no clear, objective way to measure performance, since there are no ground truths with which to compare the results. Often, the outputs of such models are evaluated based on their usefulness as inputs for another, generally supervised, task performed later. Alternatively, various models can be trained and compared with each other. For example, various clustering models can be trained, and then compared based on how homogeneous the clusters in each model are, and how distinct members of different clusters are.

In the supervised case, one can treat the loss function as a metric, preferring models that achieve the lowest loss. In linear regression, where the loss is the mean squared error, or MSE, this is often the case, particularly because it has a clear interpretation: it measures the distance between the model's predictions and the actual target values given the inputs. For classification problems, as we shall see below, choosing a metric is not as straightforward. The loss function is not as interpretable as in the linear regression case, and given one's goals, there may be alternative measures one might want to use to compare different models. These alternative methods are often hard or impossible to differentiate, and thus are not used for training.

2.2 Supervised Learning and Classification

We now focus on supervised learning, giving an overview of the concepts relevant to our research. We start with regression problems to motivate the most widely used model called linear regression, and then show its extension for classification problems in the form of logistic regression, a type of generalized linear model. We will follow [20] and [28] in what follows.

2.2.1 Regression - an Overview

Let's consider the basic regression case, with data $\{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, and $x^{(i)}, y^{(i)} \in \mathbb{R}$. One simple way to predict $y^{(i)}$ from $x^{(i)}$ is to assume that there is a linear relation between $x^{(i)}$ and $y^{(i)}$: that is, that x times some factor w, plus or minus some constant b, is approximately $y^{(i)}$. The resulting model would be

$$\hat{y}^{(i)} = f(x^{(i)}) = wx^{(i)} + b \tag{2.1}$$

where the machine would have to learn the optimal w and b such that $y^{(i)} \approx f(x^{(i)})$ for any pair $(x^{(i)}, y^{(i)})$. But this is precisely the equation of a line, and thus called a linear model. This simple idea can be extended to situations where $x^{(i)}$ is a vector, by predicting $y^{(i)}$ with a weighted linear combination of the entries of $x^{(i)}$,

$$f(x^{(i)}) = \sum_{j=1}^{d} w_j x_j^{(i)} + b = \mathbf{w}^T \mathbf{x}^{(i)} + b \tag{2.2}$$

where each w_j can be interpreted as the contribution or relevance of feature $\mathbf{x}_j^{(i)}$ in predicting $y^{(i)}$. A positive $\mathbf{w}_j$ indicates a positive correlation between $\mathbf{x}_j^{(i)}$ and $y^{(i)}$, and a unit increase in the value of $\mathbf{x}_j^{(i)}$ would correspond to an increase of $\mathbf{w}_j$ in $y^{(i)}$. Similarly a negative $\mathbf{w}_j$ indicates a negative correlation and decrease of $\mathbf{w}_j$ in $y^{(i)}$ with each unit increase in $\mathbf{x}_j^{(i)}$.

For training a linear regression model, we use the mean squared error (MSE) formula and optimize for the parameters $(\mathbf{w}, b)$ that minimize it:

$$argmin_{(\mathbf{w},b)} \frac{1}{N} \sum_{i=1}^{N} (f(\mathbf{x}^{(i)}) - y^{(i)})^2 \tag{2.3}$$

The MSE is differentiable with respect to the parameters, and furthermore provides us with closed form solutions, making for easy implementations of many optimization methods such as gradient based ones.

2.2.2 Logistic Regression

It would be convenient to extend this simple and powerful model for classification, but categorical response variables present a significant hurdle to overcome. In our case, for example, we can ascribe each class an integer value from 0 onwards and treat them as if they were real numbers, but this would produce an ordering that is meaningless. Thus, if we labeled coqui as 0, parrot as 1, and cricket as 2, we would be implying that the difference or distance between the songs of a coqui and a parrot are the same as that between those of a parrot and a cricket, and furthermore that coquis and crickets are somehow more different than either is to the parrot. Which of course, does not make sense.

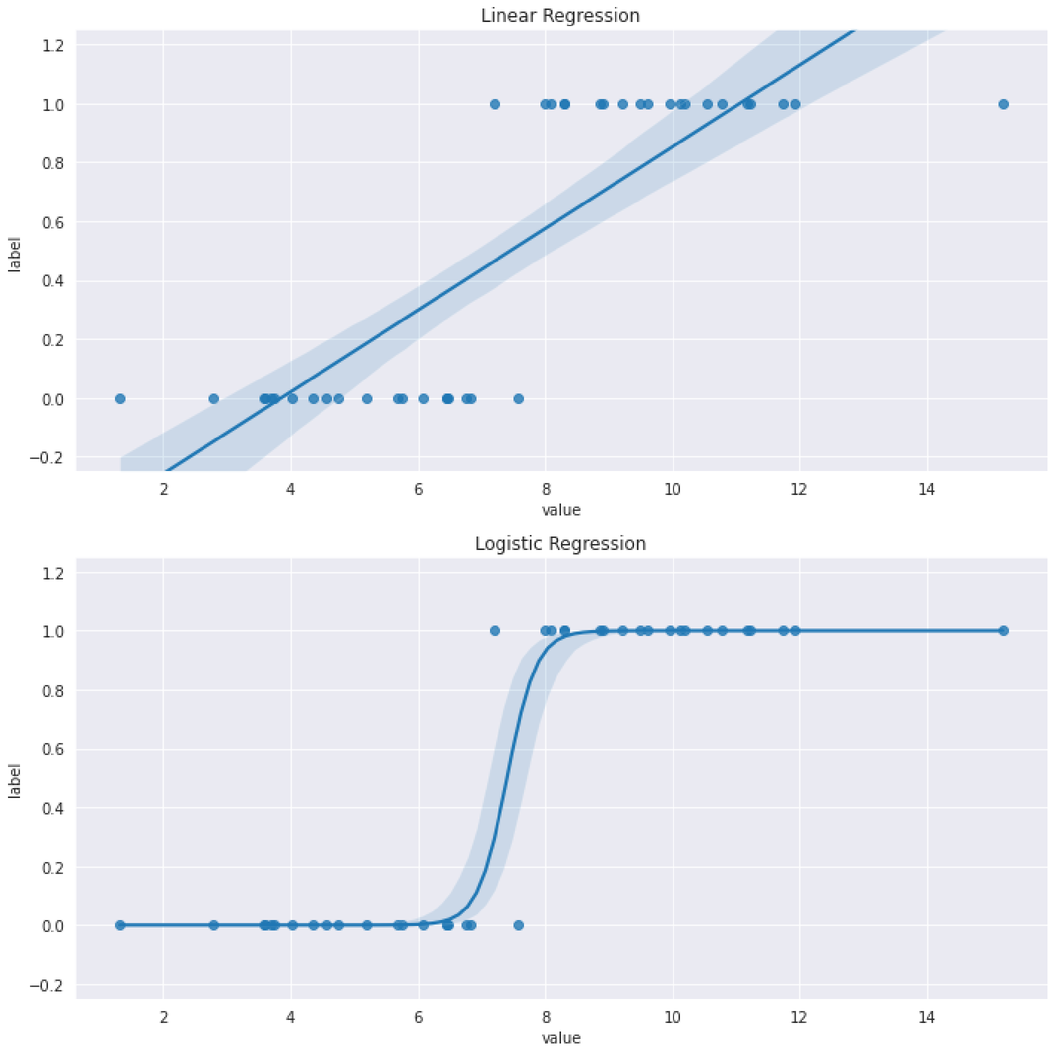


Figure 2.1: Linear (left) versus logistic (right) regression models for synthetic data sampled from Gaussian distributions $\mathcal{N}(5, 4)$ for label 0 and $\mathcal{N}(10, 4)$ for label 1

We could, however, limit ourselves to the binary case and use linear regression, establishing some threshold value, say 0.5, used to predict whether a sample belongs to class 0 or 1 (alternatively, whether some categorical condition is true or not, *e.g.*, the presence of a disease). This approach works, but as can be seen in Fig 2.1, it results in predictions that are not interpretable. Furthermore, given a plot of the data, it is clear that a line cannot be properly fitted in the same way as in regression.

A solution to this problem is then to stop trying to model the output y itself, and instead "model the probability that y belongs to a particular category" [20]. This can be done by using a function that can map any real number to the open interval $(0, 1)$. For binary classification, this is done with the logistic function,

$$\sigma(\mathbf{w}^T\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w}^T\mathbf{x})} \tag{2.4}$$

and hence results in the model called (binary) logistic regression. By inputting the weighted linear combination $\mathbf{w}^T\mathbf{x} + b$ into the sigmoid, we obtain a function that allows us to approximate a sample's probability of belonging to one of two classes. We then determine a threshold to decide the appropriate boundary between the two classes, as per our preferred evaluation metric.

Unlike with linear regression, the relation between explanatory variables $\mathbf{x}$ and the response variable in binary logistic regression is not linear and thus it is no longer clear how to interpret the weights (or parameters) $\mathbf{w}$. Thus, the quantities defined as odd and log-odds are used. The odds is a quantity defined as $\sigma(z)/(1 - \sigma(z)) = \exp(z)$, which lies in the interval $(0, \infty)$, *i.e.*, the ratio of the predicted probability of class 1 (or winning, when used in gambling and betting settings) over the predicted probability of class 0 (or losing). Since this ratio simplifies to a basic exponential form, we can now see that a unit increase in feature $\mathbf{x}_j$ will multiply the odds by a factor of $\exp(\mathbf{w}_j)$. If we apply the log function to the odds formula, we obtain the log-odds quantity which is simply $\mathbf{w}^T\mathbf{x}$, which allows us to establish a linear relation between features and model output. Thus, a unit increase in $\mathbf{x}_j$ results in an increase/decrease of $|\mathbf{w}_j|$ in the model output[1], when $\mathbf{w}_j$ is positive/negative respectively.

For training a logistic regression model, the Maximum Likelihood Estimator (MLE) is used instead of the MSE. The intuition behind the use of MLE is that when training, we would like the method to find weights that output a predictive probability close to 1 or 0 for samples labeled 1 or 0 respectively. This is equivalent to finding weights that maximize the likelihood of the samples given the model (each sample assumed independently drawn):

$$L(\mathbf{w}) = \prod_{i=1}^{N} [\sigma(\mathbf{w}^T\mathbf{x}^{(i)})^{y^{(i)}} \times (1 - \sigma(\mathbf{w}^T\mathbf{x}^{(i)}))^{(1-y^{(i)})}] \tag{2.5}$$

For any given datapoint $x^{(i)}$, the term inside the product reduces to $f(x^{(i)})$ if $y^{(i)} = 1$, or $(1 - f(x^{(i)})$ if $y^{(i)} = 0$. In order to maximize the $L(\mathbf{w})$ across all datapoints $x^{(i)}$, the model will then learn parameters $\mathbf{w}$ such that $f(x^{(i)})$ is as close as possible to $y^{(i)}$, *i.e.*, $[0, 1]$. Thus, by maximizing $L(\mathbf{w})$, we obtain the desired behavior of outputting values akin to predictive probabilities.

In practice, the negative log-likelihood (NLL) is used, which helps avoid producing 0 as a result of the product of many small values, as well as to approach the task as

[1]Recall that logistic regression outputs a probability, and thus is not equivalent to the response variable; it is more a proxy for said variable

one of minimizing the loss or cost function:

$$argmin_{(\mathbf{w},b)}NLL(\mathbf{w},b) = \sum_{i=1}^{N}[y^{(i)}\log(\sigma(\mathbf{w}^T\mathbf{x}^{(i)}))+(1-y^{(i)})(1-\log(\sigma(\mathbf{w}^T\mathbf{x}^{(i)})))] \quad (2.6)$$

Another name for this loss function is the cross-entropy error function. Here, as in the case of linear regression, a variety of optimization methods can be used, such as descent methods. We will say more on this aspect when discussing deep learning models in the next section.

2.2.3 Multi-class Case

When there are more than two classes, the above model can be modified in the following way. Define a set of weights $\mathbf{w}_c$ for a given class $c \in \{0, 1, 2, \ldots, K-1\}$, where K is the total number of classes. Imagine you only wanted to predict samples from class c; then you could treat it as a problem of 1-vs-all, thus reducing it to the binary case. For convenience, let us take the odds function for class c, namely $\exp(\mathbf{w}_c^T\mathbf{x})$. Now if we do the same for all classes, we will have a value for the odds of each versus the rest collectively. These values, of course, will not add up to 1, and we want our model to output a probability distribution across all possible classes, in imitation of the binary case. Thus we normalize by dividing each odds ratio by the sum of all odds ratio, deriving what is called the softmax function:

$$p(y = c|\mathbf{x}, W) = \frac{\exp(\mathbf{w}_c^T\mathbf{x})}{\sum_{k=0}^{K-1}\exp(\mathbf{w}_k^T\mathbf{x})} \quad (2.7)$$

where W is the matrix of weights and $\mathbf{w}_c$ is the c^{th} column associated with class c. This in turn will have the following negative log-likelihood that can be used for training:

$$NLL(W) = -\sum_{i=1}^{N}\left[\left(\sum_{k=0}^{K-1} y_k^{(i)} W_k^T \mathbf{x}^{(i)}\right) - \log\left(\sum_{k=0}^{K-1} \exp(W_k^T \mathbf{x}^{(i)})\right)\right] \tag{2.8}$$

as noted on page 255 of [28].

2.2.4 Measuring Model Performance

Before moving on to the topic of Deep Learning, we would like to discuss evaluation metrics. After a model is trained, its performance is measured on a separate, previously unseen subset of samples called the test set. There are many metrics available for this evaluation step, starting with test loss, wherein the test samples are fed into the model to then calculate predictions and the corresponding value of the loss function. This is common practice for linear regression, but is often replaced by several other possible metrics in classification problems, depending on the particular goals of a project and the desired performance interpretation among others. A standard metric for classification is Accuracy, defined as the percentage of correctly labeled samples. For the binary case, this is done by first using the desired threshold and classifying every sample with predicted probability greater than it as 1 and all others as 0, and then calculating the percent of samples correctly classified. For multinomial regression, predict instead the class with the largest predicted probability. This is the one used in previous works concerning our research problem and is thus the one we use to report results.

Other popular metrics may be more suitable, in particular for the binary case, such as for example: Precision, Recall, and F_1 score. If we first define correctly labeled (*i.e.*, "true") positives and negatives as TP and TN, and incorrectly labeled (*i.e.*, "false") positives and negatives as FP and FN, then precision is defined as $\frac{TP}{TP+FP}$. It measures the percentage of positive predictions that the model correctly made. Note that this metric shows preference for making good positive predictions, regardless of what happens with negative predictions; thus it is possible to obtain a high precision while incorrectly predicting many positives as negative. This can be obtained by setting the threshold close enough to 1. Recall is defined as $\frac{TP}{TP+FN}$, and measures the percent of samples with positive ground truth the model correctly labeled. It is ignorant of negative samples, and a perfect recall can be obtained by setting the threshold to 0. These two metrics in turn can be combined in order to find a balance between their tendencies. This is done so through the last metric we will discuss, the $F1$ score, defined by $\frac{1}{F_1} = \frac{1}{Pr} + \frac{1}{Re}$, or alternatively $F_1 = \frac{Pr \cdot Re}{Pr+Re}$. It is also called the harmonic mean of precision and recall. Note that choosing a threshold close to 1 will maximize precision but decrease recall (by increasing FN), counteracting each other in the numerator of F_1; meanwhile choosing a threshold at 0 will maximize recall but make precision 0, thus making $F_1 = 0$ (or undefined based on the first formula). F_1, then, incentivizes a threshold choice that harmonizes between the two.

2.3 Deep Learning

Deep Learning is an approach to machine learning that attempts to train a computer by '(enabling) the computer to build complex concepts out of simpler concepts' which it then uses to successfully complete a particular task [14]. Inspired by the biological brain, deep learning models receive a set of inputs which they pass through layers of 'neural' or hidden units to produce the desired output. In each layer, each neural unit processes a subset of outputs from the previous layer using a nonlinear function, and then produces an output, thus imitating how neurons receive and process inputs from other neurons in order to decide whether to activate or not, *i.e.*, propagate a signal forward. Such a sequence of layers is called an artificial neural network (ANN), or simply neural network (NN).

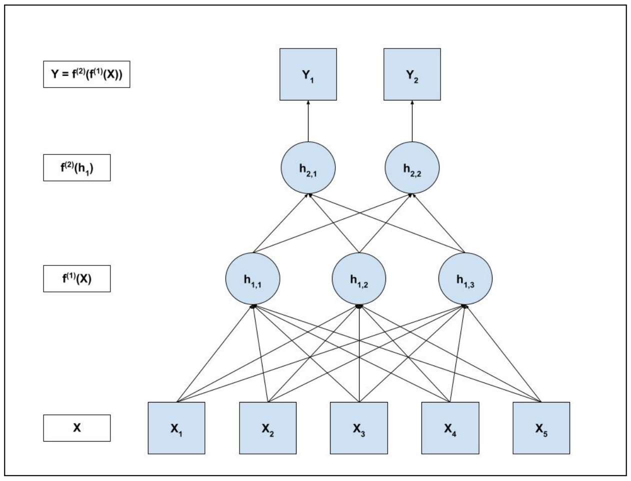


Figure 2.2: Example of a two layer NN represented as a DAG

The label of 'network' comes from the fact that we can represent a NN as a composition of many functions, with each layer k being a function

$$f^{(k)} : \mathbb{R}^{m_k} \to \mathbb{R}^{n_k},$$

where m_k and n_k are the input and output dimensions for the k^{th} function. This composition can then be described with a directed acyclic graph (DAG), as seen in Fig 2.2. Equivalently, we can write it out as

$$f^{(L)}(f^{(L-1)}(\ldots(f^{(1)}(x)\ldots)),$$

where L is the number of hidden layers plus output layer.

There are three principal types of NNs that are in common use. The first such type is the deep feedforward network, or multilayer perceptron (MLP). As the most basic type, we will discuss it to elaborate basic concepts and important aspects of building and training NNs. Then, there is the convolutional neural network (CNN), which can be viewed as a special kind of MLP. This is the model we use in our research. Finally, there is the recurrent neural network (RNN). We will not discuss RNNs, but suffice to say that they differ from MLPs and CNNs in that, while the latter two only allow information to flow forward - from input to output - RNNs allow for feedback connections so that information flowing within the network can flow backwards. We closely follow [14] and summarize key concepts and principles.

2.3.1 The Multilayer Perceptron (MLP)

As mentioned earlier, the most basic type of neural network is an MLP. The reason for this name is that they are, in effect, a stack or chain of perceptrons. The perceptron is a model or unit that takes a set of inputs, performs a linear combination on them, and then applies a nonlinear function called the activation function to produce an output. When the perceptron applies the logistic function, it is equivalent to a logistic regression model/unit. In imitation of the neuron, the activation function decides whether, given the inputs, the neural unit will “activate” and fire forward the signal(s) received. An MLP is composed of an input layer, an output layer, and one or more internal or hidden layers in between, each with any number of perceptrons as hidden units - thus receiving its name.

Key characteristics used when describing a particular MLP model are the hidden layer configurations, including model depth and width, as well as the type of its output layer. Every MLP has a fixed depth L, defined as the total number of hidden layers plus the output layer. Equivalently, we can define it as the total number of functions in the composition chain. Every MLP has exactly one output layer, while the number of hidden layers, $L - 1$, is determined by the practitioner and is considered a hyperparameter. By convention, the input layer is often called the 0^{th} layer. Similarly, its width is defined as the number of hidden units in a layer, which is also determined by the practitioner via a hyperparameter. Note, though layers can all have the same width, that need not be the case. The output layer has a fixed width, determined by the learning problem, *e.g.*, linear regression or binary/multi-class classification.

