Auditory DeepDreamer

By

Andrea Spiteri

A Dissertation

Submitted to the Department of Computing

Goldsmiths University

In Partial Fulfilment of the Requirements

For the Creative Computing Degree

May 2022

Acknowledgements

While carrying out this research I had received a great deal of support and assistance.

I would first like to thank my supervisor, Dr. Jamie Forth, whose advice, guidance and support throughout this research has been very much appreciated.

I would like to thank my sister Dr. Valentina Spitieri, who has been an invaluable supporting figure throughout the entire time of my studies. The many hours spent having technically stimulating conversations, as well as the hours spent bingeing television has been a joy.

I would like to thank my parents, who throughout my entire life have provided me with care, tools, and resources to pursue the things that are of interest to me. For this I will forever be grateful.

I would like to thank my friends Daniel, Grady, Ilya, Tea, Vivienne, Rok Pete, Nico, and Patrick who allowed me to use pictures of their lovely pets in this dissertation.

Lastly, I would like to thank the very exceptional cats Chomsky, Mazzy, Mogi Mogi, Theo, and Ellie, for provide countless comedic moments, as well as love and affection.

Table of Contents

1. Introduction

In the past decade, the field of artificial intelligence (AI) has bore witness to countless advancements due to significant increases in processors speed, which have also become increasingly more affordable. This has allowed for a greater number of people to contribute towards the development of the field. New and innovating AI algorithms are being built, researched, and deployed in systems that can be found in numerous consumer products, spanning across various applications such as drug discovery, self-driving cars, financial trading, and image/video processing.

A great deal of AI technologies that focus on the audio domain have also been emerging. These technologies are finding uses in various applications, for example vocal reconstruction, which can be demonstrated in the recently released documentary called "The Andy Warhol Diaries", where an AI system was created to reconstruct Andy Warhol's voice to narrate his personal diary entries (Resemble AI, 2022).

However, despite the rapid development of AI in the audio domain, little research can be found that contributes towards using AI systems to create audio effects units for musical use cases. This project hopes to contribute knowledge towards this particular use case, by exploring novel ways of processing audio using neural networks.

2. Background

2.1 How Neural Networks Learn.

Neural networks (also referred to as models) are composed of chained functions (Goodfellow et al., 2016), which data, that are represented as a vector, are fed through with the intent of mapping that data from one vector space to another. For example, a network could have three functions chained together (Equation 1).

$$f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(x)\right)\right)$$

Equation 1. Three functions chained together.

Each function is a "layer" in the network. In this case $f^{(1)}$ is the 1st layer (the input layer), $f^{(2)}$ is the second layer (the hidden layer), and $f^{(3)}$ is the 3rd layer (the output layer). There can be any number of hidden layers in a network. These networks are typically represented with an acyclic graph (Figure 1).

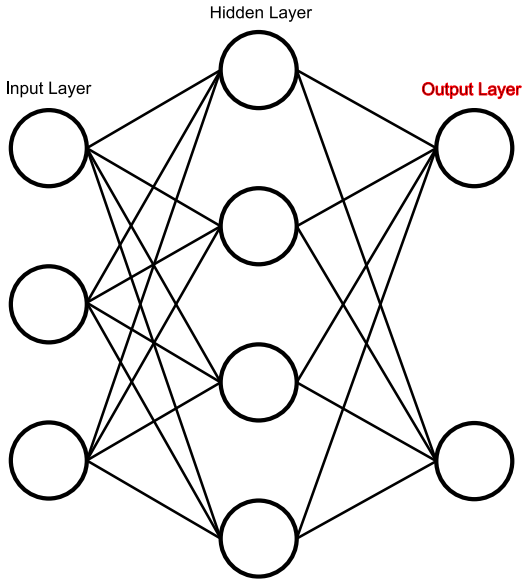
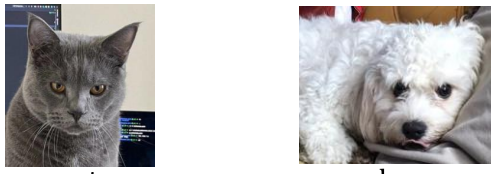


Figure 1. An acyclic graph representing a neural network.

One common use of neural networks is for solving classification problems. This would involve passing input data into the network, and having it classify what that data is. For example, a neural network could be used to classify whether an encoded input image is either a cat or a dog.

Before a network can successfully classify input data to an output, all the parameters of the functions in the network need to be fitted to better approximate a dataset, the process of fitting the data is commonly referred to as "training".

Expanding on the example of a network that classifies images of cats and dogs. To train a network for classification, a dataset with many labelled images is required (Figure 2).



cat dog

Figure 2. Example of labelled images.

Each image in the dataset would need to be processed and encoded into a matrix that can be fed into the input layer of the neural network. In this example, the image is encoded into a 3 × 1 matrix (Figure 3). Images are typically represented by their height, width, and colour depth.

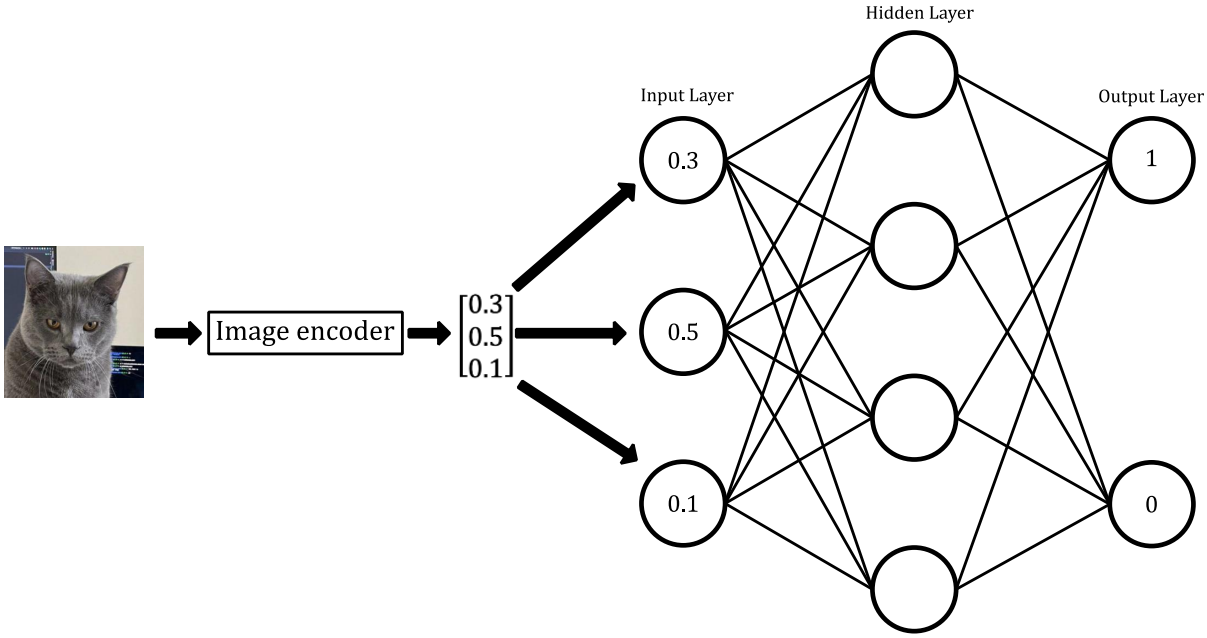


Figure 3. Matrix representation of image fed into the input layer.

