Machine Listening

Using deep learning to build an audio classifier to recognise birdsong

Capstone Project Report for the Udacity Machine Learning Engineer Nanodegree

Jaron Collis

December, 2016

1 Definition

1.1 Project Overview

The goal of this project is to build a general audio classifier capable of recognising different sounds. Initially, the sounds will be ambient noises from an urban environment, and then, a specific kind of sound - birdsong. Eventually, the ultimate goal is to create a system that can be given a previously unheard recording of a bird singing and determine its species. I consider this an intriguing and worthwhile challenge for several reasons:

- **Diversity** - Birdsong is an everyday sound that many people can readily recognise - but only after they've learned the characteristic thrills and chirps of each species. The challenge is increased by the fact that instances of birdsong can contain considerable variation, even amongst those of the same species.
- **Noisy** - Birdsong occurs in the wild, so is unavoidably noisy. Successful recognition needs to cope not only with background noise, but with the sound of other birds singing nearby.
- **Usefulness** - How many times have you stopped to listen to a beautiful birdsong and wondered: what is that singing? Wouldn't it be awesome if an app on your phone could listen in and identify the performer?
- **Originality** - there are a handful of apps that claim to be able to listen to and identify birdsong, but their users report inconsistent results, and none of those apps employ deep learning. This project will provide an opportunity to explore how good a deep neural network is at audio classification.

If this sounds like Shazam for Birds, you're thinking in the right direction - but not quite. Remember, once recorded, songs have no variation, a fact that allows music identification services to recognise tracks by creating and comparing acoustic fingerprints. But there's too much variation in birdsong for this approach, some birds might chirp faster or slower depending on their surroundings, or quieter or louder - birdsong is after all, a medium of communication. Each bird might also have idiosyncratic trills and chirps, added to all that, there might be unpredictable background noise too.

This challenge of identifying something that can appear in countless variations in any number of unpredictable noisy environments sounds a lot like the one faced by those building image recognition systems. Those working in image detection know you can't just create a visual fingerprint of a single object, because it occurs in too many different forms, varieties and contexts. But deep learning has already proved very effective at image recognition, so I thought I'd investigate: might it equally effective at audio recognition too?

One of the prerequisites of deep learning (and machine learning in general) is a high quality collection of data that can be used for training. For this project, I would need some annotated audio datasets, consisting of short sound clips and accompanying labels that tell us the subject of the recording. This would identify each sound clip as belonging to one of a finite set of categories, and enable the problem to be tackled as a supervised learning task.

I decided to begin by using the UrbanSound8K data set - a collection of 8732 short clips covering 10 different sounds from urban environments. Whilst this set doesn't contain any examples of birdsong, I chose it because the distinctive ambient sounds would enable me to explore the process of extracting features from raw audio recordings, which is an absolutely fundamental task to any audio classifier, regardless of what subjects are being identified.

Later, I would use what I'd developed to train a model on two bird detection datasets, FreeField and Warblr, which were released as part of the 2016 Bird Detection Challenge. Unlike the UrbanSound8K data, which had 10 possible labels, these files had binary labels - birds were either present or absent. So whilst this wouldn't result in a classifier capable of distinguishing between species it would enable me to validate my approach.

1.2 Problem Statement

The essence of the problem is this: given a never-before-heard recording, how can a system be trained to identify what it is actually listening to?

Let's try to refine this problem. The input format will be the digital representation of the original analogue waveforms, just as would be recorded by a microphone, encoded using pulse-code modulation and stored as (lossless) WAV files.

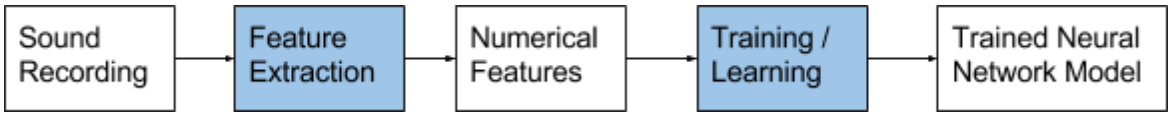
From this input, we'll need to extract features. But sound recordings are not spreadsheets, with their data neatly organised into rows and columns of known significance. Say we sample two recordings of equal duration, creating 2000 discrete floating point numbers; we can not meaningfully compare the 1000th number of one recording with the 1000th number of another. This is because we have no way of telling whether the number at any particular point in time is signal, silence or noise.

Instead, we must abstract away from individual numbers, and consider data values in the context of many of its neighbours. This will grant us the ability to start spotting patterns, and permit us to generalise - the fundamental requirement in machine learning.

The challenge then, is to find measurable properties that differ in dissimilar recordings and are alike in those from similar sources. The complexity of feature extraction makes this problem an attractive application of deep learning, whose hierarchical nature makes it capable of automated feature learning.

Once features have somehow been extracted, the question becomes how can we use them to train a model, and then use what we've learned to generate predictions?

Next, we'll want to tune the model produced, to achieve the best possible accuracy. So we'll need to consider how we can measure successful classification - and how can the performance of the model be optimised. Putting all this together suggests a processing pipeline that looks like this:



Section 2 will discuss some techniques that could be used at each of stage of the processing pipeline, whilst section 3 will describe the actual implementations I used. Section 4 will then discuss the results achieved, and section 5 will finish with a discussion of what has been built, lessons learned, and future directions.

1.3 Metrics

This project involves classifying data that is labelled - that is, we will always know the true class of each audio recording. This means we can evaluate the results produced using standard supervised learning metrics, such as F-score and the Receiver Operating Characteristic - Area Under Curve (ROC AUC).

The F-Score is a commonly used measure of classification accuracy that gives equal weight to **precision** (ratio of true positive over all positives) and **recall** (the ratio of true positives over true positives plus false negatives). It is based on the intuition that a good classifier should maximise both precision and recall simultaneously. So one with a reasonably good precision and recall will score better than one that has extremely good performance on just one.

A more sophisticated metric is the ROC AUC, which represents how far the classifier is from perfect performance. ROC curves are typically used in binary classification, so to apply this measure to multi-class classification, it will be necessary to convert the predictions into booleans representing positive or negative predictions for each class. A process called micro-averaging then enables a composite ROC curve to be created.

In supervised learning problems, knowing the true labels allows us to compose a confusion matrix, where each column of the matrix represents the number of instances in a predicted class while each row represents the instances in an actual class. As the name suggests, this depiction is particularly useful for determining where the system is confusing two classes (commonly mislabelling one as another).

Other metrics that are particularly important when evaluating training performance are the neural network model's loss and accuracy. By calculating both values for the training and validation sets we can gain an insight into whether the model is overfitting.