Another important choice by the practitioner is the activation function of the hidden units, which are in general nonlinear. It is precisely the activation functions in the hidden units that make the training of a MLP non-convex, and thus different from training more traditional ML models. First, let’s note that the computation of each layer k can be mathematically defined through vector-matrix multiplication followed by the element-wise application of the activation function

$$h^{(k)} := f^{(k)}(W^{(k)}h^{(k-1)} + b^{(k)}) \tag{2.9}$$

where each row j represents the weights of the linear combination for the j^{th} unit, $b_j^{(k)}$ is the scalar bias, and $h^{(k)}$ is the (vector) output of k^{th} layer. We could, in theory, output the linear combinations themselves, thus reducing the computation to

$$h^{(k)} := W^{(k)}h^{(k-1)} + b^{(k)}. \tag{2.10}$$

However, a chain of such linear layers can be rewritten as a single linear layer, with weight matrix W and bias term b defined as:

$$W := W^{(L)} \times \ldots \times W^{(1)} \tag{2.11}$$

$$b := b^{(L)} + \sum_{i=1}^{L-1} W^{(L)} \times \ldots \times W^{i+1}\ b^{(i)} \tag{2.12}$$

where L is the number of layers. Thus the MLP would reduce to the same types of models we saw in our previous discussion of ML.

Recalling the interpretation of ML as function approximation, we can think of MLPs as "function approximation machines that are designed to achieve statistical generalization" [14], which can be used to approximate any kind of function, including nonlinear ones. In traditional ML, nonlinear relations in the data are dealt with by using certain techniques such as the application of kernels[2], or extracting hand-crafted features designed by experts. MLPs, though, and DL in general, opt for dispensing with these traditional steps, and instead attempt to learn a nonlinear transformation of the data itself, through the chain of hidden layers. In effect, DL "gives up on the convexity of the training problem" but in exchange reaps the benefits of using a general enough family of data transformations, letting the model automatically learn features that best help perform the task at hand [14].

Currently, the most popular type of hidden unit is the rectified linear unit (ReLU), which uses the following activation function:

$$g(z) = \max(0, z) \tag{2.13}$$

As a piecewise linear function, we define its derivative dg/dz to be 0 on half its domain, $(-\infty, 0)$, and 1 for the other half, $(0, \infty)$. The advantage of using the ReLU is that, for positive activations, it guarantees a large enough, non-zero, derivative that will allow for continued updating of the weights, even as the magnitude of the activation changes/increases. We will see later the importance of this.

There are many variations of the ReLU, including the Leaky ReLU, which uses the activation function

$$g(z) = \max(0, z) + \alpha \min(0, z) \tag{2.14}$$

[2]not to be confused with kernels in CNNs

with a small $\alpha > 0$ (e.g., $\alpha = 0.01$). The maxout unit, in turn, is considered a generalization of the ReLU, which applies the ReLU activation function to a subset of linear combinations. That is, given the output of the previous layer $h^{(k-1)}$ as input, after calculating the linear combinations for the layer

$$z^{(k)} = W^{(k)}h^{(k-1)} + b^{(k)} \tag{2.15}$$

the entries of $z^{(k)}$ are divided into groups of j values. The ReLU function is then applied to each group, producing the layer output $h^{(k)}$ with size equal to the number of groups.

The sigmoid unit, using the logistic sigmoid function presented in section 2.2, was a very popular hidden unit before the ReLU. However, as the output of the sigmoid increases in magnitude, its derivative approaches 0. This behaviour is referred to as saturation. In such cases (*i.e.*, when the sigmoid unit becomes more "sure" of being active/positive or inactive/negative), its zero derivative will prevent the gradient from propagating down through the model during training, making model training difficult when using gradient based methods. As we shall see, MLPs, and DL models in general, precisely use gradient-based methods. Thus, sigmoid units are no longer recommended for hidden layers. We shall also see later, though, why the logistic sigmoid is still a valid and popular activation function for the output layer.

Finally, another activation function that was also once popular is the hyperbolic tangent, or

$$g(z) = \tanh(z) = 2\sigma(2z) - 1 \tag{2.16}$$

We note that it can be expressed in terms of the logistic sigmoid, and thus the same comments apply. There may be instances in which the sigmoid or tanh functions are preferred for hidden layers, but in general, both are discouraged.

In the output layer, there are three commonly used units: linear, sigmoid, and softmax. Each of these is problem dependent, corresponding to regression, binary classification, and multi-class classification respectively. Let us explore their properties and their impact on loss function choice for training.

Linear units are used to produce real-valued, continuous outputs, as in scalar regression where we want to predict a single value $\hat{y} = f(h) = W^T h + b$ (note that here and in what follows, we use h to mean the output of the penultimate layer, *i.e.*, $h^{(L-1)}$). This is, of course, the equivalent of linear regression, using the output of the last hidden layer as input, and thus, we can train the model with MSE, which is equivalent to using the MLE or log-likelihood (LL). Since linear units do not saturate, they pose no risk of having the derivative vanish to 0.

The sigmoid unit is used for binary classification, since it allows us to predict a Bernoulli output or probability of class 1: $\hat{y} = P(y = 1|h)$. First, let us prove that this is indeed the case, and then discuss why the sigmoid requires we use the MLE (or NLL, to be exact) for training.

Define the input of the sigmoid unit as $z = W^T h + b$. We want to use z to predict whether $y = 1$, and thus for convenience we begin by assuming that the unnormalized log probability[3] is linear w.r.t. z:

$$log\tilde{P}(y=1) = z = yz. \tag{2.17}$$

We can similarly define

$$log\tilde{P}(y=0) = 0 = yz \Rightarrow log\tilde{P}(y) = yz. \tag{2.18}$$

Thus, the more positive or negative z is, the more likely y is 1 or 0 respectively. In order to arrive at an actual, normalized Bernoulli, we simply apply the necessary operations:

$$\tilde{P} = \exp(yz) \Rightarrow P(y) = \exp(yz) / \sum_{y'=0}^{1} \exp(y'z) \tag{2.19}$$

This simplifies to

$$P(y=0) = \frac{1}{1+exp(z)} = \sigma(-z) \tag{2.20}$$

$$P(y=1) = \frac{exp(z)}{1+exp(z)} = \sigma(z) = 1 - P(y=0) \tag{2.21}$$

which is exactly what we wanted.

In order to understand the behavior of the sigmoid w.r.t. to the loss function, we further manipulate the sigmoid equation and connect it with a function called the softplus function: $\zeta(z) = \log(1+e^z)$. These functions are linked by the following:

$$\log\sigma(z) = \log\frac{1}{1+e^{-z}} = -\log(1+e^{-z}) = -\zeta(-z) \tag{2.22}$$

[3]Technically, this and following probabilities should be expressed as conditionals given z

$$\frac{d\zeta(z)}{dz} = \sigma(z) \tag{2.23}$$

Now, as for our sigmoid unit, we return to the sigmoid equation that uses both y and z: trivially, $P(y = 1) = \sigma(yz)$, and $P(y = 0) = \sigma((y-1)z)$. Now note that if we replace $(y-1)$ with $(2y-1)$ in the latter, we get the following:

$$\sigma((2y-1)z)|_{y=0} = \sigma(-z) = P(y = 0) \tag{2.24}$$

$$\sigma((2y-1)z)|_{y=1} = \sigma(z) = P(y = 1) \tag{2.25}$$

If we use the NLL for training, which is normally the case, we obtain the following loss function and derivative w.r.t. z:

$$J = -\log \sigma((2y-1)z) = \zeta((1-2y)z) \tag{2.26}$$

$$\frac{dJ}{dz} = (1-2y)\sigma((1-2y)z) \tag{2.27}$$

In the case of $y = 1$, we can see that as $z \to -\infty$, then $d/dz = -\sigma(-z) \to -1$, and as $z \to +\infty$, then $d/dz = -\sigma(-z) \to 0$. Similarly, when $y = 0$, as $z \to -\infty$, then $d/dz = \sigma(z) \to 0$, and as $z \to +\infty$, then $d/dz = \sigma(z) \to 1$. Thus, when the model confidently predicts the right output, the loss function saturates and its derivative approaches 0, slowing the learning process; else, the more the model confidently makes the wrong prediction, the loss function increases in value and its derivative remains large enough to continue driving the learning process.

If we compare the behavior of the NLL with sigmoid output versus MSE, we notice a very different, and undesirable behavior. Our loss function would then be:

$$J = \frac{1}{2}(\hat{y} - y)^2 \tag{2.28}$$

with $\hat{y} = \sigma(z)$.[4] Then the derivative w.r.t. z would be[5]:

$$\frac{dJ}{d\hat{y}} = (\hat{y} - y)\hat{y}(1 - \hat{y}).$$

Now, as $z \to -\infty$, then $\hat{y} \to 0$ and $d/d\hat{y} \to 0$. Similarly, as $z \to +\infty$, then $\hat{y} \to 1$ and $d/d\hat{y} \to 0$. Thus the more confident the model makes its predictions, the derivative approaches 0, regardless of whether said prediction is correct or not. As long as the model outputs confident predictions, its loss will saturate, leading to a vanishing derivative and a virtual stop to learning.

Unlike the MSE, the MLE, in the form of the NLL, contains a log term that undoes the exponential term of the sigmoid. This is precisely what helps keep the loss from saturating before the model has learned appropriate predictions. And thus, this is precisely why the MLE (or, equivalently, NLL) is preferred over the MSE.

For multi-class classification, we extend the logic of the binary case, taking $\log \tilde{P}(y = c) = z_c$ as the log unnormalized probability for class c, and thus $\tilde{P}(y = c) = \exp(z_c)$. We then use the softmax function, as defined earlier, to obtain a normalized proba-

[4]We use $\sigma(z)$ instead of $\sigma((2y-1)z)$ for convenience, but the same conclusion applies

[5]We use the fact that $\frac{d}{dz}\sigma(z) = \sigma(z)(1 - \sigma(z))$

bility distribution over classes,

$$softmax(z_c) = \frac{\exp(z_c)}{\sum_j \exp(z_j)} \tag{2.29}$$

Given a total of C classes, the output is now a C-valued vector, and so rather than a derivative we work with the gradient of the NLL loss function, though the same principles apply. Before moving on, though, we will point out that for any class i, its NLL term is

$$-\log softmax(z_i) = -z_i + \log \sum_j \exp(z_j) \tag{2.30}$$

The partial derivative has a term linear in z_i, and thus cannot saturate - similar to the undoing of the exponent in the binary case-, thus ensuring that the gradient remains large enough for the model to learn.

2.3.2 Training in Deep Learning

Training DL models, including MLPs, is done using a gradient method, normally some variation of gradient descent. Given the chain structure of these models' layers, it can be computationally difficult to calculate the necessary gradients to update the weights. Additionally, many sub-expressions might repeat, leading to significant increases in memory and computing resource requirements. DL circumvents this by using a technique called Backpropagation (Backprop for short), which takes advantage of the derivative chain rule. The general idea is to represent the chain structure as a DAG, with nodes as functions/outputs, and directed edges indicating the flow of intermediate function outputs, from initial data input layer to final output layer. Subsequently, a DAG is constructed, with the same nodes but reversing the direction of the edges, moving backwards from output layer down to input layer, calculating the necessary gradients node by node. This ensures that no unique gradient is calculated more than once.

Recall the derivative chain rule: for any pair of functions f and g such that $z = f(y)$ and $y = g(x)$, the derivative

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{2.31}$$

In the more general case where $g : \mathbb{R}^m \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$, the partial derivative

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i} \tag{2.32}$$

We can interpret equation (2.32) as the equivalent of summing over all possible paths between x_i and z through the various y_j, with $\partial y_j / \partial x_i = 0$ if x_i is not an input when computing the j^{th} term $y_j = g(x)_j$. We can express the gradient of z w.r.t. x in vector notation as

$$\nabla_x z = J^T \nabla_y z = \begin{bmatrix} \nabla_x^T y_1 \\ \\ \\ \nabla_x^T y_n \end{bmatrix} \nabla_y z \tag{2.33}$$

where J is the Jacobian of $g(x)$.

Let us work out how to use backprop for an MLP. We would like to know the gradient of every weight in the model w.r.t. the loss function $\mathcal{L}(\hat{y}, y)$. For simplicity of notation, we follow the custom of defining $dx := d\mathcal{L}/dx$, with the understanding that most such derivatives are in reality partial derivatives. We will also assume that the output layer is one dimensional, but the following can easily be extended to the case of multi-dimensional outputs. In our first step, we note that trivially $d\mathcal{L} = 1$ and that the derivative of the loss function with respect to the output of the last layer is

$$dh^{(L)} = \frac{d\mathcal{L}}{dh^{(L)}} = \frac{d\mathcal{L}}{d\mathcal{L}} \frac{d\mathcal{L}}{dh^{(L)}} \tag{2.34}$$

which in the notation we will follow reduces to $dh^{(L)} = d\mathcal{L}\ dh^{(L)}$. Now, let's assume that for any layer k, we have already calculated the gradient of its output, $dh^{(k)}$, with dimensions $n_k \times 1$, where n_k is the number of hidden units in the layer. We then want to calculate the remaining derivatives in that layer, namely $dz^{(k)}$, $dW^{(k)}$, and $db^{(k)}$, as well as $dh^{(k-1)}$ to backpropagate the gradient down to the next layer.

Recall that the computation of each layer can be expressed as $h^{(k)} = f^{(k)}(z^{(k)})$, with $z^{(k)} = W^{(k)} h^{(k-1)} + b^{(k)}$, and $f^{(k)}$ applied element-wise. Trivially, then, $dz^{(k)} = dh^{(k)} \odot f'^{(k)}(z^{(k)})$, where $\odot$ is element-wise multiplication. Also trivially, $db^{(k)} = dz^{(k)}$. For $dW^{(k)}$, it can be easy to see that it should be some form of $dz^{(k)}$ times $h^{(k-1)}$. We first note that every element $W_{ij}^{(k)}$ is associated with i^{th} unit (and thus $z_i^{(k)}$) and the j^{th} input $h_j^{(k-1)}$, so that

$$\frac{dz_i^{(k)}}{dW_{ij}^{(k)}} = h_j^{(k-1)} \Rightarrow \frac{dz_i^{(k)}}{dW_i^{(k)}} = h^{(k-1)T} \tag{2.35}$$

Thus,

$$dW_i^{(k)} = dz_i^{(k)} \frac{dz_i^{(k)}}{dW_i} = dz_i^{(k)} h^{(k-1)T} \tag{2.36}$$

Putting this all together across the different rows of W, *i.e.*, units of layer, we obtain the formula:

$$dW^{(k)} = dz^{(k)} h^{(k-1)T} \tag{2.37}$$

What's left is the input to the layer. It can be easy to see that it should be some form of $dz^{(k)}$ times $W^{(k)}$. Note that each input $h_j^{(k-1)}$ is associated with weight $W_{ij}^{(k)}$ for each j^{th} unit (*i.e.*, row). Thus,

$$dh_j^{(k-1)} = dz^{(k)} \frac{dz^{(k)}}{dh_j^{(k-1)}} = \sum_i dz_i^{(k)} \frac{dz_i^{(k)}}{dh_j^{(k-1)}} = dz^{(k)} W_{:,j}^{(k)} \tag{2.38}$$

where $W_{:,j}^{(k)}$ is the j^{th} column of $W^{(k)}$. Thus, $dh^{(k-1)} = W^{(k)T}\ dz^{(k)}$.

With these formulas, the calculation of all gradients required by gradient descent to update the weights can be done efficiently, layer by layer. As can be seen in our derivations, the gradient of each node can be calculated individually, as long as all the gradients of all other necessary nodes (*i.e.*, parent nodes in the backward DAG, or equivalently children node in the forward pass DAG) have been calculated. Thus, in the case of a model with a more complex network topology, backprop can still be used, exploiting the derivative chain rule.

2.3.3 Convolutional Neural Networks (CNNs)

CNNs are a type of feed-forward neural network originally designed for image data that aimed to exploit the local relationship of inputs and features. They are intended for any kind of data with a grid-like structure, such as time series and images. Inspired by the notion of a convolution in digital signal processing, it introduces the notion of a filter, or sliding window with associated parameters that is convolved along the inputs to calculate a linear combination that is then passed through an activation function. Such a convolution layer is similar to a layer in MLP, with the difference that each unit uses only a small continuous portion of the input, and shares the same weights with other units in the layer. These additional requirements help reduce the total number of weights and allow the model to learn and exploit any invariant features in the data. In general, any NN containing at least one convolution layer is a CNN, and most CNNs have a sequence of convolution layers followed by one or more fully connected layers as in MLPs.

The convolution is an operator that receives two real-valued functions as inputs, $x(t)$ and $y(t)$, and produces as output another real-valued function $s(t)$ defined as the integral

$$s(t) = (x * y)(t) = \int_{-\infty}^{\infty} x(a)\ y(t-a)\ da \tag{2.39}$$

as long as it is defined. The first input function is often called the input, while the second one is called the kernel [14]. The convolution convolves or slides the kernel over the input and provides an output that we can interpret as a similarity measure between the two. Note that the kernel is expressed as $y(t-a)$, with a being the variable in integration: thus the kernel is shifted by t units and flipped. For any shift of t, $s(t)$ compares $x(t)$ and $y\ (t-a)$ across all values a in the domain; the more similar/different the two, the larger, positive/negative a value will be returned. In practice, the discretized form is used, given by the following formula:

$$s(t) = (x * y)(t) = \sum_{n=-\infty}^{\infty} x(n)\ y\ (t-n) \tag{2.40}$$

limiting n to values where $x()$ and $y()$ have non-zero values.

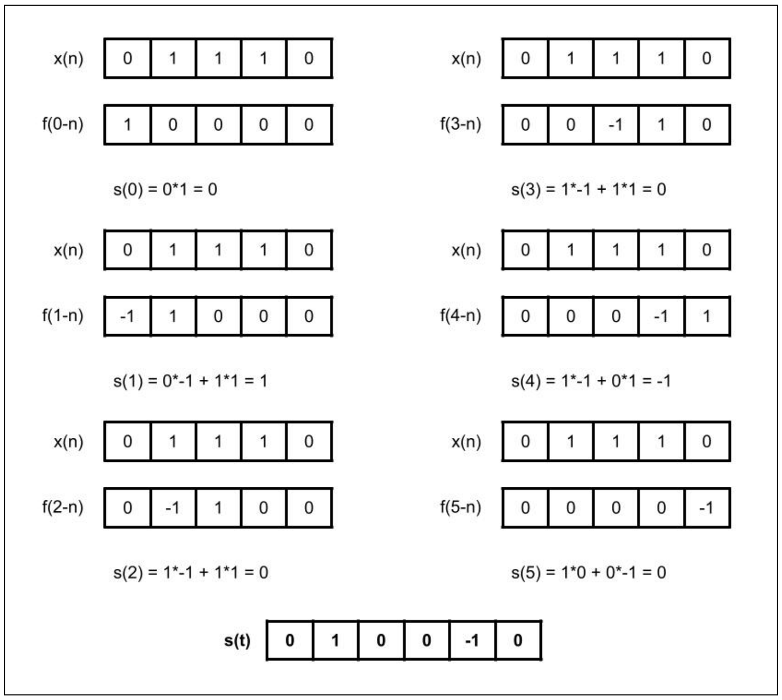


Figure 2.3: Example of convolution in DSP

In Fig 2.3, we see an example of a convolution for a one-dimensional signal, similar to audio input. The signal is defined as $x(t) = 1$ for $t \in \{1, 2, 3\}$ else 0, and the kernel, or filter, as $f(t) = [1, -1]$ for $t \in \{0, 1\}$ else 0. Thus we calculate

$$s(t) = (x * f)(t) = \sum_{n=0}^{4} x(n)\, f(t-n) \tag{2.41}$$

Notice that the convolution is commutative, and can thus be rewritten as

$$s(t) = (f * x)(t) = \sum_{n=0}^{4} f(n)\, x(t-n) \tag{2.42}$$

Since the kernel has a smaller range than the signal, this formulation results in less calculations performed and is thus preferred for implementation.