The output layer would need to be a matrix representation of the labels in the dataset, these labels are also commonly referred to as classes. A common method for encoding labels is "one-hot-encoding". This involves creating a matrix that is the same size as the number of classes, and representing each label as a 1 at its corresponding index position (Figure 4).

$$\text{classes} = \begin{bmatrix} cat \\ dog \end{bmatrix}$$

$$\text{cat} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \text{dog} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Figure 4. Example of one-hot-encoding.

Expanding on Figure 3, a one-hot encoded representation of the cat image would be $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ (Figure 5).

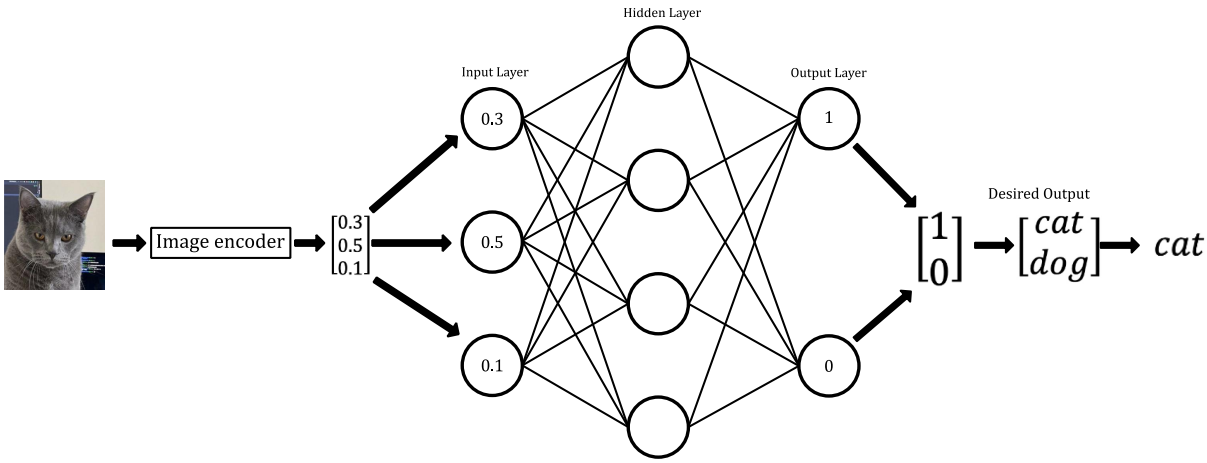


Figure 5. Matrix representation of desired output.

The hidden layer in this example has four nodes, these are also commonly referred to as "neurons" or "units" (Figure 6). Each hidden layer can have any number of units.

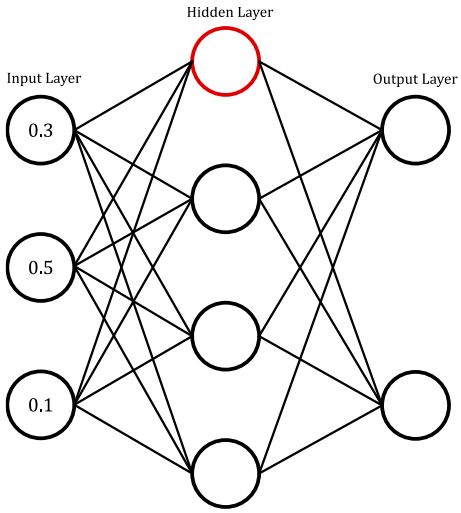


Figure 6. The red node represents a single unit.

Each unit represents two functions; one that transforms the input data (Equation 2) and an "activation" function (Figure 9).

$$f(x) = weights \cdot x + bias$$

Equation 2. Function that transforms the input data.

The two parameters', "weights", and "biase" can be adjusted to better fit the dataset. The network fitting the data is considered the neural network "learning" the data. In two dimensions this function is simply a straight line (Figure 7).

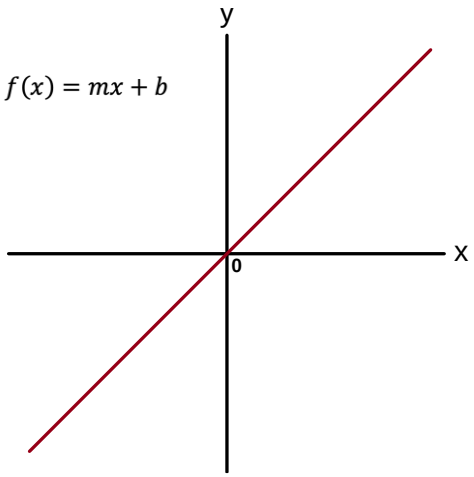


Figure 7. Function for a straight line.

Initially, all the weights and biases in a neural network are set to random values, typically in a range of -1 to 1. Expanding on Figure 6, Figure 8 visually represents the first transformation of the input data for the unit highlighted in red.

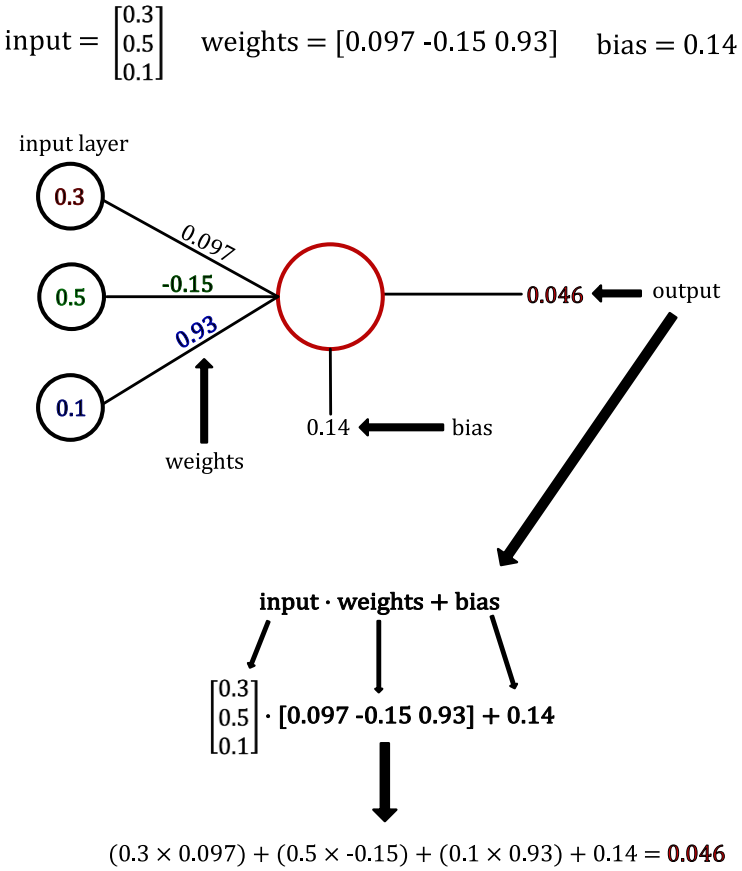


Figure 8. Transformation on the input data for the first unit of the hidden layer with randomly initialized weights and bias.