2 Analysis

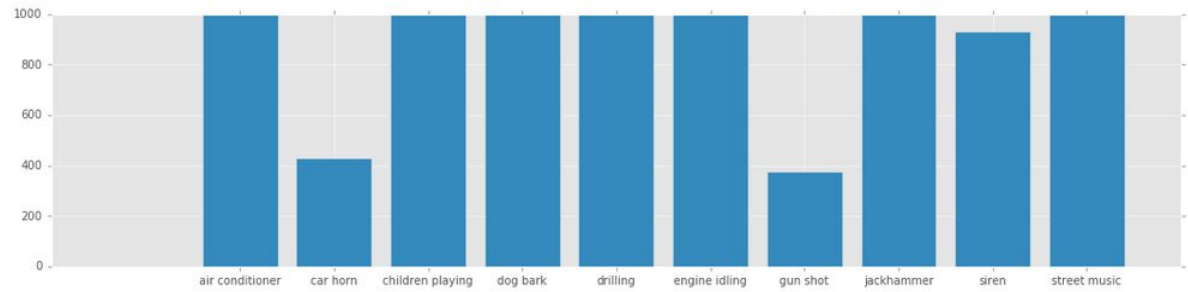
2.1 Data Exploration

- The code for exploring this dataset is available online in this Jupyter notebook

In this section I'm going to discuss the nature of the audio samples provided by the UrbanSound8K data set, a labelled collection of 8732 short clips covering 10 different sounds from urban environments. Most of the samples are stereo recordings, sampled at 48000 hertz, but to reduce the quantity of data processed, I've used default settings of the librosa library, which converts the samples to a single (mono) channel, and down-samples each to 22050 hz, which seems to be considered a good compromise quality for audio analysis.

Once loaded into memory, the audio file is stored as a 1-dimensional numpy array, representing a time series with a single floating point number that represents the change in the audio signal from one sample point to the next, (see this article for a nice explanation of the process involved). Hence a 1 second sample would be represented as 22050 floats, and a 4 second sample would be 4 times longer, with 88200 values. This is a considerable quantity of data for each sample, to make processing tractable on a laptop, it was clear to me that some kind of dimensionality reduction was not only desirable, but essential. This idea is discussed further in Section 3.

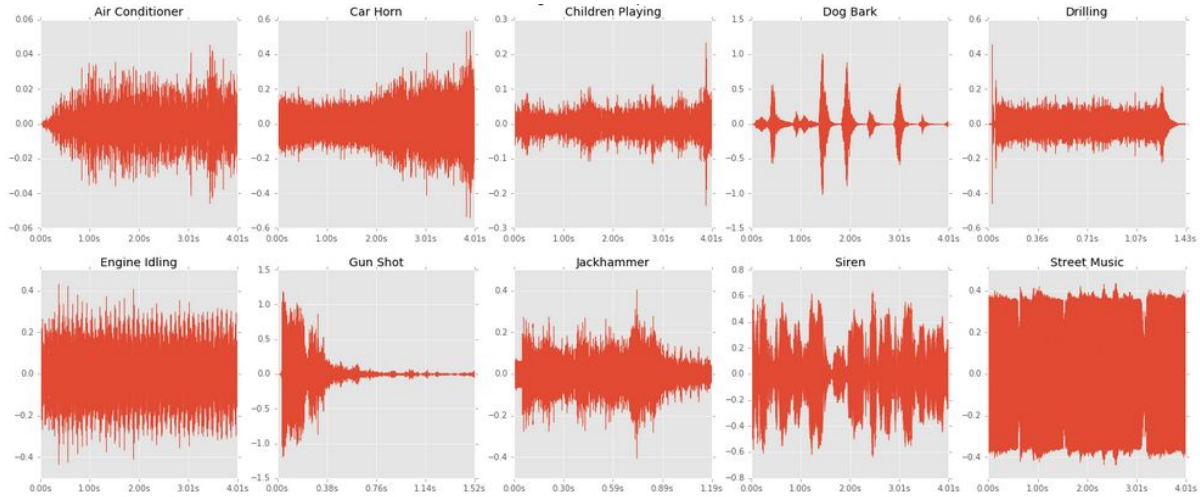
Another exploration is checking the balance of the dataset - is there a similar number of every class label or some classes that appear more much often and some that are rare? This is useful to check because of the Accuracy Paradox - we could inadvertently achieve good performance on just one class with many instances, and poor performance on all others, yet still seem to be achieving accurate results. The F1 accuracy score is a good measure for imbalanced data, as it is a weighted measure of both true and false positives, regardless of their class.



But, as we can see from this distribution of the classes, we have a good balance, with only two 2 classes under-represented: car horn and gun shot.

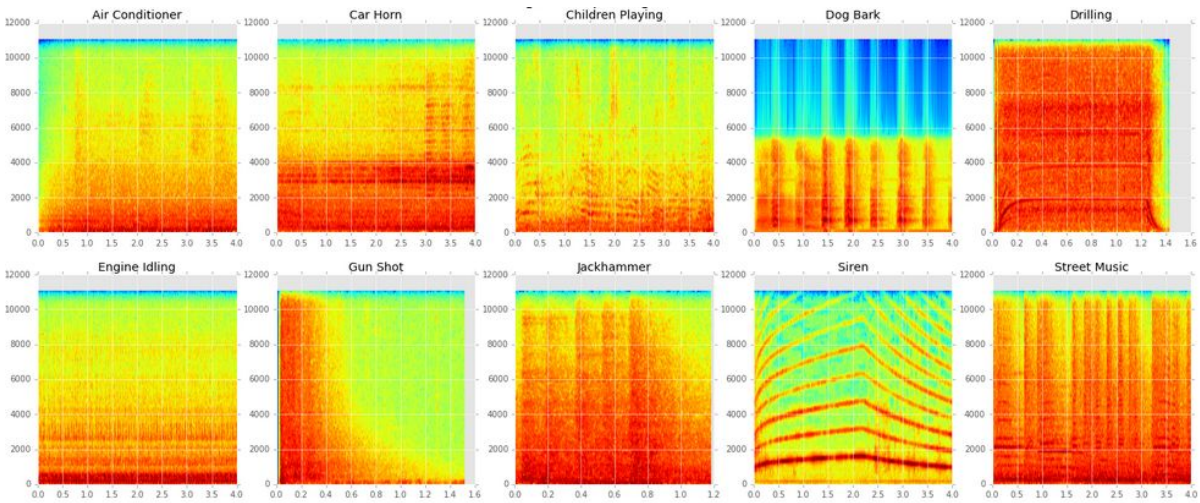
2.2 Exploratory Visualization

Once in memory, a common visualisation for audio recordings is the waveform plot, which depicts the amplitude (relative loudness) of the sound at each successive time interval, this is what you'll see if you load an audio file into a sound editor like Audacity. Here are the waveforms for a randomly chosen example of each of the dataset's 10 classes:



In these plots, time is on the horizontal axis and amplitude on the vertical. We can see that some of the samples, like a dog barking, have a shape that is distinctively different from the others, which our classifier might be able to utilise to distinguish between them. Others like air-conditioning and engine idling are superficially more similar, and might be more difficult to tell apart - even for a human listener.