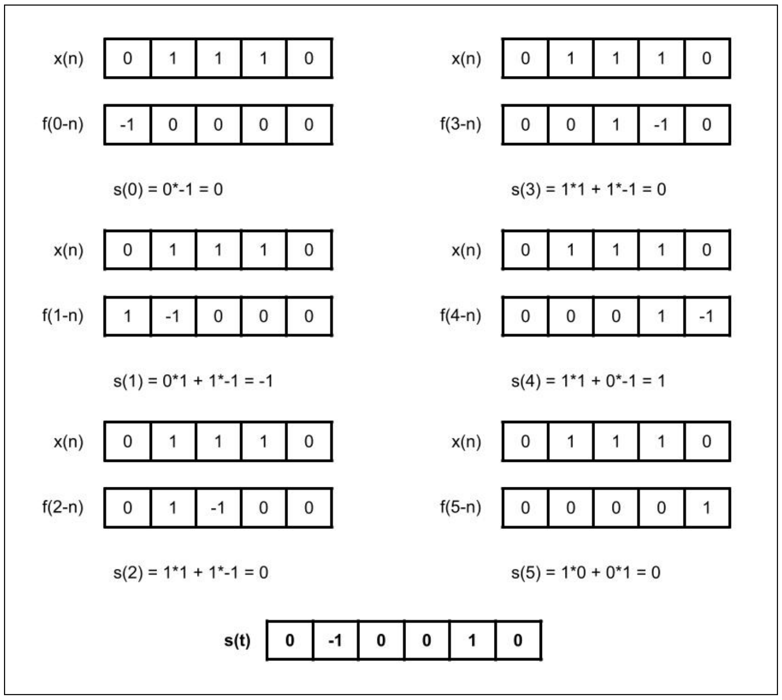


Figure 2.4: Example of cross-correlation in DSP

There is one key difference in the way that convolutions are used in DL versus other fields such as Digital Signal Processing (DSP): DL by in large does not flip the kernel before passing it across the signal, thus resulting in what in DSP is called cross-correlation:

$$s(t) = (x * y)(t) = \sum_{n=-\infty}^{\infty} x(t+n)\ y(n) \tag{2.43}$$

Due to its commutative property, the convolution of DSP is “useful for writing proofs” but “is not usually an important property of a neural network implementation” [14]. We can see in Fig 2.4 how the cross-correlation, or “convolution” in DL, produces comparable results. In what remains, we will use convolution to refer to its use in DL and not DSP, *i.e.*, equation (2.43).

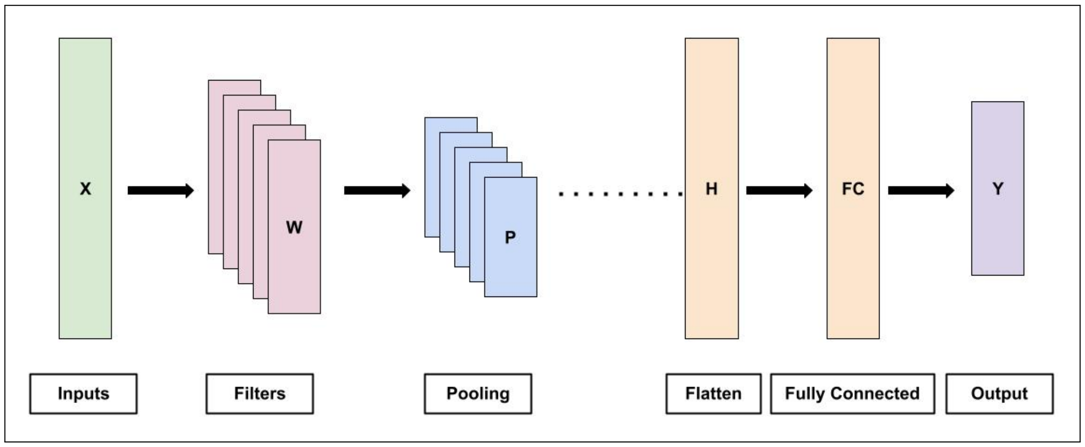


Figure 2.5: General structure of a CNN

The general structure of a CNN model consists of an input layer; one or more convolution layers with optional pooling layers; a flatten layer; then, one or more optional fully connected layers (as those used in MLPs); and the output layer 2.5. As we shall see below, convolution layers generate tensor-shaped outputs, and the output of the last convolution layer is flattened into a one-dimensional shape before applying the output layer. Note, this may vary depending on the task at hand, but for simple regression and classification problems like the ones we have been discussing, this would be the case.

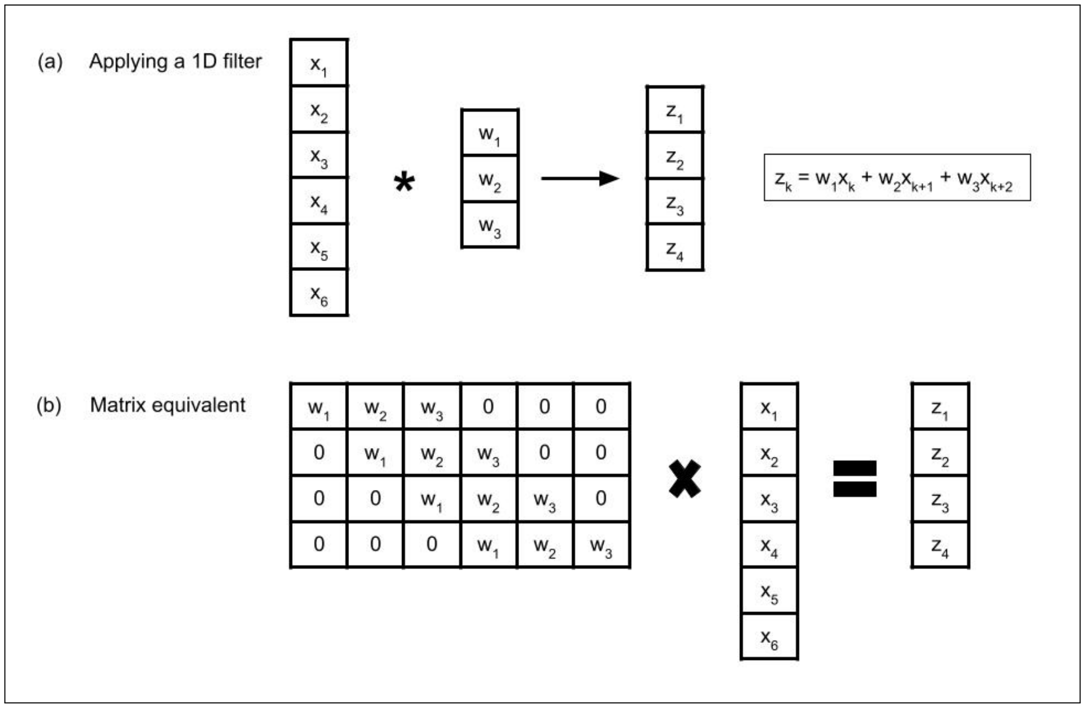


Figure 2.6: Example of a 1D convolution - filter (a) versus matrix (b) representation

To develop an understanding of how convolutions work in CNNs, let's begin by examining a simple example of a convolution layer receiving an input of dimensions 6×1 and one kernel of size $f_n = 3$ (Fig 2.6a). For the first output unit, we align the filter with the beginning of the input, and proceed as with MLP with one key difference: we only use the first f_n input elements, *i.e.*, the first to third ones. We then slide the kernel by one and repeat as before, using the second to fourth input elements, and using the same weight values. We do this until we can no longer slide the kernel without exceeding the input limit - thus our output will be of size $n_{input} - f_n + 1$. Unlike the units in an MLP layer - or fully connected (FC) layer-, we say that the units in a convolution layer only have access to the input values within their receptive field, which is determined by the size or dimension of the kernel. And the output of such a layer is determined by both the kernel size and input size. Despite these differences, though, we can also express the computation of a convolution layer with matrix multiplication. In the one-dimensional case, we do so by defining a matrix in $\mathbb{R}^{n_{input} \times n_{output}}$, where each row contains the weights of the kernel padded by zeros and is equivalent to the row above shifted to the right (Fig 2.6b). Such a matrix, with the same values along the diagonal and off-diagonals, is known as a Toeplitz matrix.

Of course, each convolutional layer can apply more than one kernel, thus producing an output with dimensions $(n_{input} - f_n + 1) \times n_c$, where n_c is the number of kernels (also known as the number of channels). All units in the same channel use the same weights, and each channel has its own set of weights or kernel. Similarly, the inputs can also have more than one channel, even the initial input, *e.g.*, stereophonic recordings. In such cases, each kernel convolves as before, including the inputs of all channels within the receptive field. That is, if the layer contains n_c one-dimensional kernels of size f_n, and receives an input h of $n' \times c'$ dimensions where c' is the number of channels, then its output is $f(z)$ where

$$z_{i,j} = \sum_{q=0}^{f_n-1} \sum_{r=0}^{c'-1} w_{q,r,j} x_{i+q,r} \tag{2.44}$$

where $z_{i,j}$ is the i^{th} output for the j^{th} kernel.

Lastly, we can alter the convolution by adding zero-padding and a stride greater than one. Zero-padding (or simply padding) by some integer p involves adding p zeros at both ends of the data. Since the edge values of the input are used only once, it can be beneficial to use padding to better capture any information in those input positions. The stride s determines how many units the kernel is shifted during convolution. Above, we assumed a stride of one, but it can be any positive integer. Taken all together, the output of a convolution layer has the following dimensions:

$$\left(\left\lfloor \frac{n' + 2p - f_n}{s} \right\rfloor + 1\right) \times n_c \tag{2.45}$$

where n', f_n, n_c are defined as above.

Besides the activation function, convolution layers can additionally apply what is called the pooling function, also called a pooling layer (or simply pooling). It is an optional layer that involves sliding a window of size f_n in the same manner as a kernel, except that it involves no weights and is applied independently, channel by channel. Its goal is to apply some function (usually average or maximum) to the values inside the window, thus replacing the activations from the convolution by a summary statistic. The benefit of pooling is to make the internal representation "approximately invariant to small translations of the input" [14]. It also helps reduce computation and memory requirements.

Although CNNs have a more complex structure than MLPs, training proceeds as before with a gradient descent method using the Backprop algorithm: we define a graph structure and exploit the derivative chain rule. See Fig 2.7 for an example of how the gradients for weights W would be calculated for an arbitrary layer.

In theory we can use CNNs as well as MLPs for any kind of dataset, but CNNs are preferable when the data is believed to contain local interactions that determine the desired output. In such situations CNNs can improve performance by leveraging the following properties: 1) sparse interactions; 2) parameter sharing; and 3) equivariant

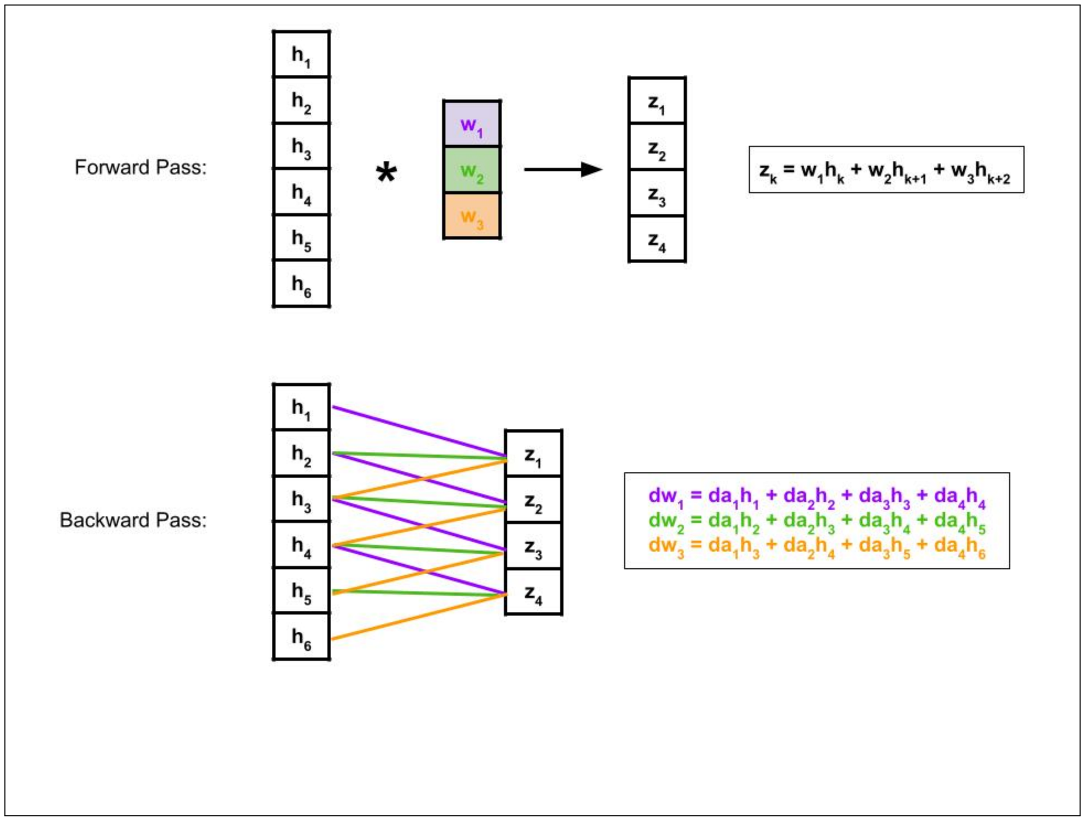

Forward Pass:
h_1 h_2 h_3 h_4 h_5 h_6
*
w_1 w_2 w_3
z_1 z_2 z_3 z_4
$z_k = w_1h_k + w_2h_{k+1} + w_3h_{k+2}$
Backward Pass:
$dw_1 = da_1h_1 + da_2h_2 + da_3h_3 + da_4h_4$
$dw_2 = da_1h_2 + da_2h_3 + da_3h_4 + da_4h_5$
$dw_3 = da_1h_3 + da_2h_4 + da_3h_5 + da_4h_6$

Figure 2.7: Example of gradient calculation for weights for arbitrary layer

representation.

By sparse interactions, we mean that, in comparison to the MLP, the inputs to any layer are sparsely connected to its outputs. Through the kernel, each output interacts with a limited number of nearby input values. This dramatically reduces the memory and computation requirements. Furthermore, deeper models do allow for farther away input values to indirectly interact through the output units of deeper layers, so that CNNs are also good for complex interactions that are assumed to be the result of simpler, more local ones.

By parameter sharing, we mean that, thanks to the kernel, for any given layer, the outputs in the same output channel share the same weights, thus greatly reducing the model size. For each channel, we only need to learn one set of weights, rather than one distinct set per output unit. As with the above case, this too results in decreased computation time.

By equivariant representation, we mean that the internal representation that the model learns is equivariant to translation. That is to say, if our function/model learns to identify a feature, it will be able to do so even if we change the position in the input sample where that feature appears. Formally, we say a function is equivariant if its output changes in the same way as its input changes. Thus, by moving the appearance of a feature in the input sample, the model's representation would simply change when/where the identification of that feature appears in the learnt, internal representation.

2.4 Sound and Spectrograms - an Overview

2.4.1 The Physics of Sound

In physics, sounds or sound waves are described as mechanical waves produced by a vibrating object. Along with a broader set of physical phenomena defined by oscillations, they are best and commonly represented through the trigonometric functions of sine and cosine and thus also described as sinusoidal waves. Sounds can be simple or complex, and we define a simple sound or sinusoidal as follows:

$$A \sin(2\pi f t + \phi) \tag{2.46}$$

where A is the amplitude, f the frequency, ϕ the phase.[6] The following definitions are key concepts defining any simple sinusoidal wave.

Amplitude is defined as the "greatest distance from the equilibrium point" [12]. That is, given a sinusoidal centered along the x-axis, it defines its height relative to 0 (*i.e.*, equilibrium point). Since sine has a maximum absolute value of 1, multiplying by A stretches the function along the y-axis. It affects the energy of the wave, since "the total mechanical energy of a simple harmonic oscillator is proportional to the square of the amplitude" [12].

Frequency is the number of complete cycles per second, and is recorded in units known as Hertz (Hz).

Phase, also known as the phase angle, is the shift of the sinusoidal along the x-axis, and defines its value when $t = 0$.

Period is the time required to complete one cycle (in seconds). Equivalently, we define it as the distance between two peaks or troughs (or "crests" and "rarefactions" in physics) of the sinusoidal. Logically, it is the inverse of frequency: $T = 1/f$.

Note that the standard $\sin(t)$ completes one full cycle in 2π seconds. Thus, we include a factor of 2π in the argument, ensuring that when $f = 1$, one cycle is indeed completed in 1 second. You may alternatively see the equation written as $A\sin(\omega t+\phi)$ where $\omega = 2\pi f$ and is called the angular frequency.