The output from the first function is used as the input to the activation function. This function is called the activation function because it determines if the neuron "activates", similarly to how neurons in the brain activate. There are numerous functions this could be, but one of the most utilized functions is the rectified linear function (ReLU) (Figure 9).

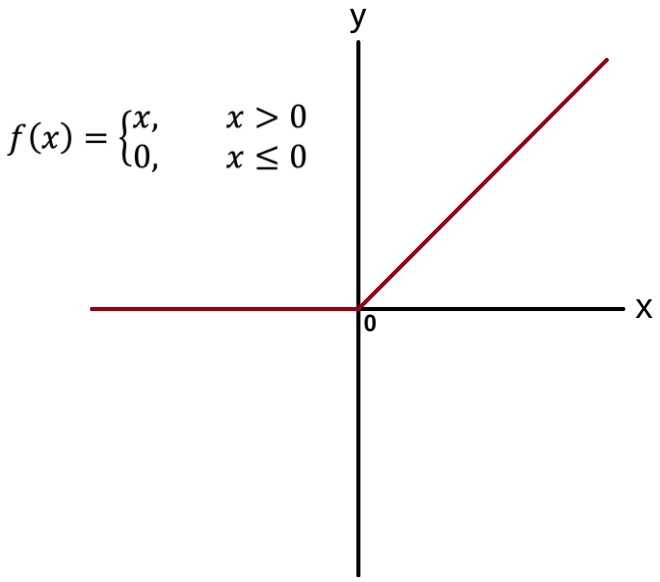


Figure 9. A plot of the rectified linear function.

The output calculated from Figure 8 would be passed on as the input to the ReLU function (Equation 3).

$$ReLU(0.046) = 0.046.$$

Equation 3. Final out of the first unit from the hidden layer.

The final output for this unit in the neural network is 0.046. The same process is repeated for the other units in the hidden layer (Figure 10).

$$\text{input} = \begin{bmatrix} 0.3 \\ 0.5 \\ 0.1 \end{bmatrix} \quad \text{weights} = \begin{bmatrix} 0.1 & 0.8 & 0.4 \\ -0.9 & 0.3 & 0.8 \\ 0.7 & 0.1 & -0.2 \\ -0.9 & 0.3 & 0.7 \end{bmatrix} \quad \text{biases} = \begin{bmatrix} 0.2 \\ 0.2 \\ -0.3 \\ 0.5 \end{bmatrix}$$

$$\mathbf{ReLU}(\mathbf{inputs} \cdot \mathbf{weights} + \mathbf{biases})$$

$$\downarrow$$

$$\left(\begin{bmatrix} 0.3 \\ 0.5 \\ 0.1 \end{bmatrix} \cdot \begin{bmatrix} 0.098 & -0.15 & 0.93 \\ 0.43 & 0.292 & -0.23 \\ 0.21 & -0.12 & 0.58 \\ 0.0898 & 0.78 & 0.058 \end{bmatrix} \right) + \begin{bmatrix} 0.14 \\ 0.85 \\ -0.86 \\ -0.83 \end{bmatrix} = \begin{bmatrix} 0.046 \\ 0.25 \\ 0.058 \\ 0.42 \end{bmatrix}$$

$$\downarrow$$

$$(0.3 \times 0.098) + (0.5 \times -0.15) + (0.1 \times 0.93) + 0.14 = 0.046$$

$$(0.3 \times 0.43) + (0.5 \times 0.292) + -0.23) + 0.85 = 0.25$$

$$(0.3 \times 0.21) + (0.5 \times -0.12) + (0.1 \times -0.58) + -0.86 = 0.058$$

$$(0.3 \times 0.0898) + (0.5 \times 0.78) + (0.1 \times 0.058) + -0.83 = 0.42$$

$$\downarrow$$

$$\mathbf{ReLU}(0.046) = 0.046$$
$$\mathbf{ReLU}(0.25) = 0.25$$
$$\mathbf{ReLU}(0.058) = 0.058$$
$$\mathbf{ReLU}(0.42) = 0.42$$

$$\downarrow$$

$$\text{first layer output} = \begin{bmatrix} 0.046 \\ 0.25 \\ 0.058 \\ 0.42 \end{bmatrix}$$

Figure 10. All calculations for the hidden layer with randomly initialized weights and biases.

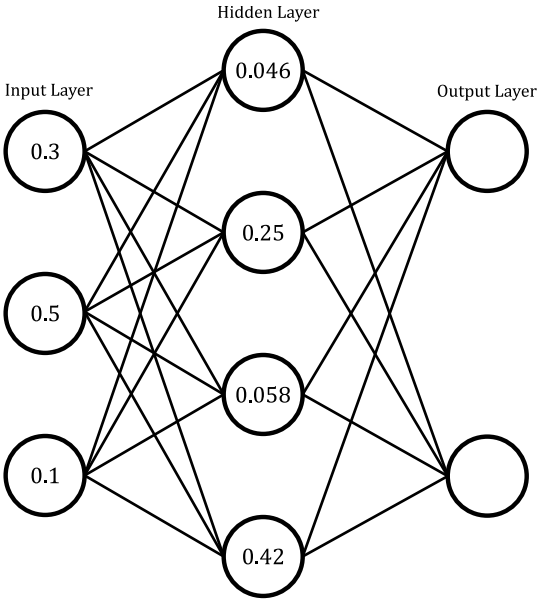


Figure 11. Current state of the neural network after the encoded input data has been passed through the hidden layer.

These hidden layers are also commonly referred to as “dense layer”. The outputs from the hidden layer are next passed to the output later (Figure 12). The output layer utilizes a different activation function called Softmax (Figure 13). The Softmax function is a commonly used function for the output layer of models intended for classification problems because it creates an output vector, that when summed together results in a value of one. This means the final output is a probability distribution. This probability distribution will be interpreted as a “prediction” made by the network.

$$\text{inputs} = \begin{bmatrix} 0.046 \\ 0.25 \\ 0.058 \\ 0.42 \end{bmatrix} \quad \text{weights} = \begin{bmatrix} -0.96 & 0.56 & 0.98 & -0.08 \\ 0.67 & 0.74 & 0.60 & 0.56 \end{bmatrix} \quad \text{biases} = \begin{bmatrix} -0.76 \\ 0.28 \end{bmatrix}$$

$$\text{Softmax}(\text{input} \cdot \text{weights} + \text{bias})$$

$$\downarrow$$

$$\begin{bmatrix} 0.046 \\ 0.25 \\ 0.058 \\ 0.42 \end{bmatrix} \cdot \begin{bmatrix} -0.96 & 0.56 & 0.98 & -0.08 \\ 0.67 & 0.74 & 0.60 & 0.56 \end{bmatrix} + \begin{bmatrix} -0.76 \\ 0.28 \end{bmatrix} = \begin{bmatrix} 0.12 \\ 0.49 \end{bmatrix}$$