Matplotlib provides an alternative visualisation method called spectrogram that calculates and plots the different intensities of the frequency spectrum. This creates a different depiction of each sound:



Visually, it seems slightly easier to distinguish between the different classes with this visualisation, and indeed the intensities of frequencies will be the basis of the features I intend to extract. The notebook contains a variation of this visualisation using Librosa's log power spectrogram plotting.

These plots illustrate the complexity of the data we need to process. In typical supervised learning projects, every example in the training set has the same features, providing comparable values. But this is not the case with media such as audio or images, because any two files will not contain a pair of values that are semantically equivalent. Even if two audio samples had the same size, we can't meaningfully compare the respective value at an arbitrary point in time. It would be like an image classifier trying to distinguish between two photos by comparing the pixels found in each at the same arbitrary location.

Consequently, in audio processing we can not consider each data value in isolation, and can not determine what is signal and what is noise merely by looking at the quantiles of a frequency distribution. Instead, the salient information is the patterns of values, spread over wide regions of the recording. This means we must consider how the signal changes over time, and what patterns can be identified. How this can be achieved will be discussed next.

2.3 Algorithms and Techniques

In an influential paper, Professor Leo Breiman discussed the basis of the fundamental task in machine learning: the creation of a model that represents the complex relationships between causes (inputs) and effects (outputs). The resulting model would ultimately be used to generate predictions and derive insights from data. Breiman argued there were two distinct approaches to model creation: Data Modelling and Algorithmic Modelling.

Data Modelling relies on human expertise to design and refine models, using tools like statistics to provide the insights that boost its predictive power. If the number of features are limited, this approach results in a transparent interpretative model that can provide useful insights into the relationships between input and output. However, if the number of features is too large, data models become over-complicated: the notorious Curse of Dimensionality. This makes this approach a poor choice for audio data, since sounds are provided as raw data consisting of a large number of unnamed features.

Algorithmic Modelling, on the other hand, involves letting the data speak for itself, so a model is built by trial and error rather than being handcrafted by experts. This approach replaces intuitive human expertise with an mathematical iterative process like a neural network, which provides a means of automatically evaluating the quality of the model as it evolves.

With a few exceptions (like decision trees), the limitation of the algorithmic approach, particularly when it comes to neural networks, is the model is a “black box”, where the relationships between input and output become highly complex and difficult to understand. But as we no longer need to intuitively understand the data, high dimensionality comes to be considered as a source of value rather than a curse, making this approach well-suited to tasks like image and audio processing.

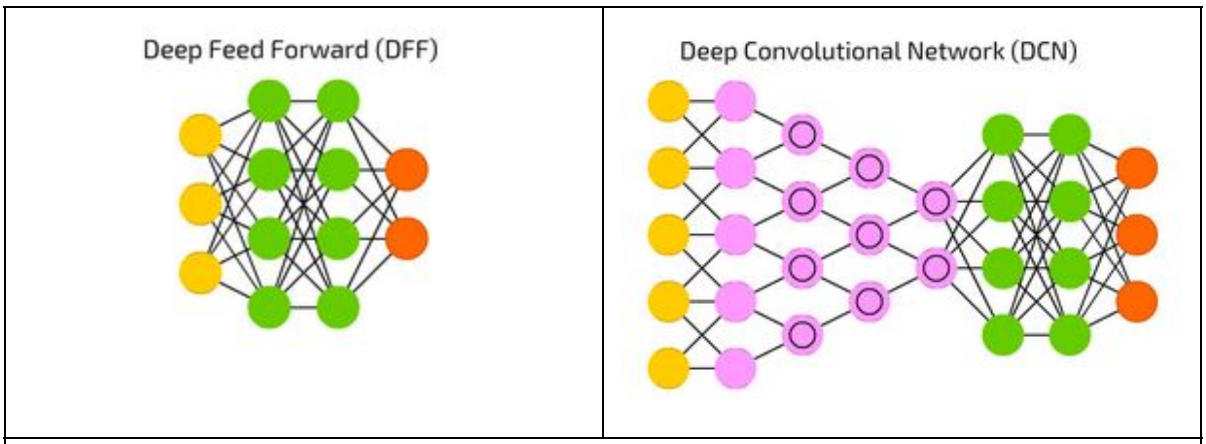
Consequently, for this project, it was clear the best choice was to build the classification model algorithmically, using a deep neural network (one with multiple stacked layers) that would be capable of automatic feature learning.

As this chart neatly demonstrates, there are many different neural network architectures. Rather than just choosing one, I was curious to compare how well two different types of neural network were able to classify sounds. The first neural network I investigated was the Feed-Forward Network (FFN), a simple deep neural net with a few hidden layers.

As their name suggests, FFNs feed information from the front input nodes to back output nodes, via several layers of hidden nodes. FFNs are trained through the process of back-propagation, where the input data generates some output, which then flows backwards through the network, using the chain rule to produce gradients that represent the current error - the difference between what we would expect the network to produce for the current input and what we are currently seeing. This error is used to fine tune the weights between the nodes of the network, which has the effect of progressively reducing the observed error as the output becomes more accurate.

FFNs have the advantage of being relatively simple to implement, but they do not have the property of translation invariance, i.e. the ordinal position of each feature matters, FFNs are not structured to look for patterns in 2-dimensional input data, regardless of where they occur. In theory this should make them fast to train, but may make them less proficient when it comes to their classification accuracy.

So I also intend to try a more sophisticated architecture, the Convolutional Neural Network (CNN). A CNN consists of convolutional layers that attempt to identify patterns present in a two-dimensional array of input data, independent of their position. CNNs have proved very effective in image classification tasks, and have been reported as performing well on audio classification tasks too. This shouldn't be surprising, just as it doesn't matter where the eyes are in photograph, it shouldn't matter where the idiosyncratic sounds that characterise a particular audio source occur in a recording.



The architecture of Feed Forward and Convolutional Neural Networks; showing input nodes (yellow), hidden nodes (green), output nodes (orange). Convolutional nodes are pink, (circles represent pooling).

When implementing neural networks it is common practice to employ a deep learning framework that provides implementations of the common algorithms and conceptual models. For this project I intend to use Keras with a Tensorflow back-end. The attraction of using Keras is that it provides higher-level abstractions for creating neural networks, hiding some of the configuration complexity and simplifying program code by reducing the need for repetitive boilerplate statements.

As later sections will explain, once the neural network was implemented, it was trained and evaluated by cross-validation, supplying a training set, and then testing its performance on held-back examples that have never been encountered before.

2.4 Benchmarks

Several papers have been published reporting classification performance achieved on the UrbanSound8K data set. One paper investigated using traditional machine learning classifiers such as Support Vector Machines and Decision Trees, and reported F1 accuracies of between 50% and 70%, with some classes of sound source being more difficult to classify than others.