Thanks to Fourier's Theorem, "it can be shown that any complex wave can be considered as being composed of many simple sinusoidal waves of different amplitudes, wavelengths, and frequencies" (see Fig 2.8) [12]. It is difficult to overstate the

[6]In some textbooks, the cosine function is used instead. Recall that $\sin(t) = \cos(\frac{\pi}{2} - t)$

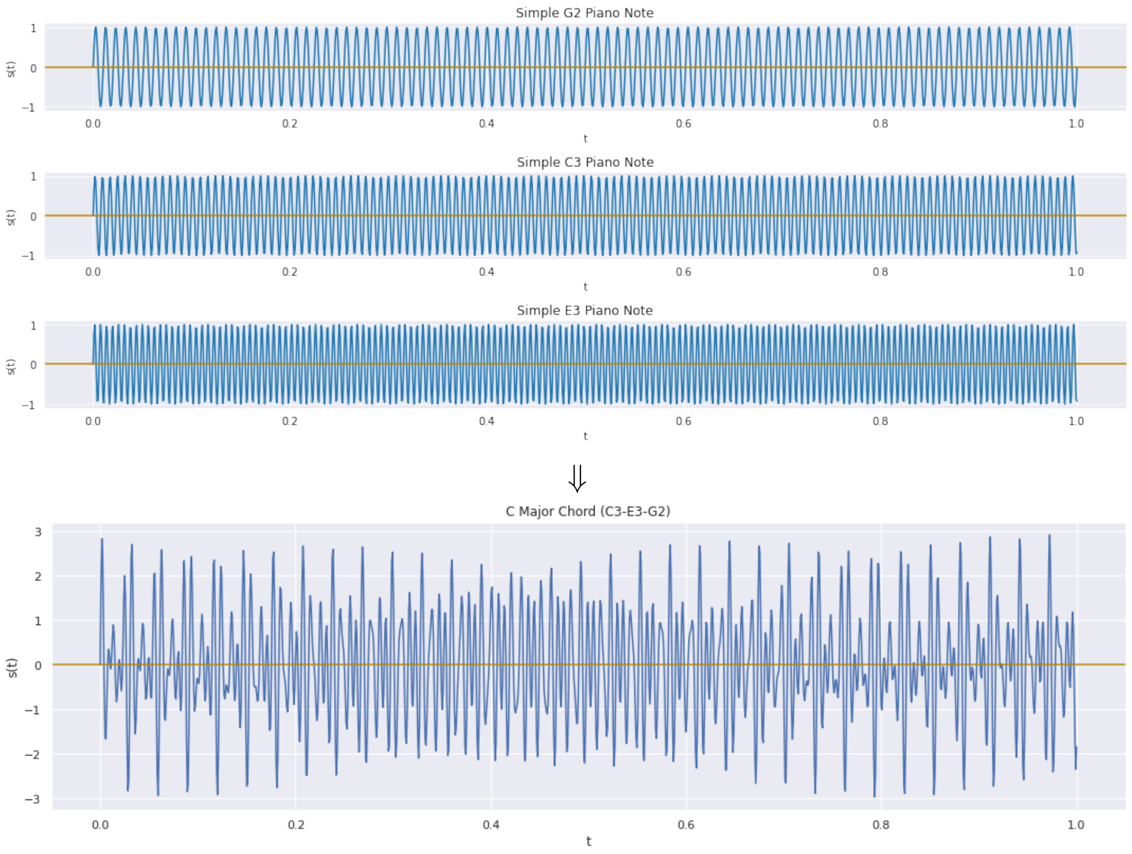


Figure 2.8: A simulated C-major chord: the three simple waves corresponding to the three notes G2, C3, E3 (above); and their sum corresponding to a C-major chord (below)

importance of this mathematical discovery, for it allows us to describe a complex wave with utmost efficiency and minimal storage. While we would have to keep in record a sequence of thousands of values to describe a recorded complex wave, Fourier's Theorem tells us that as long as we know the simple waves that make up the complex wave, we can equivalently describe it by the parametrization of its components. In the case of a simple wave, that means storing only three parameters: the frequency, phase, and amplitude. And thus for any complex wave containing N simple waves, we would need only $3N$ (*i.e.*, $O(N)$) parameters total - a remarkable compression of information.

2.4.2 The Basic Fourier Transform

The basic Fourier Transform provides a method for extracting the parameters of simple sinusoidal waves present in a complex wave that is assumed stable and constant in the recorded data (see Fig 2.9). We will first present Fourier series as a starting point, and provide an intuition for the basic Fourier Transform (FT). We will then briefly describe the two flavors of FT: continuous and discrete (DFT). The latter is used in practice, given the discrete nature of data.

Given a periodic signal $s(t)$, we can describe it as an infinite sum of cosines and sines, known as a Fourier Series:

$$s(t) = a_0 + \sum_{k=1}^{\infty} a_k \cos(kt) + b_k \sin(kt) \tag{2.47}$$

Sound waves can thus be represented as a Fourier series, with all $b_k = 0$ (we omit the

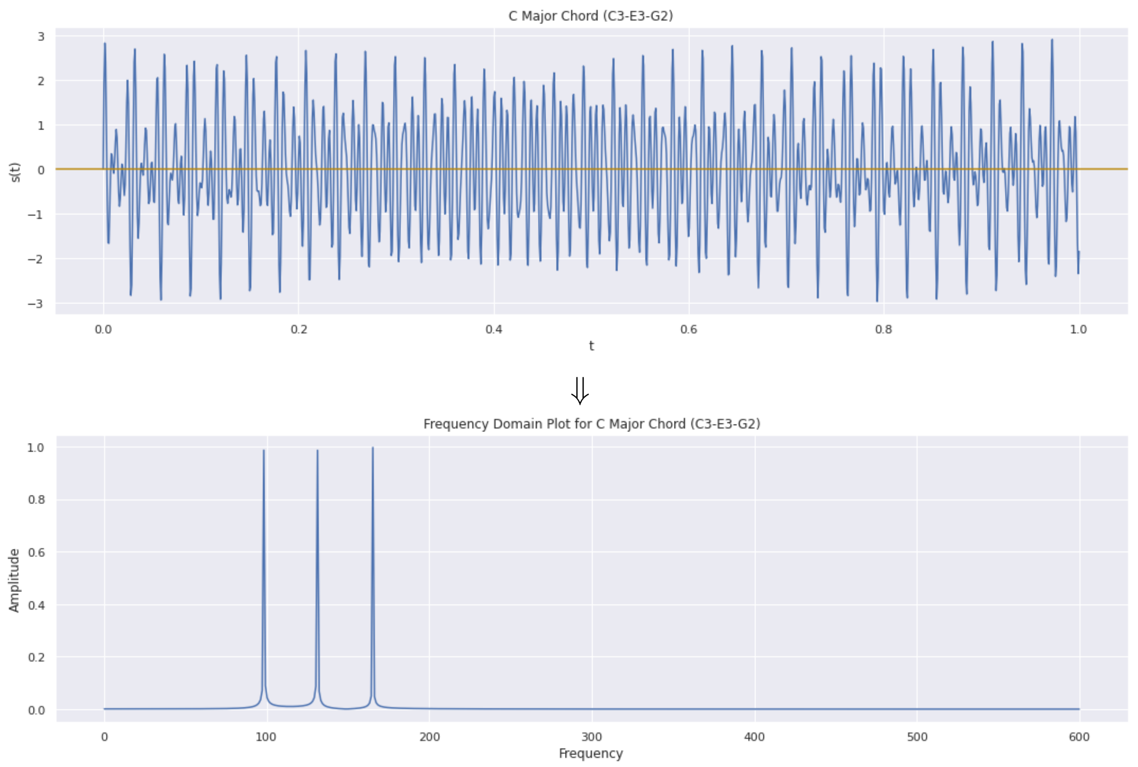


Figure 2.9: A simulated C-major chord comprised of G2, C3, E3 (above), and its corresponding spectrum (below)

phase in our discussion for simplicity, but the same conclusions hold). We can solve for the coefficients of a Fourier series by noting that in the interval of $[0, 2\pi]$ and for $k, m, n \in \mathbb{Z}_{>0}$, the following properties hold:

Property 1. Each sinusoidal integrates to 0:

$$\int_0^{2\pi} \cos(kt)\, dt = \int_0^{2\pi} sin(kt)\, dt = 0$$

This is because said interval contains an integer multiple of the period of each sinusoidal.

Property 2. The product of any $\cos(mx)$ and $\sin(nx)$ also integrates to 0:

$$\int_0^{2\pi} \cos(mt)\sin(nt)\, dt = \int_0^{2\pi} \frac{1}{2}[\sin((n+m)t) + \sin((n-m)t)]\, dt = 0$$

Property 3. The product of two sines or two cosines also integrates to 0 whenever they are distinct:

$$\int_0^{2\pi} \cos(mt)\cos(nt)\, dt = \int_0^{2\pi} \frac{1}{2}[\cos((m-n)t) + \cos((m+n)t)]\, dt = 0 \quad \forall m \neq n$$

$$\int_0^{2\pi} \sin(mt)\sin(nt)\, dt = \int_0^{2\pi} \frac{1}{2}[\cos(m-n)t) - \cos((m+n)t)]\, dt = 0 \quad \forall m \neq n$$

Property 4. The product of two sines or two cosines integrates to π when they are the same:

$$\int_0^{2\pi} \sin^2(mt)\ dt = \int_0^{2\pi} \frac{1}{2}[\cos(0) - \cos(2mt)]\ dt = \pi$$

$$\int_0^{2\pi} \cos^2(mt)\ dt = \int_0^{2\pi} \frac{1}{2}[\cos(0) + \cos(2mt)]\ dt = \pi$$

Thus, using **Property 1**, we can integrate both sides of a Fourier series to obtain the first coefficient:

$$a_0 = \frac{1}{2\pi}\int_0^{2\pi} s(t)\ dt \tag{2.48}$$

Then, using **Properties 2-4**, we can multiply by any $\cos(nt)$ and integrate to obtain each a_n:

$$a_n = \frac{1}{\pi}\int_0^{2\pi} s(t)\cos(nt)\ dt \tag{2.49}$$

Similarly, we can multiply by any $\sin(nt)$ to obtain each b_n:

$$b_n = \frac{1}{\pi}\int_0^{2\pi} s(t)\sin(nt)\ dt \tag{2.50}$$

We can now see that given any sound wave $s(t) = \sum a_k cos(kt)$, we can obtain for each angular frequency $\omega = k$ its amplitude a_k. Note that if the frequency is not present in $s(t)$, then:

$$\int_0^{2\pi} s(t)\cos(kt)\ dt = \int_0^{2\pi} a_0\cos(kt)\ dt = 0 \tag{2.51}$$

This is key to developing the FT: for any given interval of time that is an integer multiple of the periods of simple waves involved in the signal, we can perform a cross-correlation on the signal using $\cos(kt)$ as our analyzing functions. And the resulting value gives us a comparative value of the amplitude for the sinusoidal with frequency k. It is comparative because the longer the interval we integrate over, the larger the value will be; thus, it is accurate compared to the amplitudes of the other waves present.

To simplify the mathematics involved, waves are generally represented by complex numbers in polar coordinates. In such a representation, the radius is the amplitude, and the angle is the (angular) frequency. By using Euler's Formula, we can rewrite the equation for a complex wave signal as:

$$s(t) = \sum_k A_k e^{ikt+\phi_k} = \sum_k A_k[\cos(kt+\phi_k) + i\sin(kt+\phi_k)] \tag{2.52}$$

And our analyzing functions $f_k(t)$ become:

$$f_k(t) = e^{ikt} \quad \forall\, k \in \mathbb{R} \tag{2.53}$$

We can now formally state the Fourier Transform as follows:

$$\hat{A}_k = \int_{-\infty}^{\infty} s(t)\, e^{ikt} \quad \forall\, k \in \mathbb{R} \tag{2.54}$$

Variations of the formulation may include a factor of 2π inside the exponent (recall the difference between frequency and angular frequency), or a negative in the exponent - which simply inverts the direction of rotation in the polar plane. Additionally, one can obtain the phase shift by adding e^{ϕ} factor and optimizing.

Before moving on, it is important to note here that "the Fourier transform of a signal is a change of basis in an appropriate Hilbert space" which allows us to map from the time-domain to the frequency domain and back through the invertible "Fourier operators" that we have defined above [33]. These Fourier operators are independent under integration as we noted earlier. However, they are independent under integration within an interval that contains integer multiples of their period, a fact that will complicate the actual extraction of frequencies from real world data. This is due to the finite (and discrete) nature of recorded signals, as we shall now see.

The FT as we have described so far lives in continuous space, both in terms of time and frequency. Data, however, must be discrete in both time and frequency, and a Discrete Fourier Transform (DFT) must be used instead of the continuous FT. To do so, we first discretize the time-axis by choosing a sample rate. This sample rate determines how many evenly distanced values are sampled from the signal (each such value is often called a "sample" and we will adopt this convention for the remainder of the discussion). Then we quantize the samples of the signal, that is, we discretize the value-axis into a set of possible values akin to bins, and we round each sample to the nearest such bin. Thus, we transform a signal in continuous space, $s(t)$ into a discrete finite sequence of quantized values, $s[n]$ (we adopt here the conventions of brackets and n for discrete signals as used in [33]).

We can attempt a preliminary definition of the DFT as follows:

$$\hat{A}_k = \sum_{n=0}^{N-1} s[n]\, e^{\frac{ikn}{N}} \quad \forall k \tag{2.55}$$

where k is a given frequency and N is the total length of the discrete signal. In the continuous case, the range of k was $\mathbb{R}$; we now have to determine its range in the discrete case. In order to maintain the invertible nature of the transform, we must choose a finite range of values for k equal to the length of the sequence, hence N. Such values must result in a basis $\{[1, e^{ik/N}, e^{ik2/N}, \ldots, e^{ik(N-1)/N}]\}_k$, and thus the length N must equal a multiple integer of the period for each k, that is:

$$e^{\frac{ik0}{N}} = e^{\frac{ikN}{N}} = 1 \Rightarrow e^{ik} = 1 \tag{2.56}$$

which only happens when k is a multiple of 2π. Thus, we shall rewrite each frequency and its corresponding Fourier operator as [33]:

$$\omega_k = \frac{2\pi k}{N} \quad \forall\, k \in [0, 1, 2, \ldots, N-1] \tag{2.57}$$

$$f_k[n] = e^{i\omega_k n} \quad \forall\, n \in [0, 1, \ldots, N-1]$$

These discrete Fourier operators are indeed independent, since the dot product of the resulting vectors/signals $\mathbf{w}_k, \mathbf{w}_j$ for any two frequencies k, j is:

$$< \mathbf{w}_k, \mathbf{w}_j > = \sum_{n=0}^{N-1} \exp(i\omega_k n) \exp(-i\omega_j n) \tag{2.58}$$

$$= \sum_{n=0}^{N-1} \exp\left(\frac{i2\pi}{N}\right)^{(k-j)n} \tag{2.59}$$

$$= \begin{cases} N & \text{if } k = j \\ \frac{1-\exp(\frac{i2\pi}{N})^{(k-j)N}}{1-\exp(\frac{i2\pi}{N})} = 0 & \text{otherwise} \end{cases} \tag{2.60}$$

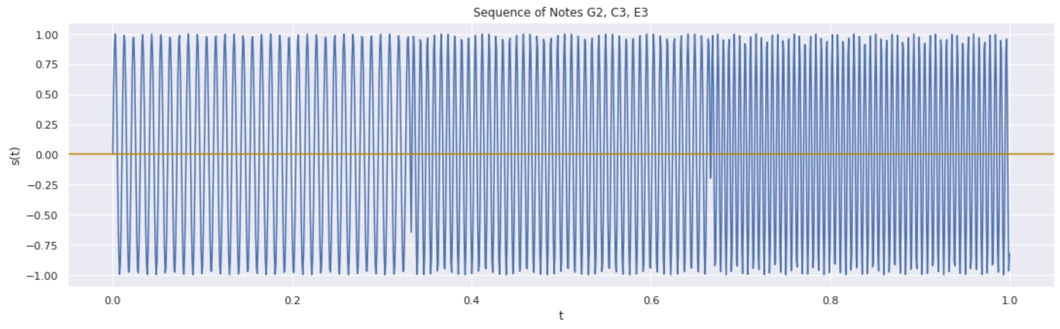
Thus they constitute a basis for the space of discrete signals of length N, as desired. We can now formally write the DFT as:

$$\hat{A}_k = \sum_{n=0}^{N-1} s[n]\, e^{i\omega_k n} \quad \forall\, k \in [0, 1, \ldots, N-1] \tag{2.61}$$

2.4.3 The Fourier Transform for Spectrograms

The finite nature of the signal and the process of discretization and quantization mean that there is no guarantee that we will be able to precisely determine the frequencies present in the original, continuous signal. The time duration of the recording and the granularity of our samples (*i.e.*, the sampling rate) impose limitations on how accurate our results can be. For example, frequencies with periods longer than our signal will not be extractable. Furthermore, for any real world recording, various sound signals may start and stop, making them not only more complicated to extract, but also requiring us to modify the standard DFT as expressed above.

The solution to this problem lies in taking a window of size $M < N$, and calculating the DFT as we slide this window over the samples. Each application of the DFT produces a set of (amplitude, frequency) pairs for a subset of consecutive samples, called a spectrum. And the sequence of such sets of pairs taken together produces



⇓

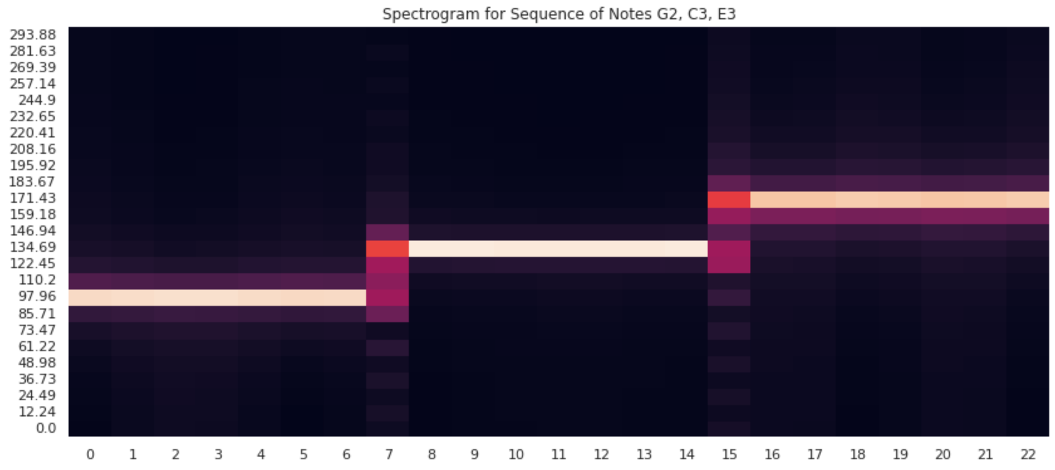


Figure 2.10: Simulated sequence of notes G2, C3, E3 (above), and the corresponding spectrogram (below)

a time-frequency representation of the signal, which allows us to observe the variations of frequencies across time. Formally, we define this procedure, known as the Short-Time Fourier Transfer (STFT), as:

$$S[k,m] = \sum_{n=0}^{M-1} s[n+mh] \cdot e^{i\omega_k n} \tag{2.62}$$

where m is the frame number (*i.e.*, time in terms of window application), and h is known as the hop length and determines how quickly we slide the window over the samples (analogous to stride in CNNs). The result is an $M \times \left(\lfloor \frac{N-M}{h} \rfloor + 1\right)$ matrix of values which, when visualized as in Fig 2.10, is known as a spectrogram. The x-axis of a spectrogram represents time in terms of frame number, and the y-axis represents frequency. The difference of color or intensity represents the magnitude of the values.

2.4.4 Limitations of the DFT and Spectrograms

We have already mentioned that, as a consequence of the discrete nature of data and the number of samples N available for a signal, the number of frequencies that we can check for is limited - specifically, to N. However, the actual unique frequencies we can identify is half this number and is often referred to as the Nyquist frequency or Nyquist limit.