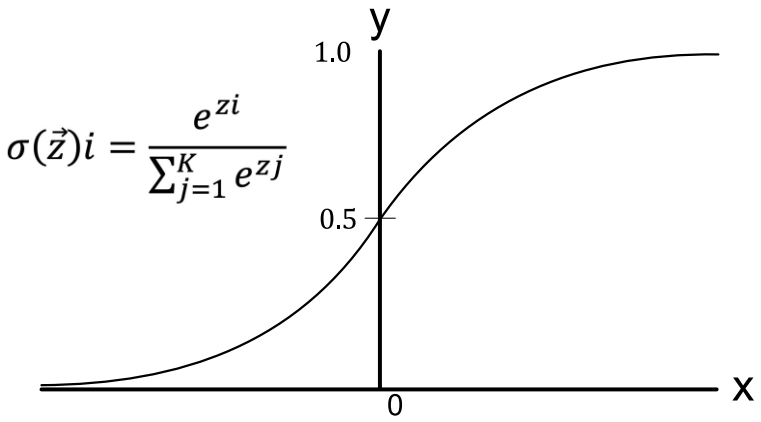
$$\downarrow$$

$$\text{Softmax}\left(\begin{bmatrix} 0.12 \\ 0.49 \end{bmatrix}\right) = \begin{matrix} \frac{e^{0.12}}{e^{0.19=2} + e^{0.49}} = & 0.41 \\ \frac{e^{0.49}}{e^{0.12} + e^{0.49}} = & 0.59 \end{matrix}$$

$$\downarrow$$

$$\text{Final layer output} = \begin{bmatrix} 0.41 \\ 0.59 \end{bmatrix}$$

Figure 12. All calculations for the final layer.



$\vec{z} = Input\ vector.$
$e^{z_i} = exponential\ function\ applied\ to\ each\ element\ of\ the\ input\ vector.\ This\ results\ in\ a\ positive\ number.$
$\sum_{j=1}^{K} e^{z_i} = Normalize\ the\ output\ value.\ The\ elements\ of\ the\ resulting\ output\ will\ sum\ to\ 1.$
$z_i = Elements\ of\ the\ input\ vector.\ This\ can\ be\ any\ real\ number.$
$k = Number\ of\ classes.$

Figure 13. Plot of Softmax function. (Wood, 2019)

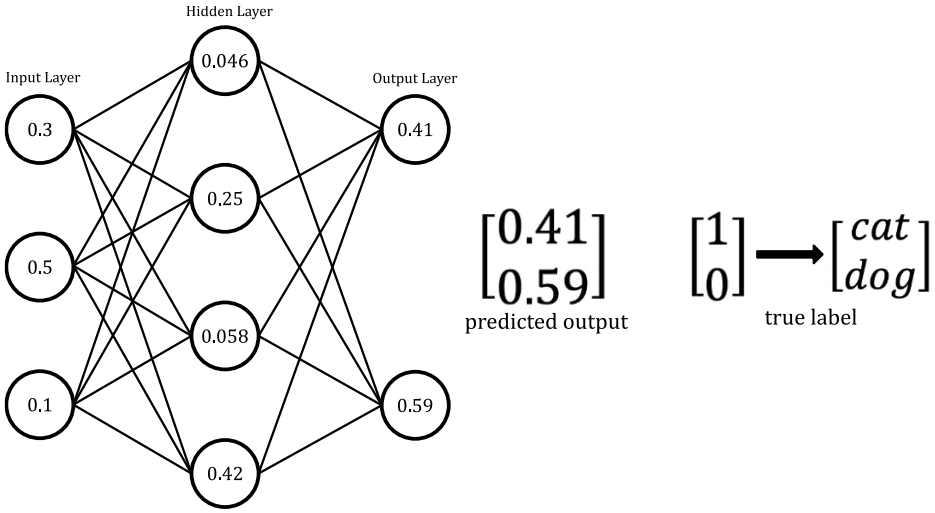


Figure 14. Current state of the network after the output layer has been calculated

The predicted output from Figure 14 can be interpreted as the network "thinking" there is a 41% probability of the input data being an image of a cat, and 59% probability of the input data being an image of a dog. How successful the network is at making predictions can be measured with a function called categorical crossentropy (Figure 15), this metric is commonly called "loss".

$$Loss = -\sum_{i=1}^{k} y_i \log \hat{y}_i$$

$Loss = messure\ of\ error\ of\ the\ model.$
$\hat{y}_i = elements\ of\ the\ output\ matrix.$
$y_i = elements\ of\ the\ target\ matrix$
$k = Number\ of\ classes.$

Figure 15. Equation for categorical crossentropy.

$$\text{Final layer output} = \begin{bmatrix}0.41\\0.59\end{bmatrix} \quad \text{target} = \begin{bmatrix}1\\0\end{bmatrix}$$

$$\text{Loss} = -(1\log 0.41 + 0\log 0.59) = 0.90$$

Figure 16. Current loss of the network. Log function is in base e.

The loss is smaller if the network confidently predicts the data correctly, and is larger if the network confidently predicts the data incorrectly (Figure 17).

Correct Prediction

$$\text{Final layer output} = \begin{bmatrix} 0.97 \\ 0.03 \end{bmatrix} \qquad \text{target} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\text{Loss} = -(1\log 0.97 + 0\log 0.03) = 0.03$$

Incorrect Prediction

$$\text{Final layer output} = \begin{bmatrix} 0.04 \\ 0.96 \end{bmatrix} \qquad \text{target} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\text{Loss} = -(1\log 0.04 + 0\log 0.96) = 3.22$$

Figure 17. Examples of loss when a correct and an incorrect prediction is made.

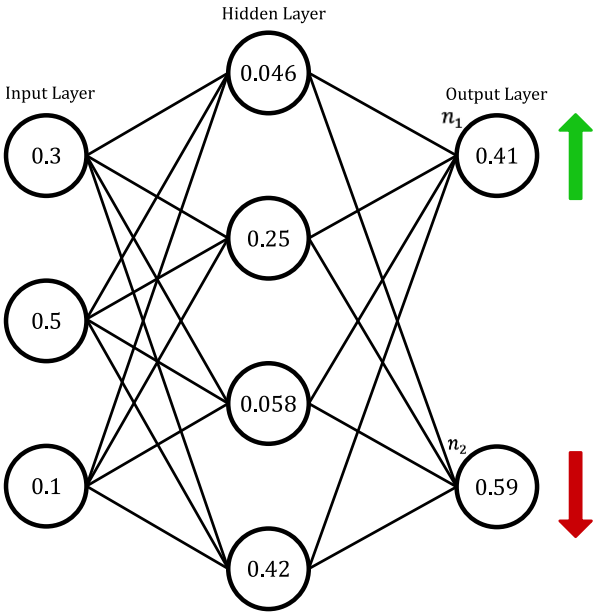


Figure 18. To achieve an improved loss, all the weights and biases of the network need to be adjusted to increase the output of n1, and decrease the output of n2 to achieve a final output that is closer to the target matrix.