In a follow-up paper the same authors discuss the results they've achieved through the use of deep learning, reporting a peak F1 accuracy of 73% when using a convolutional neural network. A paper by Piczak reported his own best accuracy using a slightly different CNN implementation, reporting an

accuracy of 74%. To put these scores into some context, Piczak also reports that in tests, untrained human listeners have scored an average accuracy of 81% on the 2000 sample ESC-50 dataset. In other words, human listeners are fallible when it comes to recognising sounds, and the deep learning systems are already coming close to matching human performance.

These researchers reported how they were able to boost their classification accuracy by employing data augmentation techniques, where noise and distortions were added to samples to grow the size of the training set. I consider implementing these noise adding techniques beyond the scope of this project, so I would expect the results my implementation achieves to be slightly lower than those reported in these papers.

3 Methodology

This section explains how a general audio classifier was implemented with two different neural network architectures. I'll begin by describing how features were extracted for input into a feed-forward network (FFN), and how that network was implemented and refined.

3.1 Audio Classification using a Feed-Forward Network

This section describes code in two Juypter notebooks, for extracting features and training a FFN.

3.1.1 Data Preprocessing

As explained in the previous Data Exploration section, each second of audio consists of 22050 data values. The convention for neural networks is to have one input node per feature, so a 2 second sample would require 44100 input nodes, and then a similar order of magnitude of hidden nodes - far too many to run on a laptop. What's needed is some means of dimensionality reduction, an insight that will extract just the salient information from audio samples.

After doing some research I found a blog post that seemed to offer a solution, using some of the feature extraction methods provided by the Python librosa library. The idea is to reduce the tens of thousands of data points in each file into a much smaller set of features of fixed number. This would allow the network to compare features, like with like, regardless of the duration of the audio sample. The 5 feature extractions involved were:

- Mel-frequency cepstral coefficients (MFCC) - the coefficients that collectively make up the short-term power spectrum of a sound
- Mel-scaled power spectrogram - the Mel Scale is used to provide greater resolution for more informative (lower) frequencies
- Chromagram of a short-time Fourier transform - projects into bins representing the 12 distinct semitones (or chroma) of the musical octave
- Octave-based spectral contrast - distributions of sound energy over octave frequencies
- Tonnetz - estimates tonal centroids as coordinates in a six-dimensional interval space

The results of these 5 extractions are concatenated to give a consistent feature vector of 193 values for every audio clip processed. In practice, the feature extraction process proved very computationally intensive, so rather than repeat this time-consuming activity each time I restarted my laptop, I made use of numpy's save facility to export the extracted features to files, which could be rapidly reloaded later when needed; (you'll see the code for this in notebooks 1 and 2).

3.1.2 FFN Implementation

I started by creating a feed-forward neural network using the Keras Sequential Model, consisting of an input layer, a densely connected hidden layer, both using relu activation (a ramping function that swaps negative values for zeroes, ensuring weights don't get too big or too small), and an output layer with a softmax activation function. The starting weights of the network nodes were initialised by assigning random values drawn from a normal distribution with a mean of zero.

As this was a multi-class classification problem, I specified the categorical_crossentropy loss function. This is the measure that allows mistakes in the network's output to be evaluated. This is vital to the training process, as it allows the network to learn when its output is decisively wrong - without it, the network would not know when it is making mistakes. As the learning rate is determined by the error in the output, the slope of the loss curve will be much steeper initially than a quadratic cost function. This is beneficial when the neuron starts out badly wrong, and helps prevent the search from getting stuck in local minima when the neuron needs to learn fastest.

When the model is trained over several successive training epochs, the console output shows how the training set loss decreases, whilst the accuracy increases - indicating that the model is learning. But we don't want the model to simply memorise the training data and not have the capability to generalise to examples it hasn't seen before. So I added a stopping function that would evaluate a separate measure of loss against an unseen validation set, which allows training to be halted at the point when the model begins to overfit and lose its ability to generalise.

Another tactic I employed to prevent overfitting was to include a dropout policy between the final two hidden layers. I used a value of 0.5, meaning there'd be a 50% chance that any neuron's activation output will be ignored and not propagated to its downstream connections, (and so it would not have its weights updated on the backward pass). The idea is that this random throwing away of information helps prevent the network from learning simple spurious dependencies, and promotes the creation of complex co-adaptations between neurons of the hidden layers. By encouraging multiple neurons to become involved in learning, they can learn on behalf of 'missing' neurons, resulting in the creation of multiple independent internal representations by different groupings of neurons across the network.

Another implementation decision was the choice of optimiser, I began by using stochastic gradient descent (SGD), but soon found better results using the Adam and Adamax algorithms - the claimed advantage of these optimisers is that they are adaptive estimators, and so don't require manual tuning of their hyper-parameters.

3.1.3 FNN Refinement

The performance of a deep neural network is highly dependent on many configurable hyper-parameters that govern how the model actually works, and which are not changed during the course of learning. Some of these parameters will have a greater influence than others, so the classic machine learning

approach to this challenge is GridSearch, which involves successively training separate instances of a model, changing only one key parameter each time, then comparing the resulting accuracy scores to identify which parameters have produced the best results. For this I used the Keras scikit-learn interface, which enabled me to utilise the GridSearchCV implementation.

The parameters that I varied in my experiments were:

- Activation functions - these govern the numerical transform applied to values going from one layer to the next. I tried 3 different functions, 'relu', 'tanh' and 'linear', and found relu (the rectified linear unit) produced the best results.
- Batch Size - this governs the number of feature rows shown to the network before the backpropagation process updates the weights of the network. I used a range of values between 10 and 60, and found 24 produced the best accuracy.
- Max Epochs - an epoch consists of one full training cycle on the training set, so that every sample in the set is seen. Then training restarts, beginning a new epoch, the idea being to continue until the training errors converges and then stop training before the model begins to overfit. I used values between 1 and 30 epochs, but found the best results were achieved with a high value and an Early Stopping condition (it usually halts training after about 10 epochs).
- Initialisation functions - these define how the initial weights of nodes are randomly chosen, and found the best results were obtained using the Lecun uniform strategy, which assigns from a uniform distribution of mean of zero and a standard distribution based on the number of inputs feeding into each node.
- Dropout rate - this is a form of regularisation that aims to prevent the network becoming too committed to its input data, which can cause overfitting. I tried a range of values between 0.1 and 0.9, and found 0.5 produced the best results.
- Optimiser algorithm - gradient descent is highly computationally intensive process, so optimisers are used to expedite it. I tried training with several algorithms: classic SGD, RMSprop, Adagrad, Adadelta, Adam, Adamax and Nadam. I found the two Adaptive Moment implementations: Adam and Adamax, produced the best results.