First, let us recall that cosine is an even function, that is, $\cos(-\omega) = \cos(\omega)$, and thus only consider angular frequencies $\omega > 0$. Then, let us further limit the range of

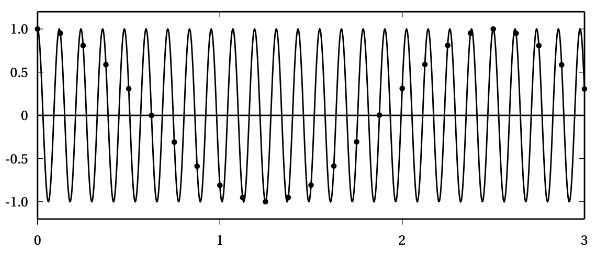


Figure 2.11: Aliasing example for $s(t) = \cos(2\pi \cdot 8400t)$ (line) sampled at frequency of $F = 8000$ Hz (dots). The samples are indistinguishable from $r(t) = \cos(2\pi \cdot 400t)$ [33]

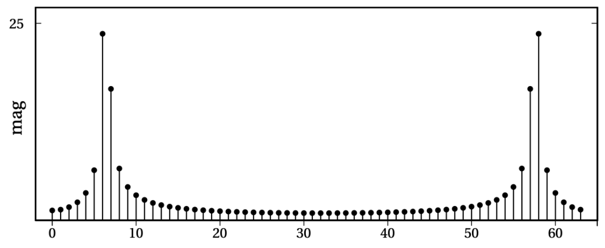


Figure 2.12: Spectrum example for $s[n] = \cos(\pi n/5)$ for $n = 0, 1, \ldots, 63$ [33]

ω to $(0, 2\pi)$, since for any $\omega > 2\pi$, we can rewrite $\omega = 2\pi + \theta$ for some θ and thus:

$$cos(\omega) = \cos(2\pi + \theta) = \cos(k2\pi + \theta) = \cos(\theta)$$

for any integer multiple k of 2π, a phenomenon known as aliasing (see Fig 2.11). Now, notice that for any $\omega > \pi$:

$$\cos(\omega) = \cos(\theta) = \cos(-\theta) \quad \text{for} \quad \theta = \omega - 2\pi < 0$$

That is, there is an angular frequency θ of opposite rotation that is equivalent with $|\theta| < \pi$. And because cosine is an even function, then there exists an angular frequency, namely $0 < -\theta < \pi$, that is equivalent to ω. This halves our range of possible frequencies, and causes spectra of signals to be symmetric around $N/2$ (see Fig 2.12).

In the case of spectrograms, there is the additional challenge of determining the window size and hop length. There are no hard rules nor mathematical formulas for making these choices. In any case, the practitioner must pay careful attention to the trade-off between frequency resolution and time resolution. On the one hand, we want high frequency resolution. In order to do this, we need to increase the window size, reaching a maximum at $M = N$, which is just the application of the standard DFT once over the whole signal. However, in doing so, we lose information about when in time the frequencies occur. If we want to know with high accuracy when the frequencies occur, we must choose a smaller window size, resulting in very course spectral information.

Chapter 3

Methodology

In this chapter, we provide the information concerning our experiments. We start by describing the datasets we used, and then lay out the settings and conditions under which we carried out our experiments.

3.1 Datasets

The first dataset we used is AudioMNIST, an open dataset available on Github, containing 30000 voiced recordings of the digits from 0 to 9 by 60 different speakers, with a sampling rate of 48kHz [5]. Each speaker produced a total of 500 recordings, 50 per digit. For our experiments, we subsampled 50 instances per each digit for a total dataset size of 500 recordings, from a total of 50 unique speakers - more specifically, we randomly selected 50 speakers out of 60, and from each speaker, randomly selected one recording per digit. We will subsequently refer to this subsample as N1, because it is the equivalent of sampling all the recordings of one speaker; for comparison, we also ran experiments with the equivalent of half (N1/2), two (N2), four (N4), and eight (N8) speakers, each composed similarly, but increasing the number of digit recordings sampled from each speaker accordingly. For evaluation, we randomly selected an additional five speakers from the remaining ten, and selected all their recordings for the test set, for a total of 2500 recordings. Additionally, we also used N3 and N5 to compare how Time and Pitch augmentation compare with subsamples of equal training set size, respectively.

The second dataset is the Xeno-canto data provided by the BirdCLEF 2021 competition on Kaggle hosted by the Cornell Lab of Ornithology [21]. This dataset consists of crowdsourced bird call recordings in the wild from hundreds of different species, with a sampling rate of 32kHz. The instances are not expert labeled, and vary significantly in length, from some lasting a few seconds to others well over a minute. For our subsample, we randomly selected ten classes with at least 80 recordings lasting no longer than fifteen seconds, and then randomly sampled 80 recordings per class for a dataset size of 800. We will subsequently refer to this subsample as XC.

The third and final dataset consists of animal sounds recorded in the wild by sensors in Puerto Rico at a sampling rate of 48kHz [1]. This dataset was constructed by clipping longer recordings into five second segments that were manually labeled. The set of labels/classes span various different species as well as different types of noise. For our subsample, we restricted ourselves to species containing 50 or more instances, resulting in a dataset of nine species each with randomly sampled 74 instances for a total dataset size of 666. We will subsequently refer to this subsample as PR.

3.2 Model Choices and Experimental Setup

As previously stated, most research on animal audio classification has used spectrograms as inputs. These are in essence extracted features using the short-time Fourier transform (STFT) or similar method, projecting raw audio onto ‘frequency maps’ along a discretized time-axis. Such a representation leverages human generated knowledge from the field of digital signal processing and facilitates analytical and predictive tasks. This in turn has led most such research to view the resulting spectrograms as a kind of image, and employ on them the same techniques used for computer vision, namely convolutional neural networks (CNNs) and various data augmentation (DA) techniques.

Our focus though, has been on raw inputs, desiring to depend as little as possible on human generated knowledge and testing how much neural networks can learn about the underlying structure of the data to perform well given un-preprocessed audio inputs. We additionally focus on evaluating the impact that DA techniques can have in cases such as ours when data is relatively scarce. Thus, the main component of our research consists on experiments with raw inputs, with and without DA, across a variety of settings as described below. We then compare these results with those for models trained on spectrograms.

We chose to explore the performance of CNNs so that our results and methods can be compared with other studies using spectrograms. Furthermore, given the temporal nature of audio signals, we expect CNNs to perform well, capturing the local dependencies between signal samples that lead to frequency detection and the extraction of traditional features for the type of problem studied.

First, we used AudioMNIST N1 to perform various preliminary explorations to help us get a feel for the problem at hand, and help narrow down various hyper-parameter choices. In particular, these initial experiments suggested to us that with ever increasing depth, increasing the number of filters by powers of 2 results in a very large number of model parameters, eventually making training difficult and putting a strain on computational resources.

It is common practice in CV to set the number of filters in each layer to some, generally large, power of two, and to either increase the number with depth, or repeat set sequences of filter numbers. This general tendency is replicated in much DL for audio, since most practitioners use spectrograms. There are thus no guidelines regarding the preferred number of filters nor their sequencing through the layers for the case of raw inputs. We thus ran a series of experiments to determine the impact of the number of filters and their sequencing, and report these results in the next chapter. Specifically, we tested performance on the basic CNN model with seven layers for: 1) randomly generated filter numbers per each layer, sampled from a Uniform$(2, 256)$; 2) constant power of two versus increasing powers of two; and 3) increasing powers of two versus increments of ten (referred to as pow2 and step10 respectively).

For our main experiments, we tested different values for the following hyper-parameters: 1) depth of convolutions, 2) filter number sequence, and 3) maxpool filter size.

We trained models of at least depth two, with each convolutional layer always followed by a maxpool layer, and no padding. When we could no longer add extra layers due to dimension reduction (≥ 8 for AudioMNIST, ≥ 10 for XC, ≥ 9 for PR), we resorted to removing maxpool layers in the first layers as needed, while adding more convolutional + maxpool layers at the end. In preliminary results, most models rarely improved much once it became necessary to remove maxpool layers; thus we opted to train models with up to three such removals, with a few exceptions where results showed the presence of considerable improvement or for comparison with such cases of other model settings. For further model information, see Table 3.1.

Filter size	3
Stride	1
Filter numbers	$[10, 20, 30, 40, \ldots]$
Dense layer size	64

Table 3.1: Model Specifications

We trained models using different powers of two for maxpool filter sizes. For any given model, all maxpool layers had the exact same filter size, with equally valued stride. For comparison across other attributes such as sample size and DA, we used a maxpool fiter size of four, which was our initial default setting.

We then compare these results with spectrograms using two-dimensional convolutions with maxpool filter sizes of 2×2 and 4×4 respectively, keeping all other hyper-parameter settings identical. Additionally, we also tested filter size and fully connected layer size, as well as different schemes for filter number sequences, but initial results showed no significant difference or improvement in result and thus we did not include these in the final experimental set-up.

3.2.1 Data Augmentation

We performed the following data augmentation (DA) methods, independently, to test whether they improve model performance. Of particular interest is whether increasing data size through DA produces comparable improvements in performance vis-a-vis collecting more real data.

Pitch Shift: we implemented this technique by shifting the frequencies of each datapoint by ± 1 and ± 2 semitones, using the librosa function effects.pitch_shift() [25]. This resulted in a final training set size five times larger than the original.

Time Stretch: we implemented this technique by speeding up/slowing down each datapoint by factors of $2/3$ and $3/2$ respectively, using the librosa function effects.time_stretch() [25]. This resulted in a final training set size three times larger than the original.

Random Perturbation: we implemented this technique by adding random values sampled from a Uniform$(0, 1)$ to each datapoint, using the numpy library random.rand(). This resulted in a final training set size twice the original.

Gaussian Noise: we implemented this technique by adding random values sampled from a Normal$(0, 1)$ or Normal$(0, 2)$ respectively, using the numpy function random.normal(). This resulted in a final training set size three times larger than the original.

3.2.2 Spectrogram

We trained models on melspectrograms generated by using the librosa function melspec() [25]. We used a window size of 2048, hop length of 256, and nmels to be equal to 256, which produced spectrograms for each sample similar in input size as the one-dimensional raw inputs. This allows us to better compare training time across models.

3.2.3 Training Conditions

We used the Keras and Tensorflow libraries in Python for our model implementations, with Adam optimizer and a batch size of 32 (early experiments on AudioMNIST with larger batch sizes showed no significant improvements). Using preliminary results from trials on AudioMNIST, we decided to train each model for 50 epochs, for each dataset. We then visualized the performance of training validation loss and accuracy to ensure that all models achieved a stable (non improving) performance. Furthermore, we ran some of the models for up to 100 epochs to ensure that further training did not improve performance.

To gauge whether the performance of our trained networks was due purely to chance or not, we trained each model for each dataset ten distinct times and used the average performance of each model per dataset for the final analysis. This allows us to have an idea of the expected performance of a given model while not requiring an excessive amount of compute time, which we hope will be useful to deep learning practitioners who, in real world situations, will unlikely have the time and resources to train their models multiple times to check the reliability of performance etc.

3.2.4 Computational Resources

Most model training was performed thanks to resources made available to us by the NSF through their XSEDE Startup Allocation [39]. Our experiments were part of the following Startup Allocation Grant:

- Project: CIS210060
- Title: Impact of Data Augmentation and Model Complexity for Species Classification Given Unprocessed Audio Recordings using Convolutional Neural Networks
- System: SDSC Dell Cluster with NVIDIA V100 GPUs NVLINK and HDR IB (Expanse GPU)

Additionally, preliminary exploration as well as additional training was performed on computing resources available through Kaggle and Google Colaboratory.

Chapter 4

Results and Analysis

Here we present the results and analyses of our experiments. We first begin with a few key results from our hyper-parameter testing experiments, and then proceed with results for AudioMNIST, XC, and PR using raw inputs. We finish by comparing these results with those using spectrograms, and assessing model preferences given time and computing resource constraints.

4.1 Hyper-Parameter Experiment Results

We first ran a set of experiments to test hyper-parameter settings. Our goal was to discover or verify guiding principles on how to choose hyper-parameteters for models to be trained on raw input audio data. Our focus was primarily the choice of filter numbers and their sequencing across layers. For these experiments, we fixed the model depth to seven and trained on AudioMNIST data.

Our first set of results are for experiments in which we randomly chose the number

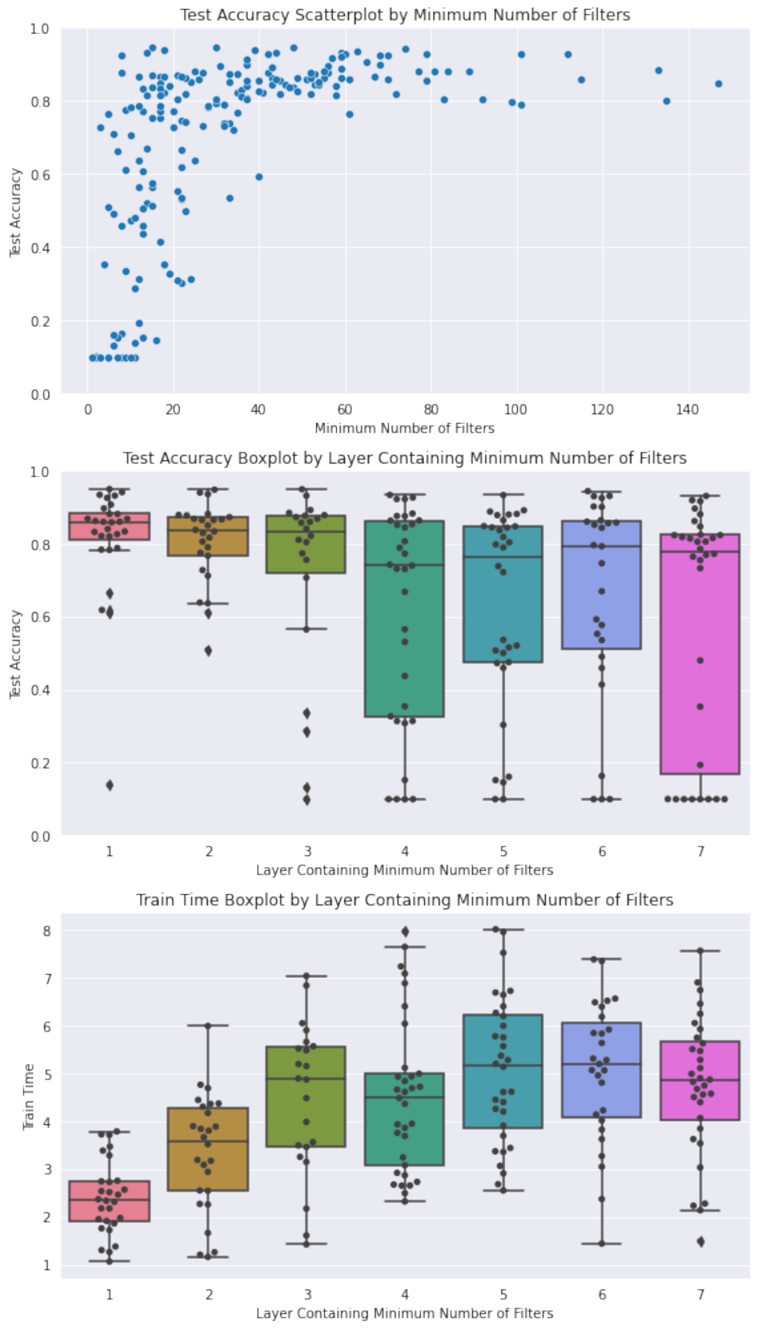


Figure 4.1: AudioMNIST random filter number sequence results, explored by minimum number of filters and the layer containing it: model test accuracies plotted by minimum number of filters (top); model test accuracy boxplot by layer containing minimum number of filters (middle); model train time (sec) boxplot by layer containing minimum number of filters (bottom)

of filters and their sequences. We examine these results to determine the impact that the number of filters has on performance (*i.e.*, test accuracy), specifically what the im-

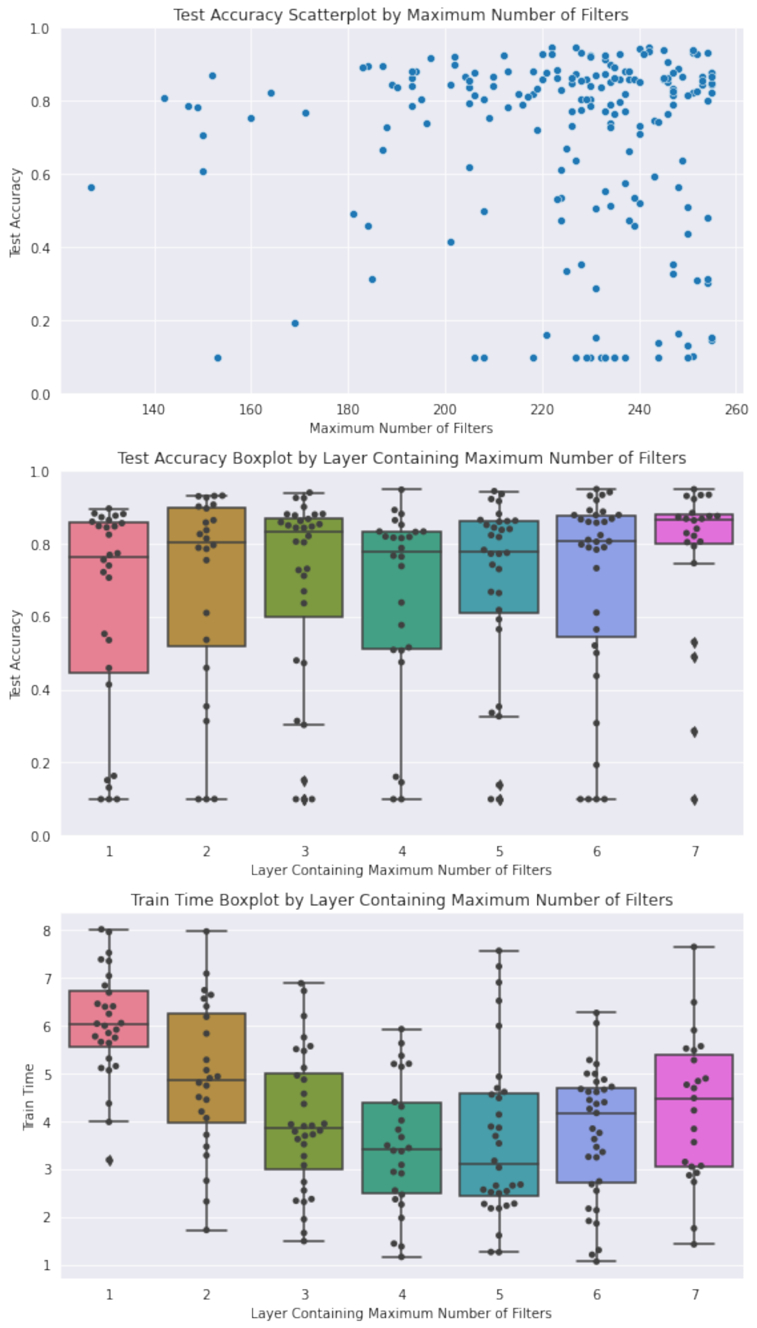


Figure 4.2: AudioMNIST random filter number sequence results, explored by maximum number of filters and the layer containing it: model test accuracies plotted by maximum number of filters (top); model test accuracy boxplot by layer containing maximum number of filters (middle); model train time (sec) boxplot by layer containing maximum number of filters (bottom)

pact of the smallest and largest number of filters is, as well as its placement. Figure 4.1 shows that: 1) a larger minimum number of filters correlates with better performance

(r = 0.56); and 2) the deeper the layer containing the minimum number of filters, the more variable and worse the performance is (compare $(\mu, \sigma) = (0.81, 0.16)$ for the minimum in the first layer versus $(\mu, \sigma) = (0.6, 0.33)$ for the minimum in the seventh layer), as well as the longer it takes to train a model ($(\mu, \sigma) = (2.4, 0.77)$ vs $(\mu, \sigma) = (4.74, 1.45)$).