The process of adjusting the weights and biases involves calculating the gradient of the loss function with respect to all the weights and biases utilizing the chain rule from calculus. This process is referred to as backpropagation (Nelson, 2020). The first step involves calculating the derivative of the loss, and Softmax functions (Figure 19).

$$\hat{y} - y$$

$\hat{y} = predictions\ from\ model.$
$y = target\ matrix.$

$$\text{predictions} = \begin{bmatrix} 0.41 \\ 0.59 \end{bmatrix} \qquad \text{target matrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 0.41 \\ 0.59 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -0.59 \\ 0.59 \end{bmatrix}$$

Figure 19. Simplified derivative of Softmax function (Kinsley & Kukiela, n.d.).

The derivates for the weights and biases attached to the output layer (Figure 20) can be calculated using the loss derivative.

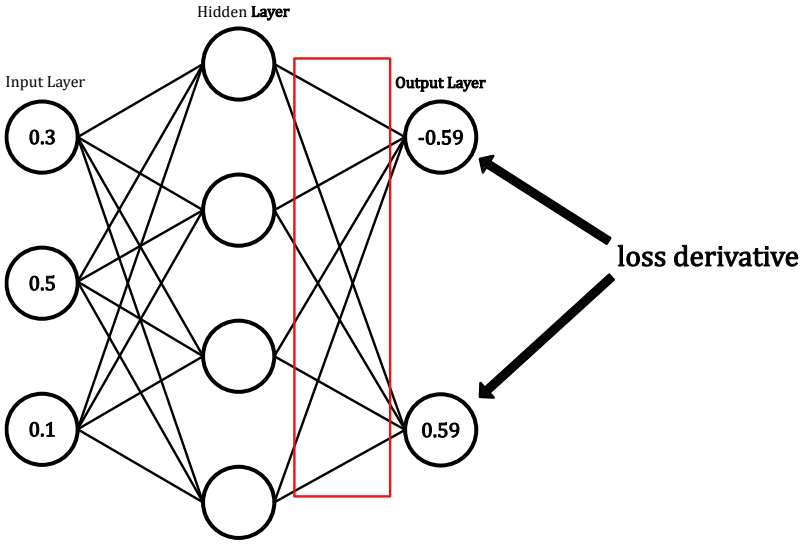


Figure 20. Weights and biases attached to the output layer.

$$\text{loss derivative} = \begin{bmatrix} -0.59 \\ 0.59 \end{bmatrix} \qquad \text{hidden layer output} = \begin{bmatrix} 0.046 \\ 0.25 \\ 0.058 \\ 0.42 \end{bmatrix}$$

$$weight\ derivative = hidden\ layer\ outputs^T \cdot loss\ \ derivative$$

$$\downarrow$$

$$\begin{bmatrix} 0.045 & 0.25 & 0.057 & 0.42 \end{bmatrix} \cdot \begin{bmatrix} -0.59 \\ 0.59 \end{bmatrix} = \begin{bmatrix} -0.027 & -0.15 & -0.034 & -0.25 \\ 0.027 & 0.15 & 0.034 & 0.25 \end{bmatrix}$$

Figure 21. Calculation of the weight derivate.

$$\text{loss derivative} = \begin{bmatrix} -0.59 \\ 0.59 \end{bmatrix}$$

$$\text{bias derivative} = \text{loss derivative}$$

Figure 22. Biases derivative simplifies to equal the loss derivative (Kinsley & Kukiela, n.d.).

The hidden layer's output derivative can be calculated using the original weights of the layer, and the loss derivative (Figure 23).

$$\text{loss derivative} = \begin{bmatrix} -0.59 \\ 0.59 \end{bmatrix} \qquad \text{weights} = \begin{bmatrix} -0.96 & 0.56 & 0.98 & -0.08 \\ 0.67 & 0.74 & 0.60 & 0.56 \end{bmatrix}$$

$$layer\ output\ derivative = \ loss\ derivative^T \cdot layer\ weights$$

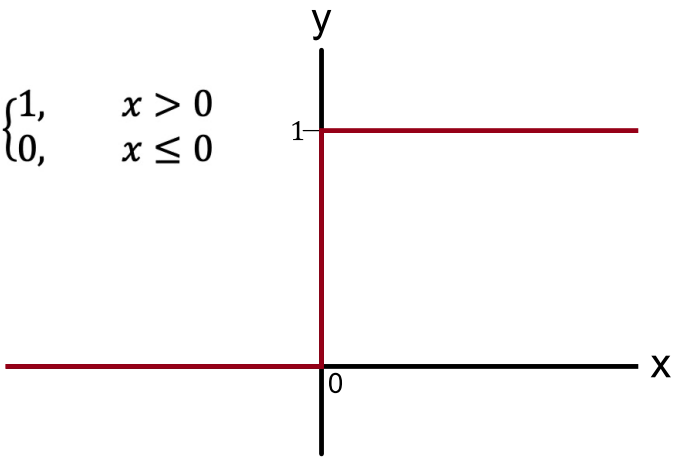
$$\downarrow$$

$$\begin{bmatrix} -0.59 & 0.59 \end{bmatrix} \cdot \begin{bmatrix} -0.96 & 0.56 & 0.98 & -0.08 \\ 0.67 & 0.74 & 0.60 & 0.56 \end{bmatrix} = \begin{bmatrix} 0.96 \\ 0.11 \\ -0.21 \\ 0.38 \end{bmatrix}$$

Figure 23. Calculating the hidden layer output's derivatives.

The hidden layer's activation function (ReLU) derivative can be calculated using the hidden layer's output derivative (Figure 24).

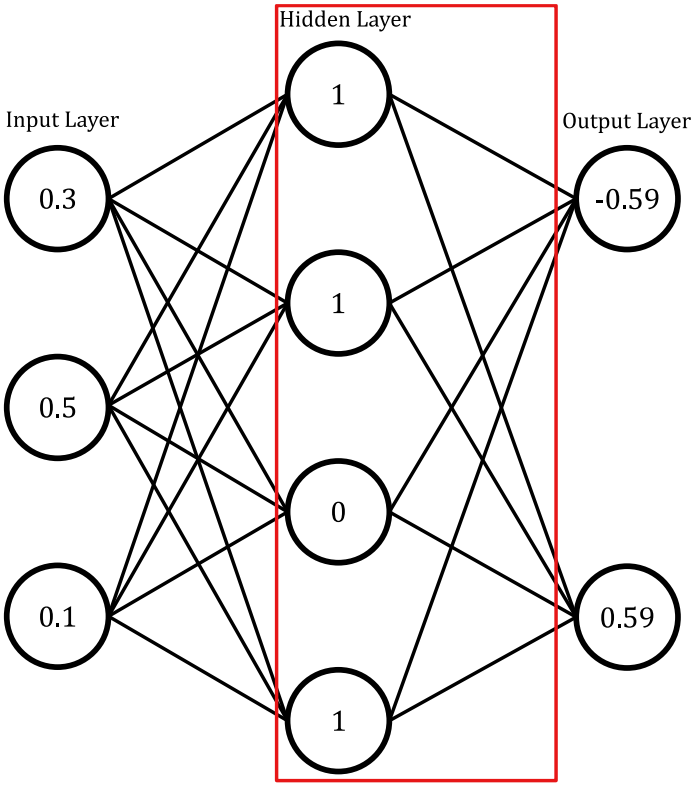
$$f'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$



$$layer\ output\ derivative = \begin{bmatrix} 0.96 \\ 0.11 \\ -0.21 \\ 0.38 \end{bmatrix}$$

$$f'\left(\begin{bmatrix} 0.96 \\ 0.11 \\ -0.21 \\ 0.38 \end{bmatrix}\right) = [1 \quad 1 \quad 0 \quad 1]$$

Figure 24. Calculating the ReLU function derivative from the hidden layer.