Using adaptive moment optimisers meant not having to choose a learning rate, a hyper-parameter with important consequences. A learning rate that's too small converges painfully slowly, but one too large can hinder convergence, as the loss function fluctuates around the minimum (or even diverges). Adaptive optimisers vary the learning rate, reducing taking smaller steps when converging, and larger steps whilst searching.

Another refinement I explored was to add another hidden layer to the model. When using the Adam optimiser and a batch size of 32, I found a 1 layer FFN produced an F-score of 0.51. Adding a second layer increased the F-score to 0.58, and a third layer increased the F-score to further 0.6. Adding dropout between these layers increased this still further to 0.62. But adding a fourth layer proved counterproductive, and actually reduced accuracy to 0.55. Consequently, I settled on using 3 layers.

3.2 Audio Classification using a Convolutional Network

This section describes the code in notebook 3 (simple CNN) and notebook 4 (Salamon & Bello CNN)

3.2.1 Data Preprocessing

Section 3.1.1 explained how audio features were extracted and aggregated using processing methods provided by the librosa library, creating an array of values that could be fed into a Feed-Forward Network. This section is going to explain a different approach, how to preprocess audio data so it is in a form that can be fed into a Convolutional Neural Network (CNN).

A CNN organises hidden units to take advantage of the local structure present in two-dimensional input data (the classic example being edges in images). By concentrating on identifying local features each hidden unit only needs to process a tiny part of the whole input space, instead of being connected to all the inputs coming from the previous layer. Processing proceeds by successively considering small windows of the data set, (e.g. 3x3 pixels), called the receptive field.

The weights of hidden units create a convolutional kernel (or filter) that is applied to the whole input space, like a succession of tiles, resulting in a feature map. As a result, one set of learned weights can be reused for the whole input space. As a feature of interest can occur anywhere in the input space, the CNN approach makes it unnecessary for the model to learn where features occur in input data, greatly reducing the number of parameters required, as well as improving its robustness.

A typical convolutional layer will consist of numerous filters (feature maps). Further dimensionality reduction can be achieved through pooling layers, which merge adjacent cells of a feature map, using pooling operations such as max (winner takes all) or the mean of the input cells. This downsampling further improves the tolerance of the network to variation and noise.

CNNs have proved especially successful at classification tasks, particularly of images. To classify audio files this way, a means of extracting audio features is required that creates the kind of input data a CNN expects. A method is described by Piczak in his paper Environmental sound classification with convolutional neural networks, which describes how to get equal size segments from varying length audio clips, and how audio features can be fed into the network as separate channels (akin to a colour image's RGB channels).

I've used an implementation based on sample code posted by Aaqib Saeed, which calculates log-scaled mel-spectrograms and their corresponding deltas from a sound clip. Because we need fixed size input, each sound clip is split into 41 overlapping frames, one for each of the 60 mel-bands, giving an array of 60 rows and 41 columns. The mel-spectrogram for each band/segment and its time-series deltas will become two channels, which becomes our input to feed into the CNN. Other features could be calculated in the same way and supplied as a separate input channel, but we'll stick with just the mel-spectrogram data for now.

If the sound clip is longer than the expected window size, the extraction process will create several features rather than just one. As the sound recordings are often repetitive, this is an example of Data Augmentation, creating several training examples from a single source recording. In the notebook you'll see an example where one sample audio file results in the creation of 7 feature rows, each consisting of 3-dimensional array (60x41x2) of data values. This is an important point, not all feature rows contain

equal amounts of salient information, some might be mostly silence or unrecognisable noise, and may not be able to contribute much to training the network.

At 60x41x2, the feature representation we'll feed into a CNN is clearly not as compact as what was fed into a Feed-Forward Network, which had only 193 features. But at least the 88200 data points have been reduced to 34440. Whilst computation will inevitably be more intensive, the hope is by retaining more of the source data, the CNN will be able to learn more subtle patterns from it.