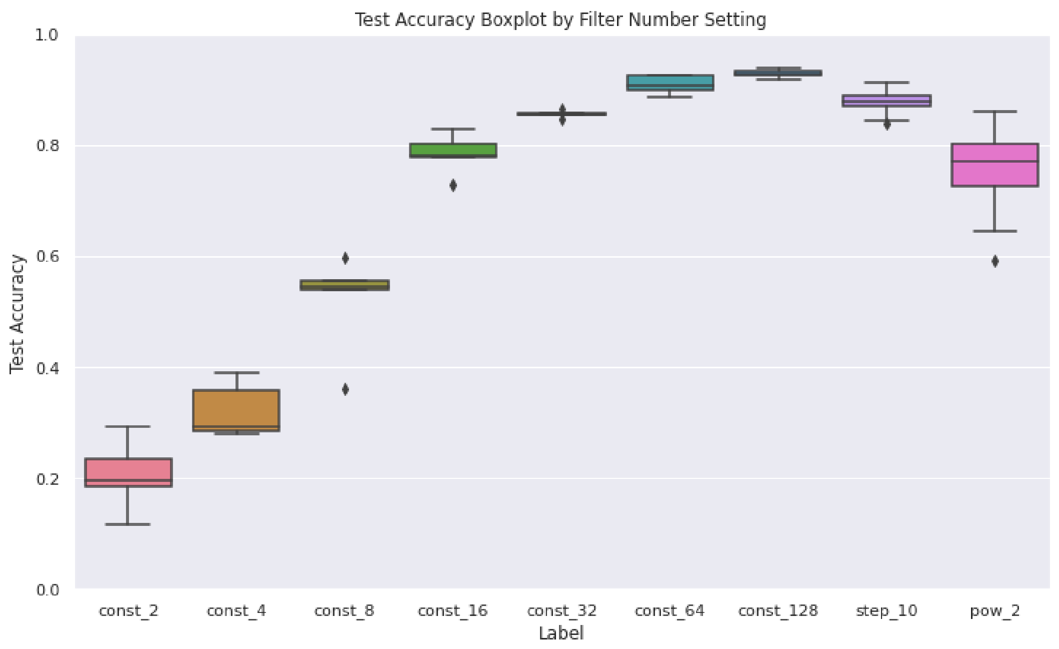


Figure 4.3: AudioMNIST accuracy results for models using the following number of filter sequences: a constant number of filters (const_2, const_4, const_8, const_16, const_32, const_64, const_128); a sequence increasing by increments of 10, step_10 (([5, 15, 25, 35, 45, 55, 65]); and a sequence increasing by powers of 2, pow_2 ([2, 4, 8, 16, 32, 64, 128])

In contrast, figure 4.2 shows that: 1) the maximum number of filters does not correlate with performance ($r = -0.05$); 2) when the maximum number of filters comes in the last layer, performance is slightly improved, with slightly reduced variation (compare $(\mu, \sigma) = (0.63, 0.3)$ for the maximum in the first layer versus $(\mu, \sigma) = (0.77, 0.22)$ for the maximum in the seventh layer); and 3) when the largest number of filters comes in the first layers, required training time tends to be higher ($(\mu, \sigma) = (6.04, 1.13)$ vs $(\mu, \sigma) = (4.26, 1.53)$). These results are consistent with the general custom in DL of increasing the number of filters per layer with depth.

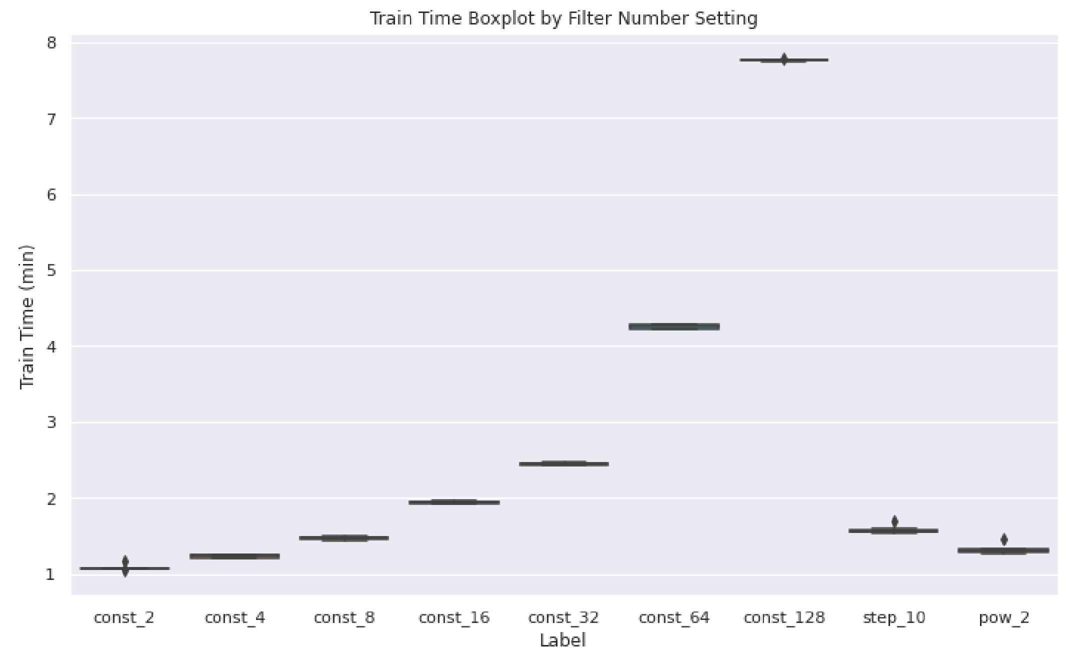


Figure 4.4: AudioMNIST train time for models using the following number of filter sequences: a constant number of filters (const_2, const_4, const_8, const_16, const_32, const_64, const_128); a sequence increasing by increments of 10, step_10 (($[5, 15, 25, 35, 45, 55, 65]$); and a sequence increasing by powers of 2, pow_2 ($[2, 4, 8, 16, 32, 64, 128]$)

Figures 4.3 and 4.4 shows results for constant number of filters, step10, and pow2 - again, for a seven layer CNN model. For a constant number of filters, performance improves with increasing number of filters, though train time also increases. However, we see that increasing the number of filters by ten (step10) performs comparably well, with considerable reduction in train time. Lastly, comparing step10 with pow2, results suggest that incrementing by ten rather than increasing by powers of two perform better, while requiring a minimal increase in train time.

As a consequence of these results, we chose for our main experiments below to use filter sequences with increments of ten. This additionally allows for continual increase of depth with linear rather than exponential growth, making model size and training time more manageable.

4.2 Raw Input Experiment Results

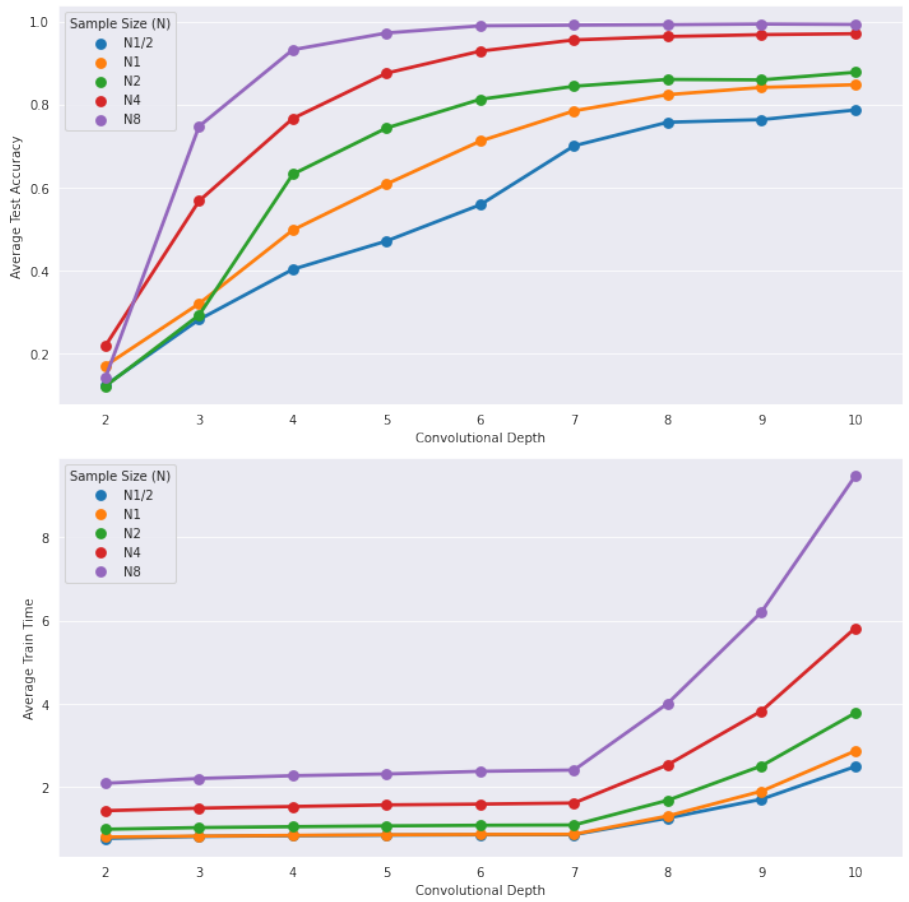


Figure 4.5: AudioMNIST: average accuracy and train time (sec) for varying sample size

We first compare performance on different sample sizes for AudioMNIST (Fig 4.5). We corroborate the expected improvement with larger sample size, and note that there is a considerable jump in performance between N2 (*i.e.*, 1000 samples) and N4 (*i.e.*, 2000 samples), with an increase of 0.09 in average test accuracy. This suggests that for a very structured problem with a balanced dataset of 10 classes like AudioMNIST, a dataset size equal or larger than that of N4, with 200 samples per class, would be ideal. We note that the real world datasets that we concern ourselves with rarely if ever reach such a size, especially when quality and careful labeling is concerned. Finally, we note that as dataset size decreases, deeper models are required to obtain similar results, up to an apparent upper bound for each sample size.

Next we look at the impact of data augmentation techniques, on AudioMNIST N1, XC and PR (Fig 4.6). Time Stretch is the only augmentation technique to improve average performance across all datasets. The effect is most visible for AudioMNIST and XC, where models of six layers or more outperformed those without augmentation by at least 0.1 in both cases, achieving peak average performance at ten layers on AudioMNIST and at eleven layers on XC (see Table 4.1 and 4.2 respectively). Furthermore, for AudioMNIST, Time Stretch augmentation yields results comparable to that obtained with a subsample of equivalent size, N3 (0.01 difference), as can be seen in Fig 4.7. As for PR, the gains are more subtle, performing similar to None on shallower networks, while improving on None by at least 0.04, achieving peak average performance also at ten layers (see Table 4.3).

The positive impact of Time Stretch is unsurprising for AudioMNIST - it is common knowledge that people can speak faster or slower without fundamentally altering the words and thus the meaning being transmitted. Thus the class of spoken digits can be said to be invariant to the speed of vocalization. The improvements on the

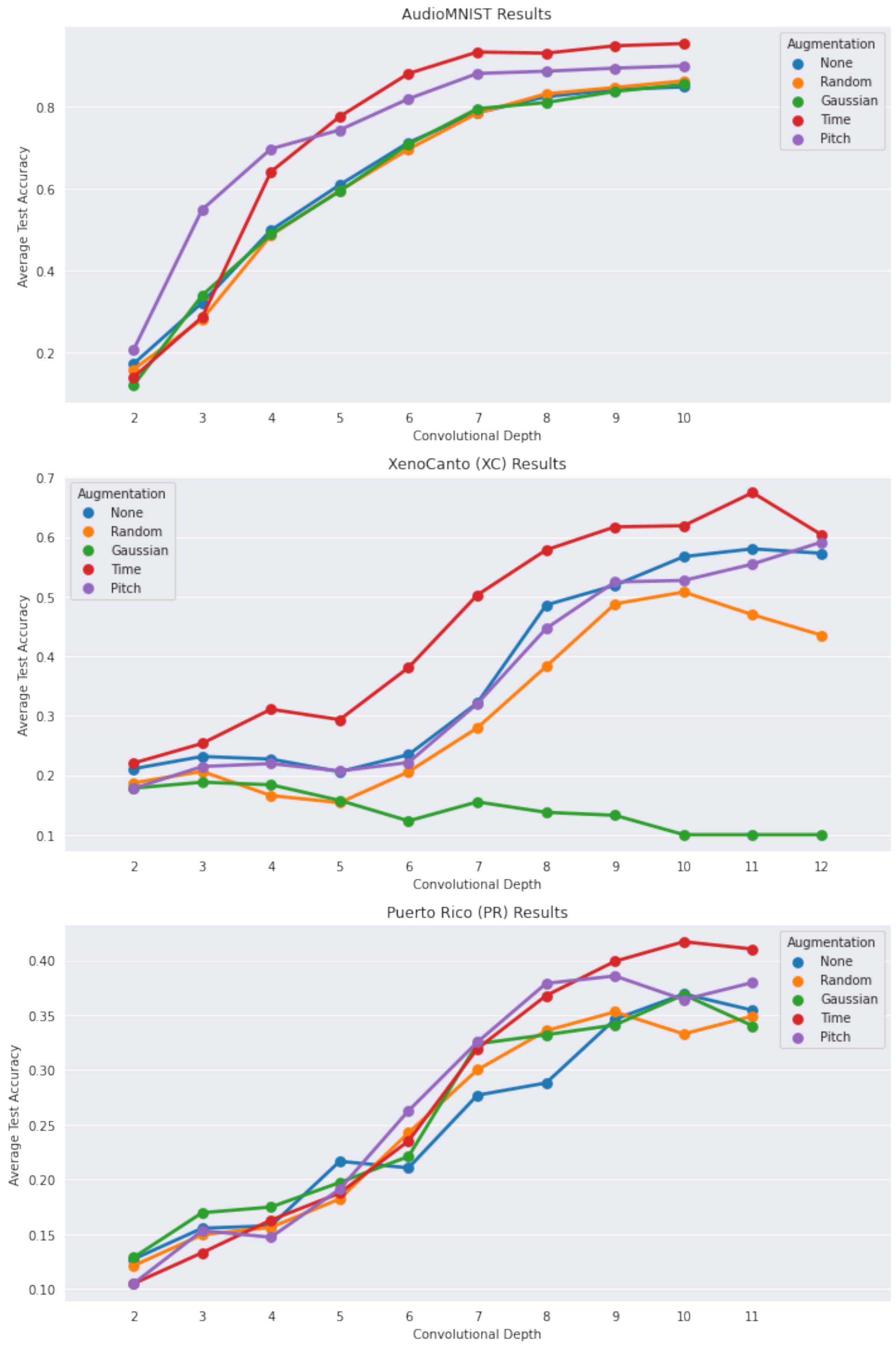


Figure 4.6: Average test accuracy results for augmentation experiments by datasets: AudioMNIST (top), XC (middle), and PR (bottom)

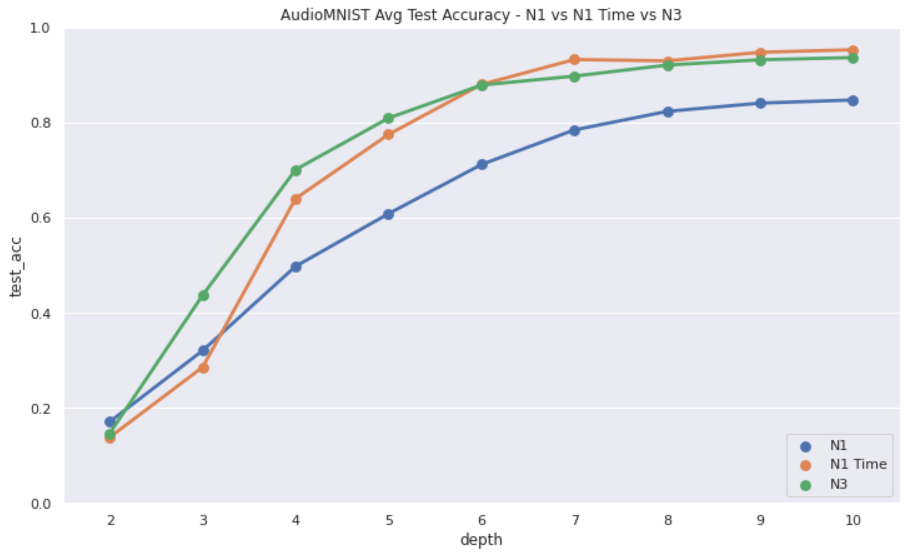


Figure 4.7: Test accuracy results for AudioMNIST, Time Stretch VS N3

two animal sound datasets similarly indicate that animal calls might also be invariant to the speed of vocalization.

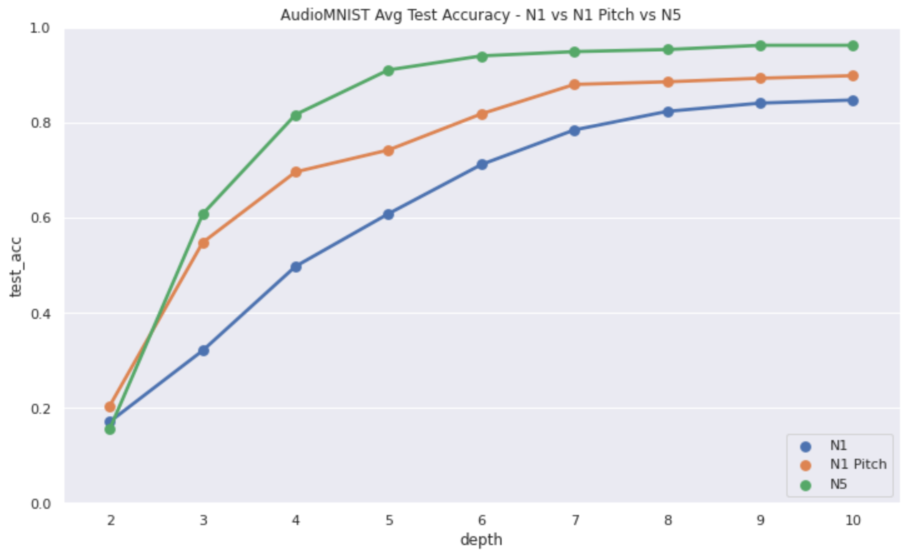


Figure 4.8: Test accuracy results for AudioMNIST, Pitch Shift VS N5

Pitch Shift most notably improves performance for AudioMNNIST, peaking at ten layers (Table 4.1). As with Time Stretch, human speech is known to have varying pitch ranges both within the speech of a single individual as well as across individuals, without changing the meaning of words. That is, human language is pitch invariant[1]. Thus this expected boost in performance is unsurprising. Unlike with Time Stretch, though, the gains in accuracy are less than those obtained by using the size equivalent subsample of N5 (0.06 difference) as can be seen in Fig 4.8.

On PR, Pitch Shift shows slight improvement over None, though within one standard deviation (compare Pitch L9 at 0.39 ± 0.026 versus None L10 0.37 ± 0.032). On XC, however, its performance is comparable to None, surpassing the latter only by 0.01 at its peak (0.59 ± 0.046 for Pitch L12 versus 0.58 ± 0.045 for None L11). These results suggest that bird calls in particular are not pitch invariant, and that animal calls in general are either not pitch invariant, or may differ depending on the species.

The poor performance of both Noise augmentations were surprising to us, since noise injection techniques are commonly believed to help make models more robust. Both Random Perturbation and Gaussian Noise had virtually no impact on performance for AudioMNIST and PR, while it hurt performance on XC (Fig 4.6). We hope further exploration of the nature of the recordings and fine-tuning of the hyperparameter settings for each particular dataset will yield better results, which we will explore in future work.