Current derivatives calculated.

$$output\ loss\ derivative = \begin{bmatrix} -0.59 \\ 0.59 \end{bmatrix}$$

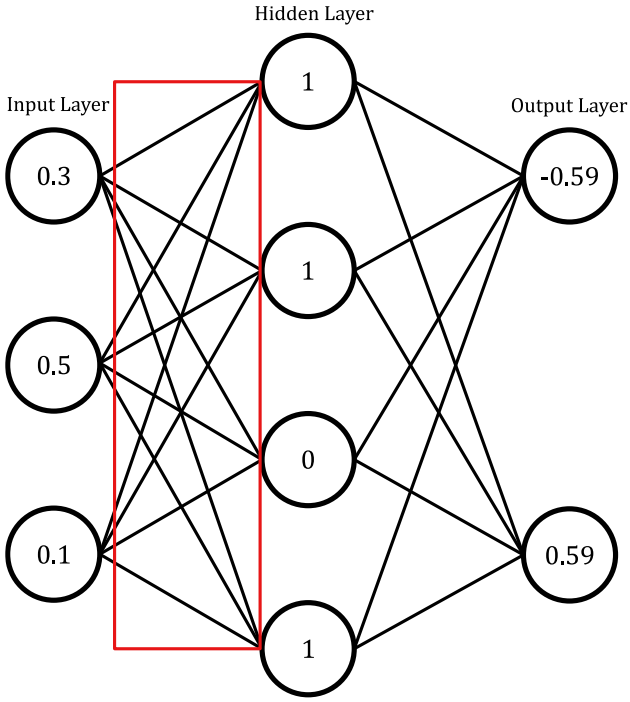
$$weights\ derivative = \begin{bmatrix} -0.027 & -0.15 & -0.034 & -0.25 \\ 0.027 & 0.15 & 0.034 & 0.25 \end{bmatrix}$$

$$bias\ derivative = \begin{bmatrix} -0.59 \\ 0.59 \end{bmatrix}$$

$$hidden\ layer\ output\ derivative = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

Figure 25. Current state of the model after calculating the derivatives of the weights, and biases from the output layer to the hidden layer.

The same process is repeated for the weights, and biases from the hidden layer to the input layer (Figure 26).



Derivatives for weights and biases.

$$bias\ derivative = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

$$weights\ derivative = \begin{bmatrix} 0.3 & 0.5 & 0.1 \\ 0.3 & 0.5 & 0.1 \\ 0 & 0 & 0 \\ 0.3 & 0.5 & 0.1 \end{bmatrix}$$

Figure 26. Calculated derivates for the weights and biases attached to the hidden layer.

The calculated derivatives for all the weights, and biases in the network can now be used to update the initial weights, and biases (Figure 27). This process is referred to as optimization. There are numerous optimization techniques that can be utilized, which are all variations of a method called stochastic gradient descent (SGD). SGD can have several parameters associated with it. However, in this example only one will be utilized, which is the "learning rate". This parameter determines how large a "step" in a particular direction all the weights, and biases matrices should take to minimize the loss. This process of minimizing loss by taking incremental steps towards a vector space associated with the minimal loss is referred to as gradient descent.

$$learning\ rate = 0.8$$

$$updated\ weights = original\ weights + -learning\ rate \times derivative\ weights$$

$$updated\ biases = original\ biases + -learning\ rate \times derivative\ biases$$

Figure 27. Calculations required to update the weights, and biases in the neural network.

$$original\ weights = \begin{bmatrix} 0.098 & -0.15 & 0.93 \\ 0.43 & 0.292 & -0.23 \\ 0.21 & -0.12 & 0.58 \\ 0.0898 & 0.78 & 0.058 \end{bmatrix} \qquad original\ biases = \begin{bmatrix} 0.14 \\ 0.85 \\ -0.86 \\ -0.83 \end{bmatrix}$$

$$weights\ derivative = \begin{bmatrix} 0.3 & 0.5 & 0.1 \\ 0.3 & 0.5 & 0.1 \\ 0 & 0 & 0 \\ 0.3 & 0.5 & 0.1 \end{bmatrix} \qquad bias\ derivative = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

$$updated\ weights = \begin{bmatrix} 0.098 & -0.15 & 0.93 \\ 0.43 & 0.292 & -0.23 \\ 0.21 & -0.12 & 0.58 \\ 0.0898 & 0.78 & 0.058 \end{bmatrix} + \left(-0.8 \times \begin{bmatrix} 0.3 & 0.5 & 0.1 \\ 0.3 & 0.5 & 0.1 \\ 0 & 0 & 0 \\ 0.3 & 0.5 & 0.1 \end{bmatrix} \right) = \begin{bmatrix} -0.14 & -0.55 & 0.85 \\ 0.19 & -0.11 & 0.31 \\ 0.21 & -0.12 & 0.58 \\ -0.15 & 0.38 & -0.02 \end{bmatrix}$$

$$updated\ biases = \begin{bmatrix} 0.14 \\ 0.85 \\ -0.86 \\ -0.86 \end{bmatrix} + \left(-0.8 \times \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} -0.66 \\ 0.05 \\ -0.86 \\ -1.63 \end{bmatrix}$$

Figure 28. Using SGD to update the weights, and biases from the input layer to the hidden layer.

$$original\ weights = \begin{bmatrix} -0.96 & 0.56 & 0.98 & -0.08 \\ 0.67 & 0.74 & 0.60 & 0.56 \end{bmatrix} \quad original\ biases = \begin{bmatrix} -0.76 \\ 0.28 \end{bmatrix}$$

$$weights\ derivative = \begin{bmatrix} -0.027 & -0.15 & -0.034 & -0.25 \\ 0.027 & 0.15 & 0.034 & 0.25 \end{bmatrix} \quad bias\ derivative = \begin{bmatrix} -0.59 \\ 0.59 \end{bmatrix}$$

$$updated\ weights = \begin{bmatrix} -0.96 & 0.56 & 0.98 & -0.08 \\ 0.67 & 0.74 & 0.60 & 0.56 \end{bmatrix} + \left(-0.8 \times \begin{bmatrix} -0.027 & -0.15 & -0.034 & -0.25 \\ 0.027 & 0.15 & 0.034 & 0.25 \end{bmatrix}\right) = \begin{bmatrix} -0.94 & 0.68 & 0.98 & 0.12 \\ 0.64 & 0.62 & 0.57 & 0.36 \end{bmatrix}$$

$$updated\ biases = \begin{bmatrix} -0.76 \\ 0.28 \end{bmatrix} + \left(-0.8 \times \begin{bmatrix} -0.59 \\ 0.59 \end{bmatrix}\right) = \begin{bmatrix} -0.29 \\ -0.19 \end{bmatrix}$$

Figure 29. Using SGD to update the weights, and biases from the hidden layer to the output layer.