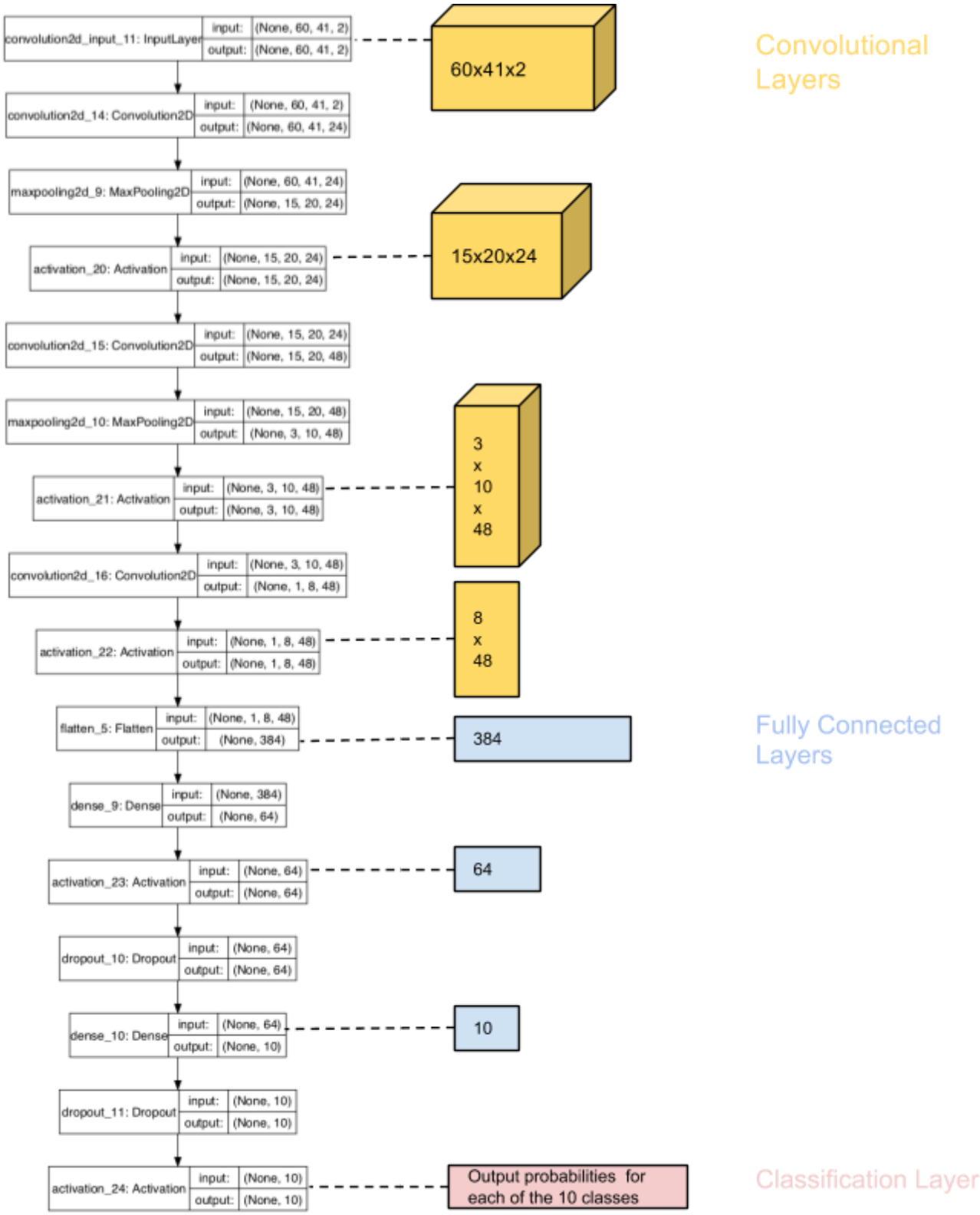
3.2.2 CNN Implementation

As Section 2.3 explained, CNNs differ from FFNs by having dedicated convolutional layers between the input layer and the hidden layers. The model I implemented in notebook 3 begins with two Convolution2D layers, the first with 48 kernels, the second with 96, following the CNN convention of having more of the later layers as they process combinations of the priors. The convolution kernels will progressively squeeze the spatial dimensions while increasing the depth - in effect, transforming space into depth, removing spatial information until only semantic complexity remains.

Once a deep and narrow representation has been created, the output is fed into a regular deep neural network of densely connected layers, and used to train a classifier just as we did with the Feed-Forward Network described previously.

As well as familiar techniques like Dropout and Activation Functions, something new is present in this model - a process called Max Pooling. This works by looking at a small neighbourhood around every point in the future map, and computing the maximum of all the responses around it. Combining many weaker signals into one stronger signal helps reduce the risk of overfitting.

In notebook 4, a more sophisticated CNN is implemented, using the architecture described in a paper by Salamon and Bello, which is shown in the following diagram.



This diagram shows the layers of the CNN, and the flow of the data through them. The initial input is a 4-dimensional, n instances of 60x41x2 feature arrays. As the data flows through the convulsion layers and the output is pooled, its dimensionality is reduced, shrinking the amount of data passed onto the subsequent layer. Once the feature maps have been extracted by the convolutional filters, they're flattened into a 384 element array, before being passed to the two densely connected layers that will be trained to classify their input into one of the 10 final possible outcome classes.

3.2.3 CNN Refinement

All the parameters that were varied and evaluated using GridSearch for the previous FFN implementation can also be fine-tuned for a CNN. In addition, the CNN introduces several more parameters whose values can be optimised by experiment:

- Number of Convolution Kernels - this defines the number of feature detectors (also called filters) in each convolutional layer. Each filter generates a feature map, and the accepted wisdom is to have a smaller number of filters to extract simpler features, then increase the number as more complex agglomerations begin to emerge. The number of kernels tends to be found by experiment; I found the best results with two layers of 48, then two layers of 96.
- Filter Size - this defines the dimensions of each kernel (its number of rows and columns), and thus the neighbourhood of data considered when determining features. This is typically a small number such as 3 or 5, meaning the filter considers a patch 3x3 or 5x5 in size. The larger this number, we prioritise larger-scale features over fine detail. It is also possible to use a 1x1 convolution, which will consider individual data values in isolation. After running some experiments, I seemed to get quite good results with a 1x1 filter, which also made the network much quicker to train.
- Max Pooling Size - the value used for pooling determines the amount by which the data from the previous layer is reduced, so a value of 2,2 will downscale the data by half, both vertically and horizontally. By reducing the quantity of data, using pooling can speed up training by concentrating strong signals and excluding weaker ones. Again, the best value can be found by experimentation, with 2,2 producing the best results in my tests.

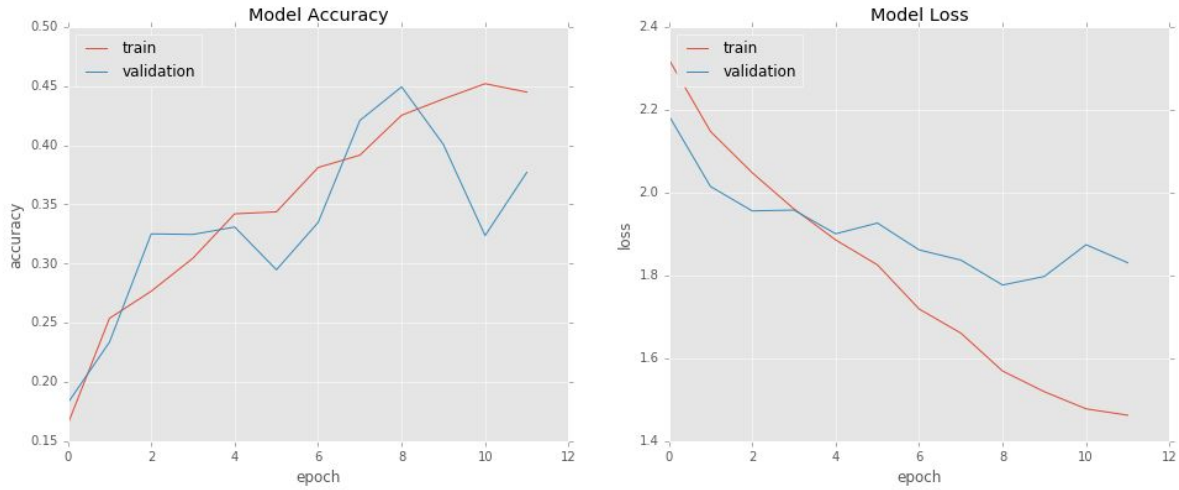
To summarise the story so far, I've successfully implemented a FFN and two different CNNs, and the two different feature extraction processes. Next, we'll consider the results I've achieved.

4 Results

This section reviews the results collected by the best model I implemented, the Salamon & Bello CNN, and considers whether the results reflect a good answers.

4.1 Model Evaluation and Validation

Let's begin by considering two plots generated from the CNN's training history.



The Accuracy plot (above left) provides an insight into whether the model has reached a plateau of performance or could be potentially trained further. This plot shows comparable accuracy on both train and validation datasets - the latter being unseen data that had not been used for training. The results are promising, showing a progressive and proportional increase in accuracy in both sets, indicating the model has the ability to generalise from training data and accurately classify unseen data. Had the two sets diverged significantly it would be an indication that the model had over-learned the training dataset.

The Loss plot (above right) shows comparable performance in both the training and validation datasets. Again, this is a good sign, and indicates the early-stopping function is doing its expected job and halting training when the loss in each set begins to diverge.

To double-check that the model gave reliable predictions, I fed individual sounds into the CNN model and examined the prediction probabilities returned. This is analogous to the process that would be involved if the classifier was used in a real-world application, such as an app. We'd capture sound from the microphone, extract its features, feed it into the model and evaluate the output received.