Finally, we look at the impact of maxpool filter sizes (Fig 4.9). What stands out the most is the improvement of performance with increasing size, up to a point. In AudioMNIST, we see steady improvements as we increase maxpool size, reaching a maximum for pool8 with 0.96 ± 0.011 at L14. Similar to results for increasing sub-

[1]By pitch invariant, we mean in relation to the baseline pitch of a speaker and not to sequences of pitches. Naturally, languages such as Mandarin Chinese use pitch phonemically, where the relative change in pitch from syllable to syllable helps determine meaning. Many other languages, such as English, use relative changes in pitch across words and phrases for intonation

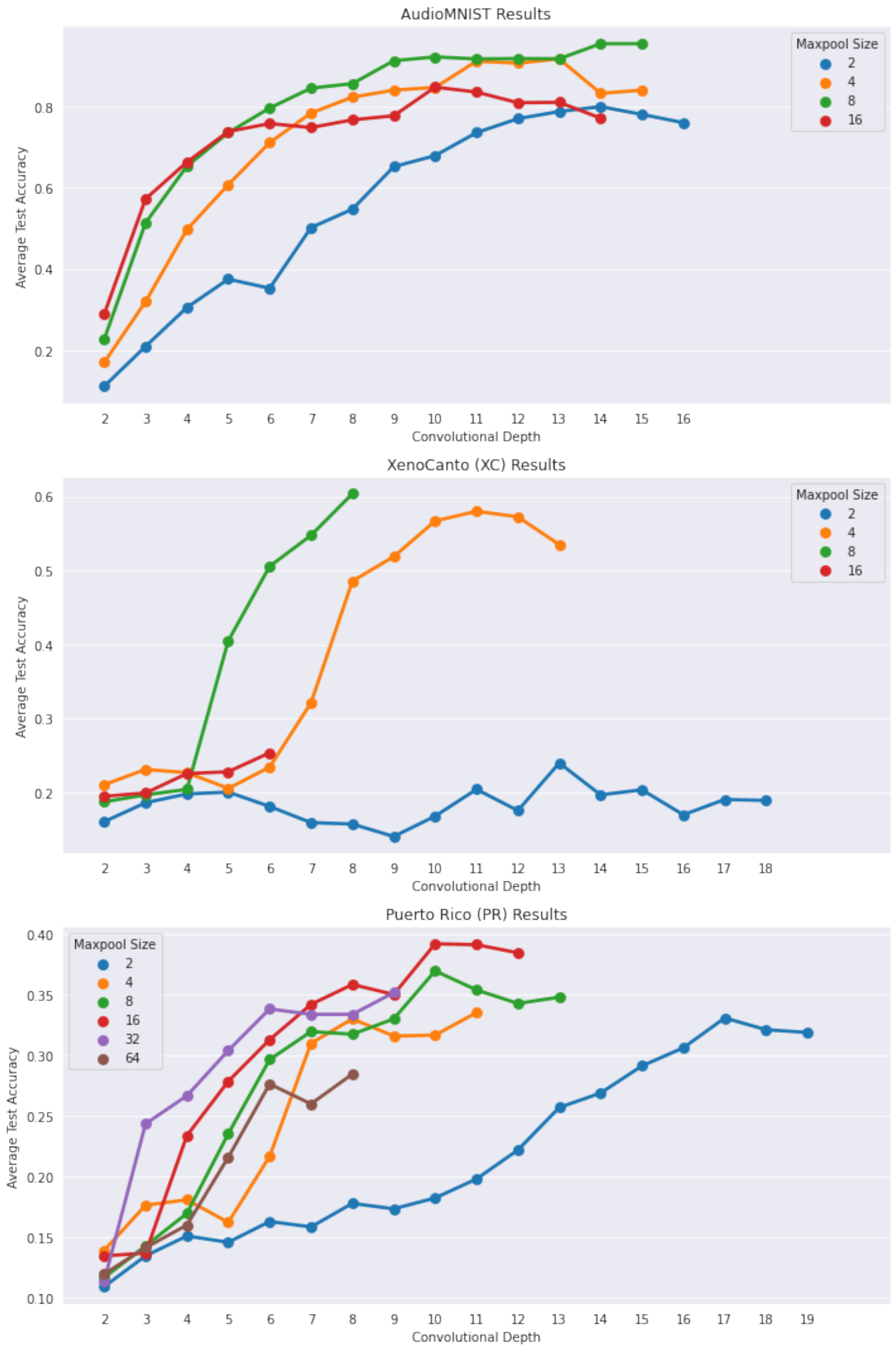


Figure 4.9: Test accuracy results for maxpool experiments by datasets: AudioMNIST (top), XC (middle), and PR (bottom)

sample size, shallower models with larger maxpool filters can achieve equal or better results than deeper models with smaller maxpool filters. For XC, we similarly see improved performance for pool8 and shallower models, though its peak performance of 0.6 ± 0.028 at L8 is comparable to the peak performance of 0.58 ± 0.045 for pool4 at L11. Lastly, for PR, performance improves up to pool16, achieving 0.39 ± 0.028 at L10.

These results were truly unexpected and we as yet have no explanation for this behavior, but suspect that it may be due to the periodic nature of sounds. It is possible that larger maxpool filters are behaving as a kind of maximum value downsampling applied on latent representations. After each convolution, maxpool layers are selecting the largest value within a window and forwarding it, effectively downsampling the latent representation of the audio produced by the previous convolutional layer. Additional experiments using strides in the convolutional layers instead failed to produce similar improvements, suggesting maxpool layers are playing a special role. Furthermore, the top performing models for larger maxpool filters all contain multiple consecutive convolutional layers without maxpool (11 for AudioMNIST, 2 for XC, and 5 for PR), also suggesting the benefit of an initial, un-pooled phase followed by pooled convolutional layers.

We conclude the analysis of raw input models by commenting on the best performing models per data set. In the case of AudioMNIST (Table 4.1), the best model trained on augmented data is Time Stretch, 0.03 higher than None. However, using larger maxpool filters achieves better (though still comparable) results, achieving 0.96 ± 0.014 with L14. The former requires more memory for the augmented data,

Augment/Pooling	Avg Acc	Stand Dev	Train Time	Depth
None/Pool2	0.8	0.047	2.1 min	14
None/Pool4	0.92	0.007	7.2 min	13
None/Pool8	**0.96**	**0.011**	14 min	14
None/Pool16	0.85	0.0088	8.9 min	10
Random/Pool4	0.86	0.014	3.8 min	10
Gaussian/Pool4	0.85	0.02	4.7 min	10
Time/Pool4	**0.95**	**0.014**	3.1 min	9
Pitch/Pool4	0.9	0.015	6.6 min	10

Table 4.1: AudioMNIST - Best Results by experiment (raw inputs)

while the latter requires more train time due to a deeper architecture, establishing a trade-off between resources. We will further discuss this later.

Augment/Pooling	Avg Acc	Stand Dev	Train Time	Depth
None/Pool2	0.24	0.036	5.9 min	13
None/Pool4	0.58	0.045	4.9 min	11
None/Pool8	0.6	0.028	7.2 min	8
None/Pool16	0.25	0.027	12 min	6
Random/Pool4	0.51	0.034	6.1 min	10
Gaussian/Pool4	0.19	0.027	4.6 min	3
Time/Pool4	**0.67**	0.031	13 min	11
Pitch/Pool4	0.59	0.046	34 min	12

Table 4.2: XC - Best Results by experiment (raw inputs)

In the case of XC, no model trained on un-augmented data surpasses or even approaches the best result of Time Stretch (see Table 4.2). In fact, as we shall see in the next section, Time performs best overall, beating even models trained on spectrograms. Thus, there is a clear preference for augmentation with XC.

Lastly, for PR, non-augmented models also failed to outperform Time Stretch, though given standard deviations, some are close enough for comparison (Fig 4.3). The best Time Stretch model achieves 0.42 ± 0.036 with L11 while the best non-augmented model (Pool16) achieves 0.39 ± 0.028 with L10. However, both results still

Augment/Pooling	Avg Acc	Stand Dev	Train Time	Depth
None/Pool2	0.33	0.022	2.3 min	17
None/Pool4	0.37	0.032	2.1 min	10
None/Pool8	0.37	0.035	5.8 min	10
None/Pool16	0.39	0.028	8.1 min	10
None/Pool32	0.35	0.031	8.1 min	9
None/Pool64	0.28	0.04	4.2 min	6
Random/Pool4	0.35	0.047	5.8 min	11
Gaussian/Pool4	0.37	0.039	5.5 min	10
Time/Pool4	**0.42**	0.036	5.5 min	10
Pitch/Pool4	0.39	0.026	5.6 min	9

Table 4.3: PR - Best Results by experiment (raw inputs)

perform relatively poorly, and no model trained on raw data manages an accuracy of 0.5 or more. As we shall see in the next section, only models trained on spectrograms manage to do so. Given the complexity of the dataset composed of recordings from various distinct types of animals (*i.e.*, using different biological hardware to produce sounds) with ample environment noise, the number of samples available might be too small to allow the model to effectively learn representations of each species.

4.3 Spectrograms

Fig 4.10 shows the results of experiments trained on spectrograms along with the default setting (pool4raw) and the best non-augmented raw input model for each of AudioMNIST, XC, and PR respectively, for comparison. These results paint an uneven picture. For AudioMNIST, the best non-augmented raw input model outperforms the best spectrogram model. For XC, the best non-augmented raw input model performs comparable to the best spectrogram model (0.6 ± 0.028 versus 0.61 ± 0.035), while the best with Time DA decisively outperforms the latter (see Table 4.4). How-

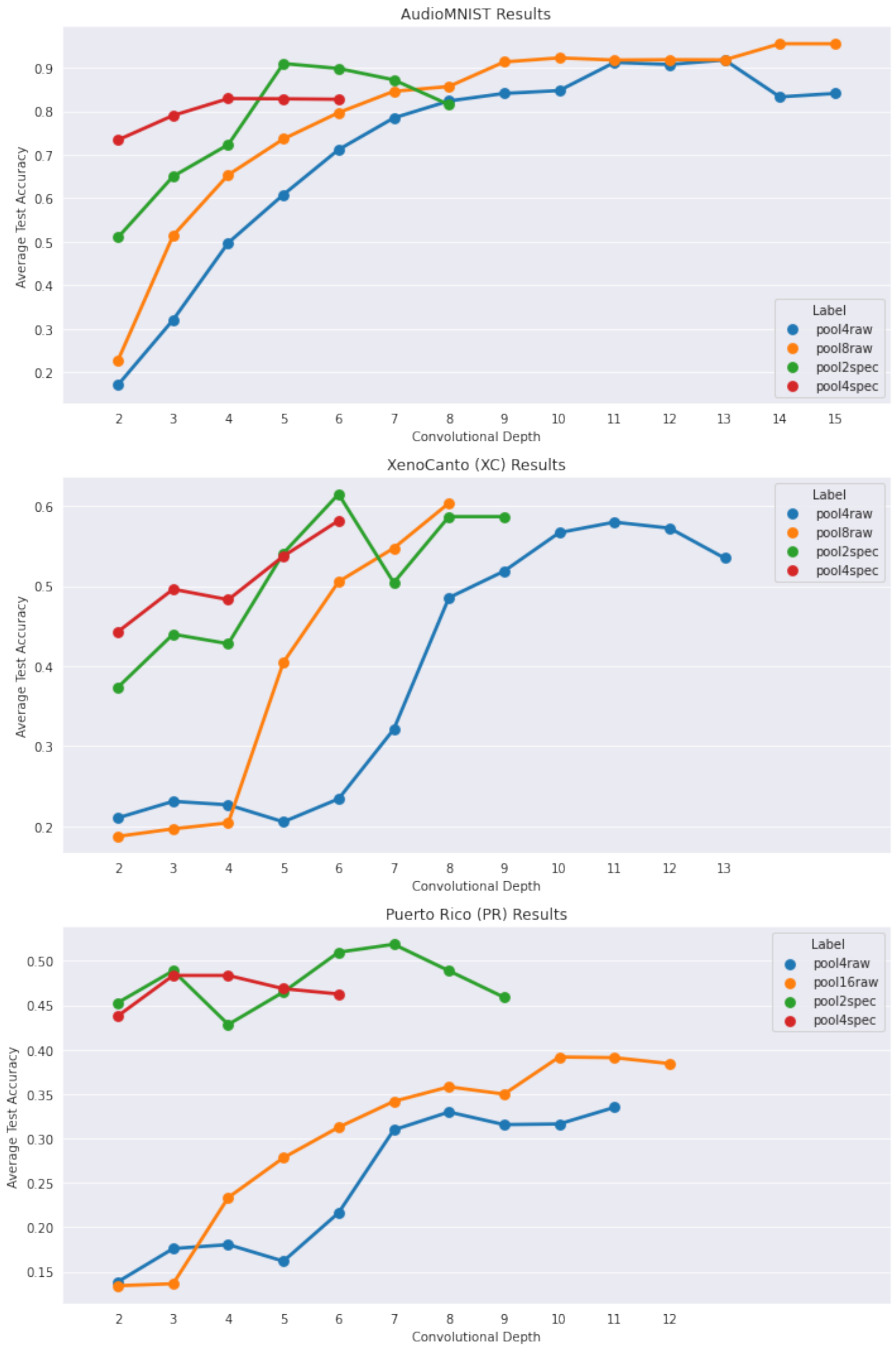


Figure 4.10: Test accuracy results for spectrogram vs raw input experiments by datasets: AudioMNIST (top), XC (middle), and PR (bottom)

ever, for PR, no raw input model outperforms the best spectrogram model, and this latter is the only one surpassing 0.5 in average accuracy.

	Avg Acc	Stand Dev	Train Time	Depth
AudioMNIST				
None Pool8 (raw)	**0.96**	0.011	14 min	14
None Pool2 (spec)	0.91	0.022	**0.7 min**	**5**
XC				
Time (raw)	**0.67**	0.031	13 min	11
None Pool2 (spec)	0.61	0.035	**2 min**	**6**
PR				
Time (raw)	0.42	0.036	5.5 min	10
None Pool2 (spec)	**0.52**	0.036	**1.1 min**	**7**

Table 4.4: Best Results by experiment (raw vs spec)

Since AudioMNIST is the simplest and most regular dataset, we are able to obtain good results even with only raw input data. There is thus little room left for improvement, and thus no special advantage to using spectrograms. This contrasts with PR, which is a highly complex dataset, for which the models are clearly not provided with enough samples per class to learn effective representations of the different animal calls. In such a case, the use of spectrograms successfully leverages human generated knowledge to improve performance. That said, even spectrogram models do not perform as well as we would like, much less reaching an ideal accuracy of 0.95 or higher. Both raw input and spectrogram results point to a need for more data, or further experimentation with other architecture features.

As for XC, we suspect its results differ from PR because all of the classes are bird species. As a result, the models most likely leverage all the training data to learn features that characterize bird calls in general alongside more specific features for each species. As a result, the best spectrogram and raw input models show comparable results (though spectrogram models perform considerably better even with the shallowest models, *i.e.*, L2-4).

4.4 Memory and Computational Constraints

When deciding how to improve models, availability of resources becomes a key limiting factor. Although collecting more data improves results, it is not always possible to do so. DA is often used to circumvent such limitations, but it comes at a price vis-a-vis resources - namely computation time and memory. Furthermore, using spectrograms rather than raw inputs can in turn reduce train time.

We first note in Figs 4.11, 4.12, and 4.13 the average running times for augmentation, maxpool, and spectrogram experiments respectively. For augmentation experiments, we note that with increase in dataset size due to augmentation, there is an increase in the baseline time it takes for the model to train. Thus more data means longer training time. Across all experiments, we notice that for the same model or augmentation setting, the training time remains stable for various layers, until at some depth it begins to increase with each additional layer. Said depth when training time first increases corresponds to the first model where an additional layer was added without a maxpool layer. Thus, adding convolutional layers without maxpool, though necessary to continue growing the model, contributes to an increase in train time. These increases happen at L8 for AudioMNIST, L10 for XC, and L9 for PR, for raw input and pool4.

In Tables 4.5, 4.6 and 4.7, we present the best models for each of the three datasets,

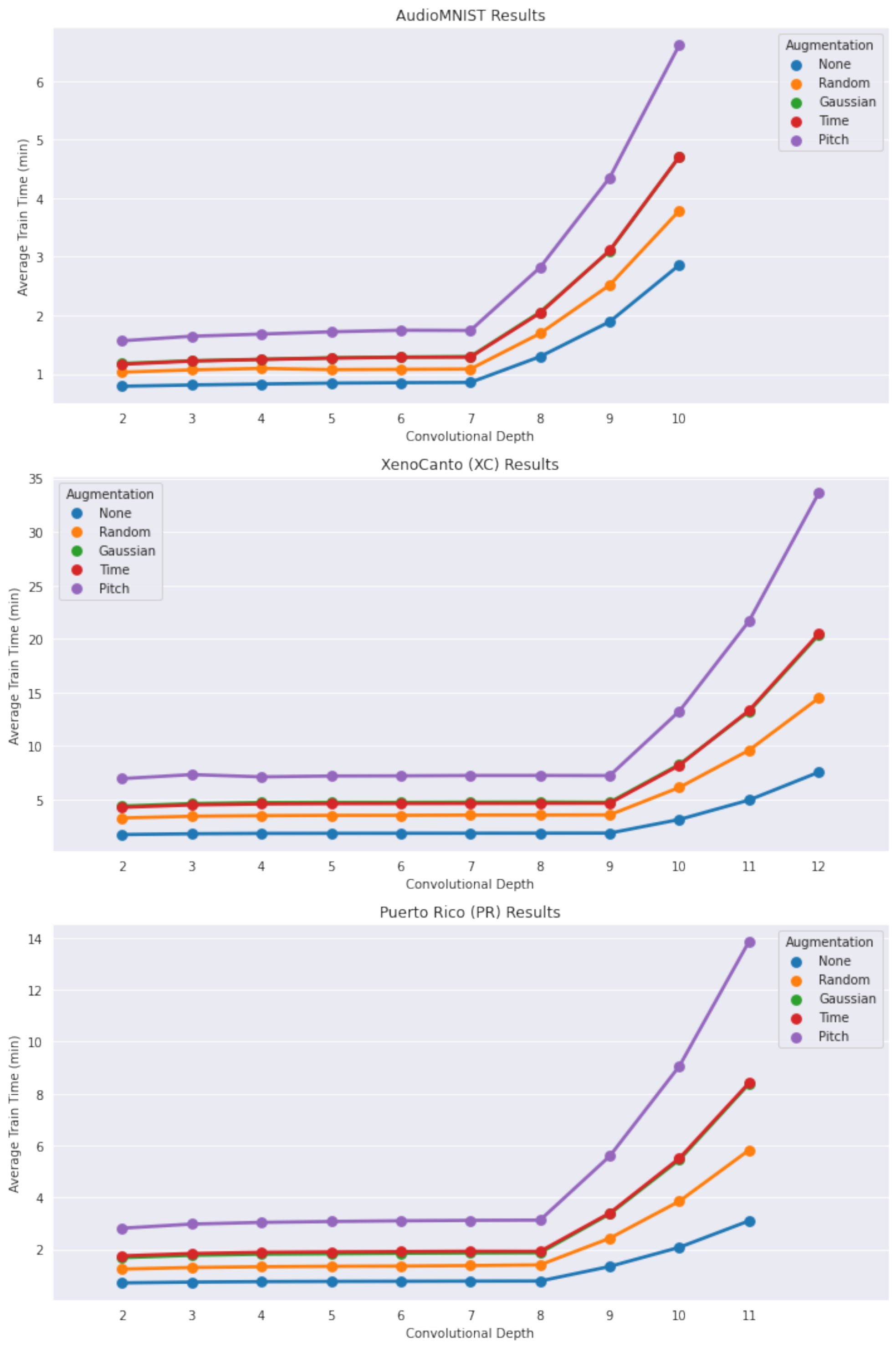


Figure 4.11: Average train time for Augmentation experiments by datasets: AudioMNIST (top), XC (middle), and PR (bottom) (note: Gaussian and Time overlap)