With updated weights, and biases, the dataset can be run through the network again to calculate a new loss (Figure 30 and Figure 31).

$$input = \begin{bmatrix} 0.3 \\ 0.5 \\ 0.1 \end{bmatrix} \quad updated\ weights = \begin{bmatrix} -0.14 & -0.55 & 0.85 \\ 0.19 & -0.11 & 0.31 \\ 0.21 & -0.12 & 0.58 \\ -0.15 & 0.38 & -0.02 \end{bmatrix} \quad updated\ biases = \begin{bmatrix} -0.66 \\ 0.05 \\ -0.86 \\ -1.63 \end{bmatrix}$$

$$\mathbf{ReLU}(\mathbf{inputs \cdot weights + biases})$$

$$new\ hidden\ layer\ output = \begin{bmatrix} 0 \\ 0 \\ 0.06 \\ 0.14 \end{bmatrix}$$

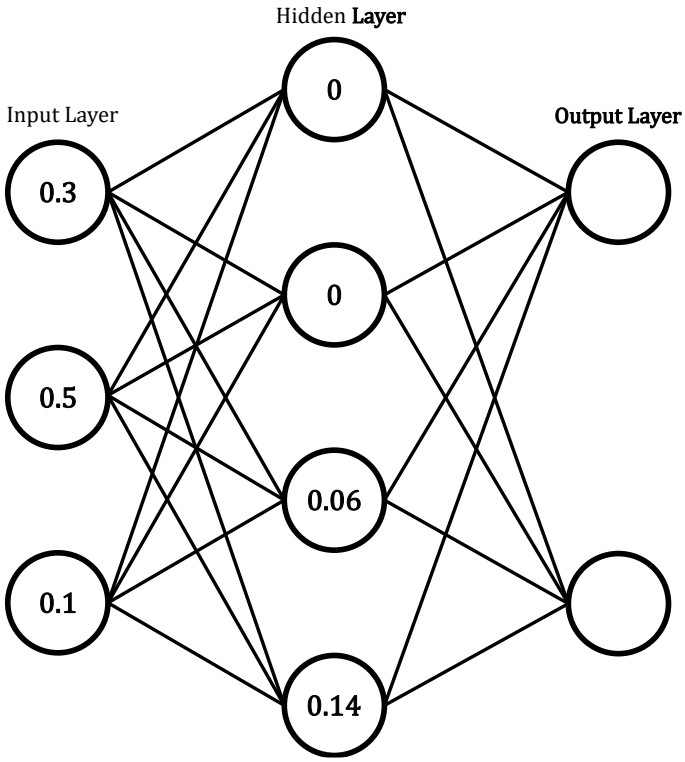


Figure 30. Calculate new output for hidden layer with updated weights and biases.

$$input = \begin{bmatrix} 0 \\ 0 \\ 0.06 \\ 0.14 \end{bmatrix} \quad updated\ weights = \begin{bmatrix} -0.94 & 0.68 & 0.98 & 0.12 \\ 0.64 & 0.62 & 0.57 & 0.36 \end{bmatrix} \quad updated\ biases = \begin{bmatrix} -0.29 \\ -0.19 \end{bmatrix}$$

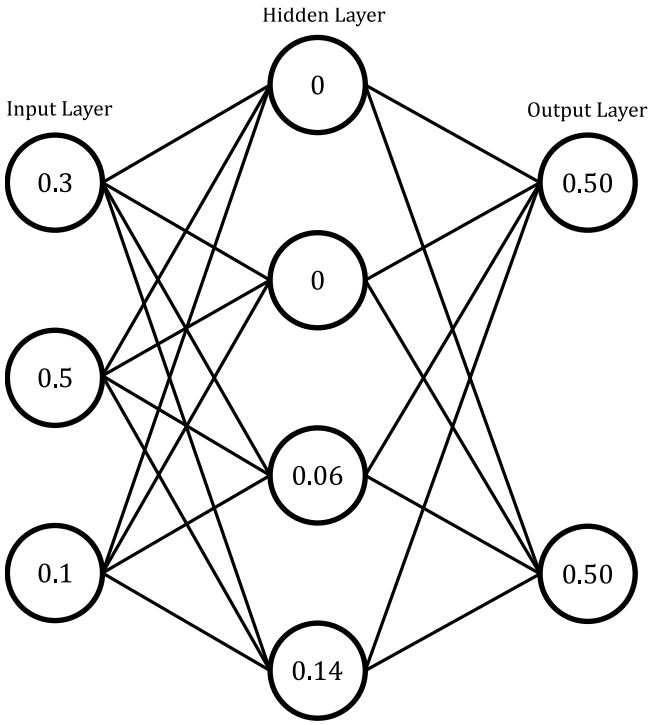
$$\mathbf{Softmax}(\mathbf{input \cdot weights + bias})$$

$$\downarrow$$

$$new\ prediction = \begin{bmatrix} 0.50 \\ 0.50 \end{bmatrix}$$

Figure 31. Calculate new final prediction with updated weights, and biases.

The network is now predicting that the input image has a 50-50% probability of being either a cat or dog. This is an improvement from the initial output, and results in a reduced loss (Figure 32).



$$previous\ prediction = \begin{bmatrix} 0.41 \\ 0.59 \end{bmatrix}$$

$$previous\ loss = 0.90$$

$$new\ prediction = \begin{bmatrix} 0.50 \\ 0.50 \end{bmatrix}$$

$$new\ loss = 0.70$$

Figure 32. Final state of network after the weights, and biases have been adjusted, and a new prediction has been made.

This network now better "fits" the dataset, which for the sake of this example is a single "image" of a cat. This would not be the case normally; a network requires a substantial amount of data for training it to get desirable results. The process that's been

demonstrated so far can be quantified as one "epoch". A network might require numerous epochs before a minimum loss is found.

Once a network is trained, it is evaluated on data it's not been trained with. This is done to ensure that the network did not simply "memorize" how to classify the training dataset, but it instead learnt "generalization" from the data, and therefore will be able to accurately classify previously unseen data. A model that has memorized the training data is referred to as a model that is "over-fitting".

Typically, a portion of the entire dataset is omitted for evaluating the model after training, this is referred to as the "test set". The primary metric of interest for classification problems is the "accuracy" of the model. Expanding on the example of classifying images of cats, and dogs, this would involve letting the model classify all the images in a test set, and comparing the predictions against the "true" labels of the image (Figure 33).