Below are the prediction probabilities for 10 different recordings, chosen at random so I had sample instances of each of the 10 classes. From the results we can see that the trained CNN has a 91% confidence a sample horn sound is a horn - but much less confidence when identifying the sound of children playing or drilling.

	aircon	horn	children	dog	drill	engine	gun	hammer	siren	music
Probability	**0.914**	**0.873**	**0.306**	**0.652**	**0.595**	**0.876**	**0.621**	**0.794**	**0.97**	**0.729**

Remember that the sounds being classified are short snippets, and it is not easy for a human ear to consistently identify a sound, especially one robbed of its context. Even to a human ear, some of the sample sounds appear to be just white noise.

This is important when it comes to interpreting the accuracy achieved by the system. Machine learning systems never achieve 100% accuracy, not even a panel of human experts would perform that well. For the UrbanSound8k data, published papers have reported F-score accuracies in the range of 0.5 to 0.7, with Piczak's CNN achieving 0.74 being the best performance I've seen.

I've run numerous experiments using the same architecture and parameters, but varying the amount of input data it is trained with. With 5000 features, a typical F-score result is 0.3, with 20000 the F-score is about 0.5, and with 35000 features we'd expect an F-score closer to 0.6. This seems to indicate that the model is indeed a robust classifier, whose performance is predictable rather than erratic, i.e. if we supply more data, it will become more accurate (up to a point).

4.2 Justification

I managed to get slightly better results with the CNN than my best FFN, an ROC of 0.91 with the CNN, compared to 0.89 with my FFN. My top F-score accuracy for both was about 60%.

Getting higher results proved difficult, and there may be diminishing returns for effort given the constraints of the data set. After all, 8000 samples isn't a huge amount, and it does include some repetition of same sound source.

In their paper, Salamon and Bello report an accuracy of 73% - but it's worth bearing in mind that their CNN is optimised for a different feature extraction process. I was feeding my implementation of their CNN features extracted with Piczak's method. The published results also used data augmentation to add noise and variation to boost the size of the same set. It seems together these factors are worth around a 10% improvement in accuracy. Perhaps in the future I'll investigate replicating the CNN that Piczak used, and see how it compares.

But having demonstrated the ability of both the Feed-Forward and Convolutional Neural Network to accurately classify audio data from the UrbanSound8K dataset, I also wanted to move on to investigate how both FFNs and CNNs perform when used to identify birdsong.

The first challenge is to find a high quality labelled set of recordings of birds. Unfortunately it seems difficult to find an equivalent of the UrbanSound8k data for birds, i.e. one where the recordings are labelled with the bird species, but I did find some field recording data sets with binary labels. These labels are indicate that a human listener has stated that either they believe a bird is present, or that no birds are present. No species information is included, so the goal here is bird detection rather than identification, i.e. to find segments of 24-hour automated field recordings that might have interesting content, which can be forwarded to human experts capable of identifying what might be present.

The code for bird detection is in this notebook - although the data sets were too large to check into the git repository, links to the data are included, and it can be downloaded separately.

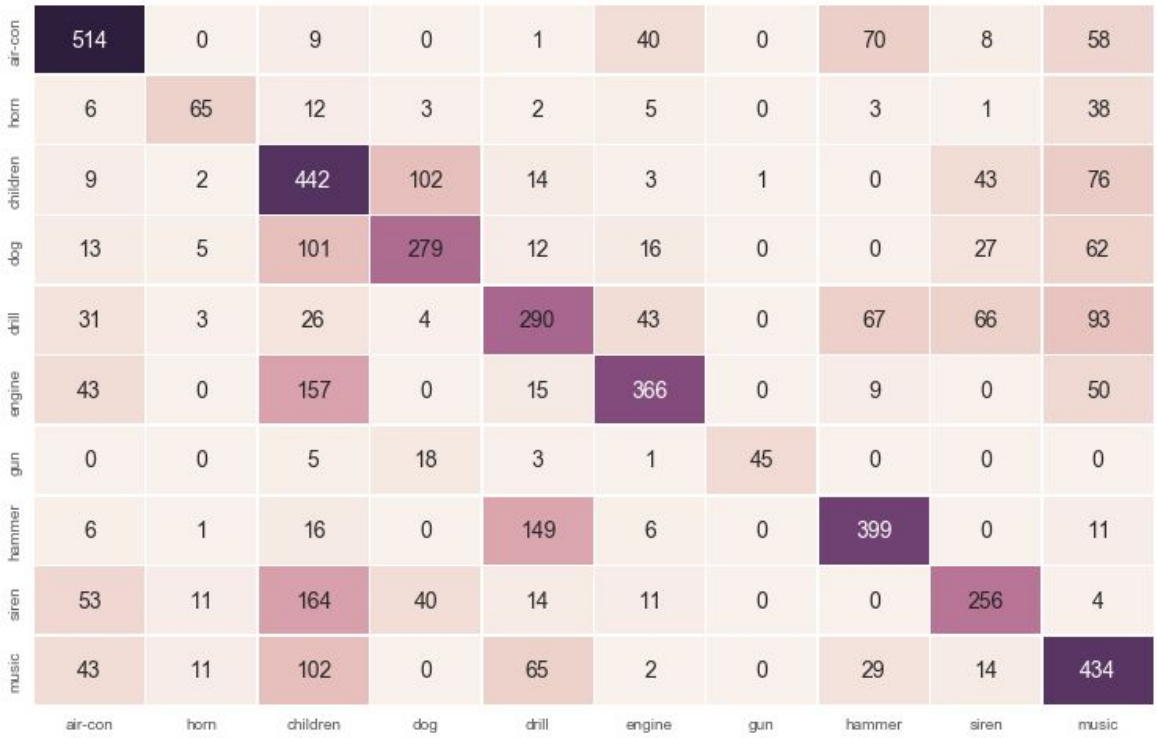
Training on a smaller dataset, using 2 folds of the FreeField1010 data set, I achieved an F-score accuracy of 0.77. Using optimised hyper-parameters (Adagrad optimiser, batch size of 128, patience of 2), the F-score accuracy rose to 0.79.

However, even better results were obtained by combining two sets of field recordings, the Warblr data set and the FreeField1010 data set, which achieved an F-score accuracy of 0.87. This task was however well beyond the computational abilities of a laptop, and took considerable processing on a powerful high-memory cloud computing instance. The curators of the bird detection data will be opening up a challenge soon so researchers will be able to evaluate the performance of their models against each other, I look forward to participating in this.

5 Conclusion

5.1 Free-Form Visualization

The audio classification papers I've seen tend to use a Confusion Matrix to depict the reported results, as this not only demonstrates what is identified correctly, but also the model's limitations, (what it still misclassifies). Below is the confusion matrix for a classifier trained using the Salamon & Bello CNN, on a large data set (~40000 features). The test set is unseen, and consists of 5218 features.