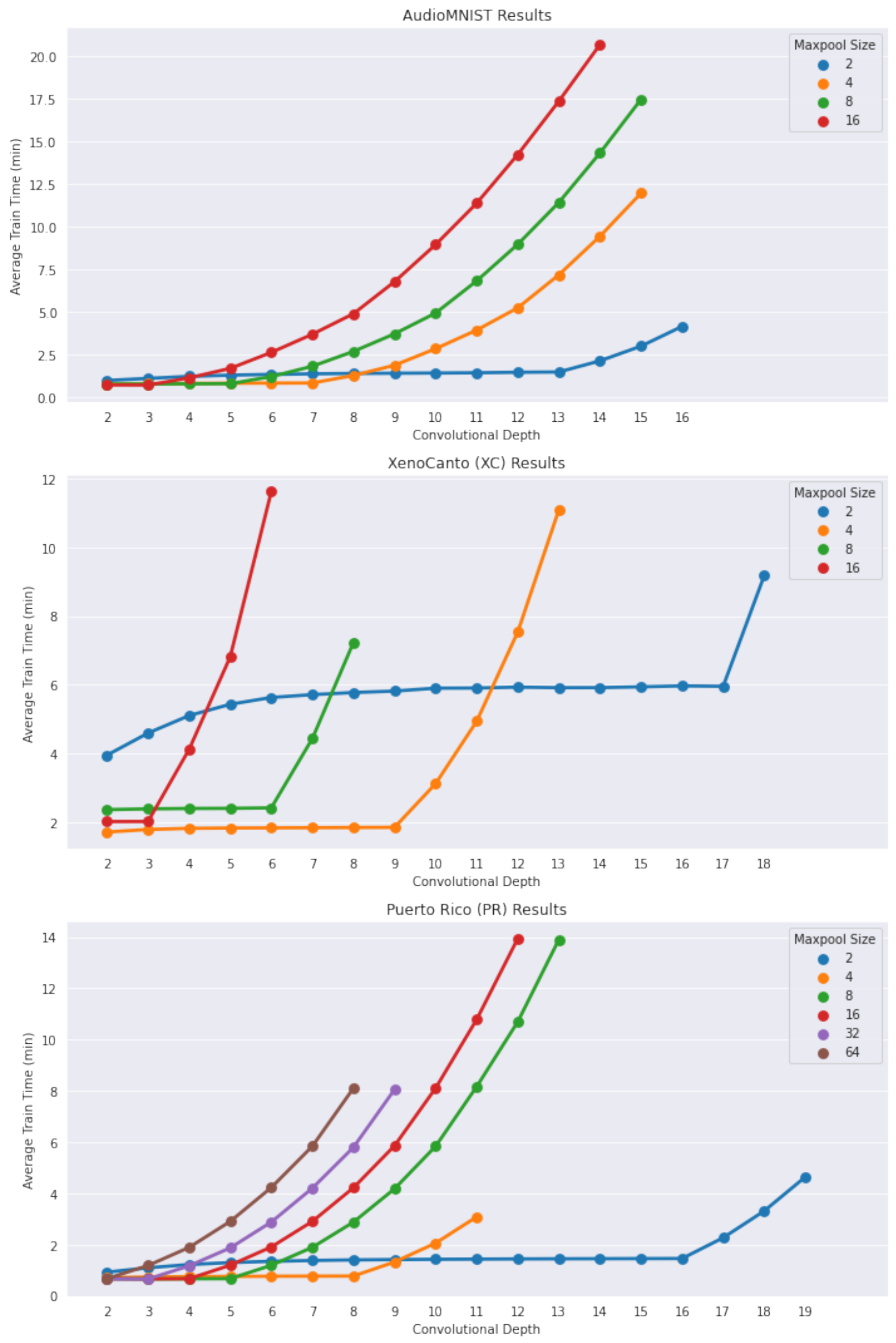


Figure 4.12: Average train time for maxpool experiments by datasets: AudioMNIST (top), XC (middle), and PR (bottom)

given computation time restriction (x-axis) and memory restriction (y-axis). The latter allows us to factor in augmentation methods, which cause an increase in training dataset size: None corresponds to the base case, $\times 1$; random perturbation to $\times 2$;

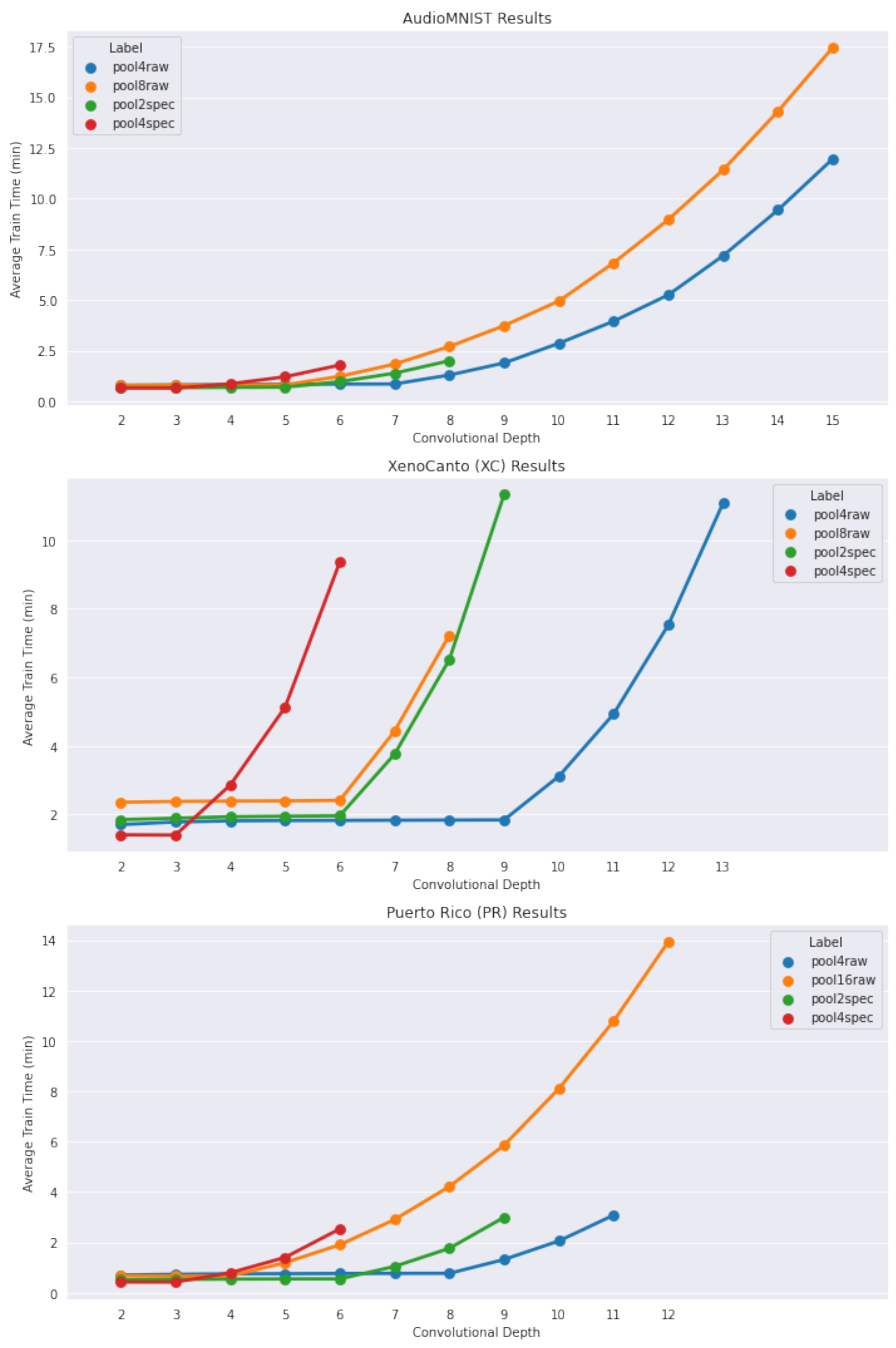


Figure 4.13: Average train time for spectrogram vs raw input experiments by datasets: AudioMNIST (top), XC (middle), and PR (bottom)

Time and Gaussian Noise to $\times 3$, and Pitch to $\times 5$. Note: for all three datasets, no model requiring more than 16 min outperformed the best model running within that time limit; hence, the best models at ≤ 16 min can be equally interpreted as the best models absent a time constraint.

	$\leq$ 1min	$\leq$ 2min	$\leq$ 4min	$\leq$ 8min	$\leq$ 16min
$\leq \times 5$	SpecP2L5	Time P4L7	Time P4L9	Time P4L9	P8L14
$\leq \times 3$	SpecP2L5	Time P4L7	Time P4L9	Time P4L9	P8L14
$\leq \times 2$	SpecP2L5	SpecP2L5	SpecP2L5	P8L11	P8L14
$\leq \times 1$	SpecP2L5	SpecP2L5	SpecP2L5	P8L11	P8L14

Table 4.5: AudioMNIST - Best Models Under Constraints

For AudioMNIST, we first notice that as training time limit increases, greater depth improves performance. More generally, spectrograms dominate the low and medium time limit regimes as well as the low data limit regime. Meanwhile, Time augmentation dominates the medium time limit regimes and higher memory limit regimes, while None with larger pooling size dominates in the absence of time constraints, though its performance is almost identical to that of the best Time model (0.96 vs 0.95).

	$\leq$ 1min	$\leq$ 2min	$\leq$ 4min	$\leq$ 8min	$\leq$ 16min
$\leq \times 5$	NA	Spec P2L6	Spec P2L6	Time P4L9	Time P4L11
$\leq \times 3$	NA	Spec P2L6	Spec P2L6	Time P4L9	Time P4L11
$\leq \times 2$	NA	Spec P2L6	Spec P2L6	Spec P2L6	Spec P2L6
$\leq \times 1$	NA	Spec P2L6	Spec P2L6	Spec P2L6	Spec P2L6

Table 4.6: XC - Best Models Under Constraints

For XC, we first note the overall preference for spectrogram models. These dominate both low to mid memory and time limit regimes. However, in the mid to high memory and time limit regimes, Time augmentation models with raw inputs performed best. Note: no model for XC had average training time $\leq$ 1min.

For PR, there is no question that spectrogram models are the overall preference, with a slight increase in depth with an increase in the time limit, though the im-

	$\leq$ 1min	$\leq$ 2min	$\leq$ 4min	$\leq$ 8min	$\leq$ 16min
$\leq \times 5$	Spec P2L6	Spec P2L7	Spec P2L7	Spec P2L7	Spec P2L7
$\leq \times 3$	Spec P2L6	Spec P2L7	Spec P2L7	Spec P2L7	Spec P2L7
$\leq \times 2$	Spec P2L6	Spec P2L7	Spec P2L7	Spec P2L7	Spec P2L7
$\leq \times 1$	Spec P2L6	Spec P2L7	Spec P2L7	Spec P2L7	Spec P2L7

Table 4.7: PR - Best Models Under Constraints

provement is minimal (0.51 for Spec P2L6 versus 0.52 for Spec P2L7). Given the complexity of the dataset, it stands to reason that PR contains too few samples to effectively learn useful features for classification. This then makes the use of spectrograms preferable, leveraging expert knowledge and handcrafted features. Having said that, even the performance of spectrograms leaves much room for improvement.

Chapter 5

Conclusion and Future Work

In our experiments, we tested a basic convolutional model for animal sound classification using raw inputs. Our goal was to ascertain the performance of such models for small datasets, and to evaluate the effectiveness of data augmentation techniques and different hyper-parameter settings on task performance. We performed the experiments on three distinct datasets, each at a different level of data complexity and difficulty: AudioMNIST, XC, and PR.

With the simplest dataset, AudioMNIST, we first corroborated two common assumptions: 1) more data improves performance; and 2) deeper models perform better (with potential bounds). We also replicate expected improvements for speech data using Time Stretch and Pitch Shift augmentations.

Across all three datasets, our results show that Time Stretch is the only augmentation to produce improvements in performance. Similarly, increasing pooling size improves performance, up to a limit that is dataset dependent: size 8 for AudioMNIST and XC, size 16 for PR. However, for both XC and PR, Time Stretch outperformed models with larger pooling sizes.

When comparing results between raw and spectrogram inputs, we obtained mixed results. For AudioMNIST and XC, larger pooling size models performed comparable to spectrograms, and for XC Time Stretch augmentation outperformed both larger pooling sizes and spectrograms. However, for PR, the more complex dataset, spectrogram models handily outperformed all raw models. Furthermore, even for XC, spectrogram models are preferred when there are strict memory and compute time limits.

In future work, we will experiment with more modified versions of CNNs - such as incorporating attention to train deeper models or using dilated convolutions-, with other model architectures such as RNNs, and with models pre-trained on raw inputs from larger, more diverse datasets - such as auto-encoders. Furthermore, we will repeat our experiment with larger datasets - mostly non animal sound - to further corroborate the superior performance of spectrograms. One possible hypothesis is that improved performance is due to the small dataset size regime: given scarce data, feeding the models data features such as spectrograms lessens the learning load of the model and boosts performance - an advantage that might be overcome with enough data. We thus hypothesize that the advantage of spectrograms would diminish when data size is large enough in relation to data complexity.

Bibliography

[1] Miguel A Acevedo et al. “Automated classification of bird and amphibian calls using machine learning: A comparison of methods”. In: *Ecological Informatics* 4.4 (2009), pp. 206–214.

[2] Antreas Antoniou, Amos Storkey, and Harrison Edwards. “Data augmentation generative adversarial networks”. In: *arXiv preprint arXiv:1711.04340* (2017).

[3] Anand Avati. *Stanford CS229: Machine Learning — Summer 2019 — Lecture 21 - Evaluation Metrics.* 2019. URL: `https://www.youtube.com/watch?v=Lb1-iNIOLBw` (visited on 05/09/2022).

[4] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. “An empirical evaluation of generic convolutional and recurrent networks for sequence modeling”. In: *arXiv preprint arXiv:1803.01271* (2018).

[5] Sören Becker et al. “Interpreting and Explaining Deep Neural Networks for Classification of Audio Signals”. In: *CoRR* abs/1807.03418 (2018). arXiv: `1807.03418`.

[6] Xuanyi Chen et al. “The human language system does not support music processing”. In: *BioRXiv* (2021).

[7] Jan Chorowski et al. “Unsupervised speech representation learning using wavenet autoencoders”. In: *IEEE/ACM transactions on audio, speech, and language processing* 27.12 (2019), pp. 2041–2053.

[8] Ekin D Cubuk et al. “Autoaugment: Learning augmentation strategies from data”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 2019, pp. 113–123.

[9] Terrance DeVries and Graham W Taylor. “Dataset augmentation in feature space”. In: *arXiv preprint arXiv:1702.05538* (2017).

[10] Jesse Engel et al. “Neural audio synthesis of musical notes with wavenet autoencoders”. In: *International Conference on Machine Learning.* PMLR. 2017, pp. 1068–1077.

[11] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* ” O’Reilly Media, Inc.”, 2019.

[12] Douglas C. Giancoli. *Physics for Engineers and Scientists.* Pearson Upper Saddle River, NJ, 2009. ISBN: 0131495089.

[13] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

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

[15] Awni Hannun et al. "Deep speech: Scaling up end-to-end speech recognition". In: *arXiv preprint arXiv:1412.5567* (2014).

[16] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[17] Shawn Hershey et al. "CNN architectures for large-scale audio classification". In: *2017 ieee international conference on acoustics, speech and signal processing (icassp)*. IEEE. 2017, pp. 131–135.

[18] Irina Higgins et al. "Towards a definition of disentangled representations". In: *arXiv preprint arXiv:1812.02230* (2018).

[19] Navdeep Jaitly and Geoffrey E Hinton. "Vocal tract length perturbation (VTLP) improves speech recognition". In: *Proc. ICML Workshop on Deep Learning for Audio, Speech and Language*. Vol. 117. 2013.

[20] Gareth James et al. *An introduction to statistical learning*. Vol. 112. Springer, 2013.

[21] Stefan Kahl et al. "Overview of BirdCLEF 2021: Bird call identification in soundscape recordings". In: *Working Notes of CLEF 2021 - Conference and Labs of the Evaluation Forum*. 2021.

[22] Pieter-Jan Kindermans et al. "Learning how to explain neural networks: Patternnet and patternattribution". In: *arXiv preprint arXiv:1705.05598* (2017).

[23] Tom Ko et al. "Audio augmentation for speech recognition". In: *Sixteenth Annual Conference of the International Speech Communication Association*. 2015.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.

[25] Brian McFee et al. "librosa: Audio and music signal analysis in python". In: *Proceedings of the 14th python in science conference*. Vol. 8. Citeseer. 2015, pp. 18–25.

[26] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997. ISBN: 9780070428072.

[27] Noam Mor et al. "A universal music translation network". In: *arXiv preprint arXiv:1805.07848* (2018).

[28] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012. ISBN: 9780262018029.

[29] Preetum Nakkiran et al. “Deep double descent: Where bigger models and more data hurt”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2021.12 (2021), p. 124003.

[30] Loris Nanni, Gianluca Maguolo, and Michelangelo Paci. “Data augmentation approaches for improving animal audio classification”. In: *Ecological Informatics* 57 (2020), p. 101084.

[31] Aaron van den Oord et al. “Wavenet: A generative model for raw audio”. In: *arXiv preprint arXiv:1609.03499* (2016).

[32] Daniel S Park et al. “Specaugment: A simple data augmentation method for automatic speech recognition”. In: *arXiv preprint arXiv:1904.08779* (2019).

[33] Paolo Prandoni and Martin Vetterli. *Signal processing for communications.* EPFL press, 2008. ISBN: 9782940222209.

[34] Anton Ragni et al. “Data augmentation for low resource languages”. In: *INTERSPEECH 2014: 15th Annual Conference of the International Speech Communication Association.* International Speech Communication Association (ISCA). 2014, pp. 810–814.

[35] Justin Salamon and Juan Pablo Bello. “Deep convolutional neural networks and data augmentation for environmental sound classification”. In: *IEEE Signal Processing Letters* 24.3 (2017), pp. 279–283.

[36] Steffen Schneider et al. “wav2vec: Unsupervised pre-training for speech recognition”. In: *arXiv preprint arXiv:1904.05862* (2019).

[37] Connor Shorten and Taghi M Khoshgoftaar. “A survey on image data augmentation for deep learning”. In: *Journal of Big Data* 6.1 (2019), pp. 1–48.

[38] Vivienne Sze et al. “Efficient processing of deep neural networks: A tutorial and survey”. In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[39] John Towns et al. “XSEDE: accelerating scientific discovery”. In: *Computing in science & engineering* 16.5 (2014), pp. 62–74.

[40] Aaron Van den Oord, Yazhe Li, and Oriol Vinyals. “Representation learning with contrastive predictive coding”. In: *arXiv e-prints* (2018), arXiv–1807.

[41] Qingsong Wen et al. “Time series data augmentation for deep learning: A survey”. In: *arXiv preprint arXiv:2002.12478* (2020).