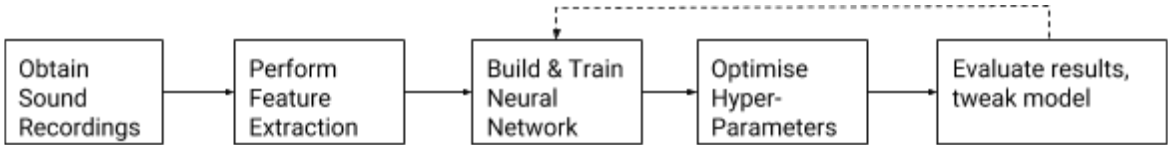
The horizontal axis shows the predicted classes, and the vertical axis the actual classes. The numbers in each cell refer to the number of times the class was predicted to be X, and was actually Y, so the true positives are shown along the diagonal. Happily, the true positives along the diagonal have the darkest shades, indicating the model is performing well at correctly identifying the actual type of a sound.

Just as interesting are the darker shaded cells not on the diagonal - these are the classes that the model tends to confuse most often. These cells provide a valuable insight into the limitations of the model, for instance, why is the sound of children playing being misidentified as a siren 164 times? What it is about their shrieks that the classifier is having difficulty with? These cells show where the model accuracy might be improved, but also presents a new challenge: how can we recognise new salient information without it inappropriately dominating, and creating more false positives?

From my investigations, the sounds the classifier tends to have most difficulty recognising are very short recordings, perhaps only 1 or 2 seconds long, and those that consist of only rapid repetitive noise. As seems to be way with deep learning, the more data the better; if we want to achieve better accuracy, the key is obtaining a diverse, high-quality data set.

5.2 Reflection

The end-to-end problem solution in this project looks like this, the first stage of which was locating appropriate data sets, which were labelled collections of sound recordings.



Next, the sounds needed to be turned into numeric data that reflected their salient features. Here I investigated two possible approaches, one based on spectrogram and harmonic metrics, which produced small feature sets well-suited for feed-forward deep networks. The second approach was based on periodic changes in mel-spectrograms, which was analogous to the changes between successive pixels in image processing. This second approach generated larger feature sets, but enabled the learning by a Convolutional Neural Network which, although more computationally intensive, did achieve the most accurate results.

The next challenge was implementing the neural networks. I originally started by using Tensorflow, but found it rather verbose, its complexity tending to introduce bugs that were frustrating to solve. I seemed to me it was missing the kind of conceptual abstractions that made scikit-learn so easy to use. So I looked for a framework that allowed Tensorflow to be used, but hid its complexities. I considered both Keras and TFLearn, but I preferred Keras on account of its superior documentation, quality of examples and larger user community.

Once built, the models could be trained. With large data sets, even on a fast laptop, this proved a very time-consuming process - especially during the phase when I was using GridSearch to repeatedly run the experiments in order to tune the various hyper-parameters. I was able to create a cloud-based environment to run my larger experiments, but used that sparingly lest this become a very expensive project!

Once a cycle of experiments was complete, I reviewed the results achieved and looked to see what aspects of the model might be tweaked, such as the number of nodes at each layer, or drop-out to govern how much information flows from layer to layer. I'm currently investigating software (Keras.js) to visualise the run-time behaviour of my models, to gain better insights into what actually happens inside the black box at run-time, and thus how greater accuracy might be achieved, but that's a work in progress, and this report is quite long already.

Looking back, there were several challenging aspects to this project. The first was teaching myself the subject of deep learning itself, starting with just four brief Udacity videos on the basic theory, and then learning how to use the Tensorflow framework to build and train simple neural networks.

When I was sufficiently confident with building networks, I began considering what problem I'd like to try solving. My ultimate goal remains birdsong classification, but I needed to walk before I could run, and the UrbanSound8k data provided a means to test and explore different approaches to audio classification, as well as published results I could compare my own model's performance to.

From this experience, I can now see that the chief challenge in producing a birdsong identifier will be accumulating a large, diverse set of bird recordings. I've already seen how dependent deep learning

networks are on having sufficient number of observations for training. If this quantity is too small, the model will not be able to generalise, and will just overfit to the training data provided.

Other challenges were the complexity of extracting features from raw audio data, which required some signal processing expertise that I had to find online in blog posts. Then there was the practical difficulty of the sheer processing power required for both feature extraction and model training.

The opacity of the data was also a challenge, this was the first machine learning project I'd attempted where the data couldn't be viewed as a spreadsheet and intuitively understood beforehand. Instead, I had to rely completely on the metrics reported by my model to determine if the features I'd extracted had any value at all, and that the model was effectively learning and producing sensible predictions.

Nevertheless, it was fascinating to work on a system capable of recognising such complex real-world multimedia data. As a software engineer used to programmatically solving problems, it was an intriguing experience to build a system and then train its capabilities rather than encode them myself.

And ultimately, I'm very pleased with the results I've achieved so far. I've created a general audio classifier, able to distinguish between 10 different ambient sounds, and even determine if it's listening to birdsong. If I was to build an app that could recognise sounds (like birds) from the user's surroundings, I'd definitely continue with the approach I've described in this report.

5.3 Improvement

A direction I'd like to investigate is the use of a Recurrent Neural Network (RNN) for audio classification. Whereas a CNN uses shared parameters across space to extract common patterns, an RNN finds patterns over time rather than space.

An RNN might be well-suited for audio classification. In both the FNN and CNN implementations described in this report, the time sequence of events in a recording is ignored - instead, audio files are snipped up into short segments and learned individually. Then when we attempt to classify a sound, it too is divided up, and each segment classified separately, with each separate prediction combined to produce one final composite prediction.

This approach does seem rather crude though, as it doesn't take into account the context of a sound. It might well be that the meaning of a sound comes from a sequence of events, and we want to make a decision about what's happened so far. In birdsong, for instance, a bird that goes chirp-chirp-cheep may be a completely different species from one whose call is cheep-chirp-chirp.

To attempt audio classification with an RNN, I'd have to experiment with the kind of input data. I expect I'd segment the audio recording into periods of a fixed duration, and then either use the spectrogram metrics (as I did for the FFN), or the harmonic deltas (as I did for the CNN). In either case, the extracted features would be considered in time sequence, analogous to words when RNNs are used for text processing.

A key part of building an RNN seems to be providing the network with an effective memory, so I might utilise LSTM units (long short-term memory). LSTMs help solve the so-called Vanishing Gradient Problem - where the RNN only remembers recent events and forgets the more distant past. This is particularly important if accurate classification relies on not just what was heard a second ago, but what happened ten seconds ago.

Source Code

The code for this project is available in Jupyter notebook format at:

https://github.com/jaron/deep-listening

Because of the huge sizes of the raw sound data sets, I haven't included them in the git repository, but you will be able to download them for yourself if you want to try this code with the full datasets. Otherwise I've included some minimised versions of the data that will allow you to get the code running and see sample results, they just won't be as accurate as the top results I've reported with the full data sets.

I hope you find it as interesting as I have...