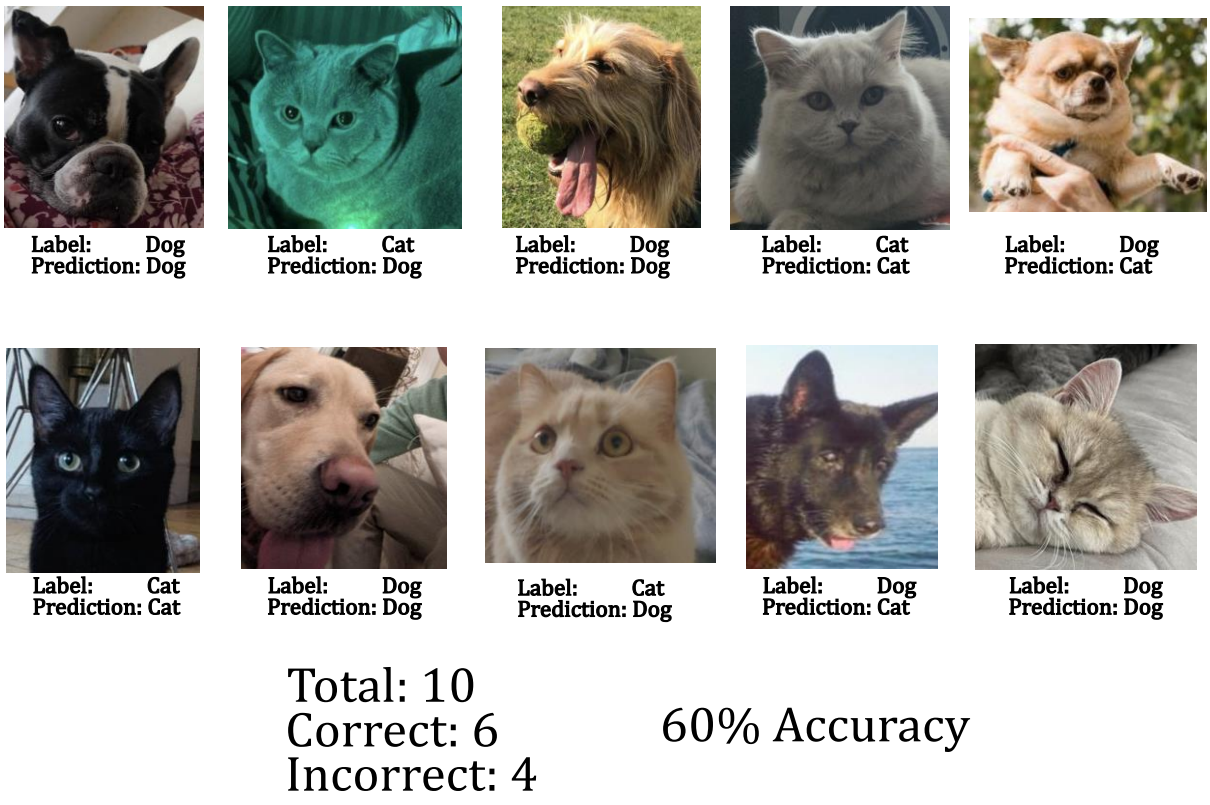


Figure 33. Example of how a classification model is evaluated with a mock test set.

2.2 Convolutional Neural Networks, and Feature Visualization.

The neural network demonstrated so far is one of many different types of neural networks. A more widely used neural network that performs well with image and audio data is the convolutional neural network (CNN). A CNN uses the convolutional operation in place of matrix multiplication (Goodfellow et al., 2016). Convolutional layers have a "kernel" that represents an "area" of the input (Figure 34).

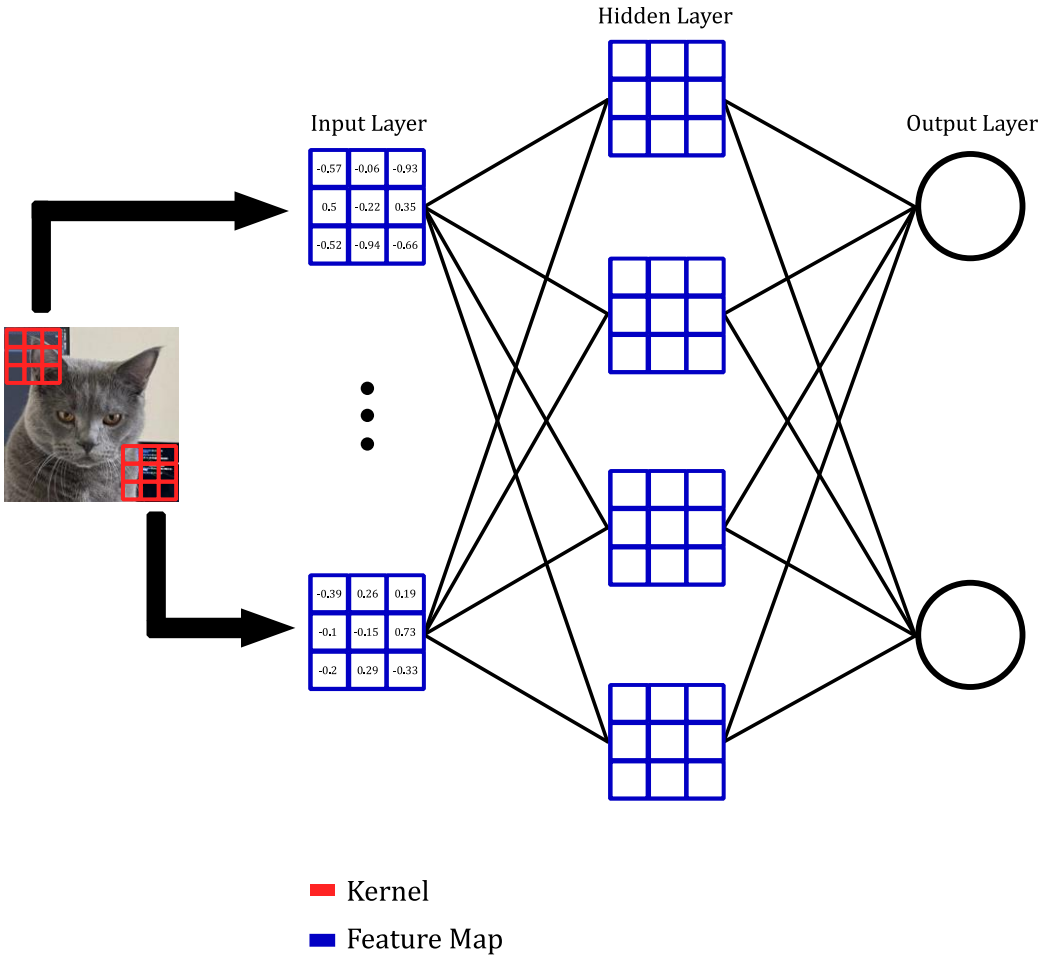


Figure 34. Visual representation of a Convolutional Neural Network.

All the operations that need to be performed to train a CNN are the same as the regular neural network demonstrated previously (Figure 35).

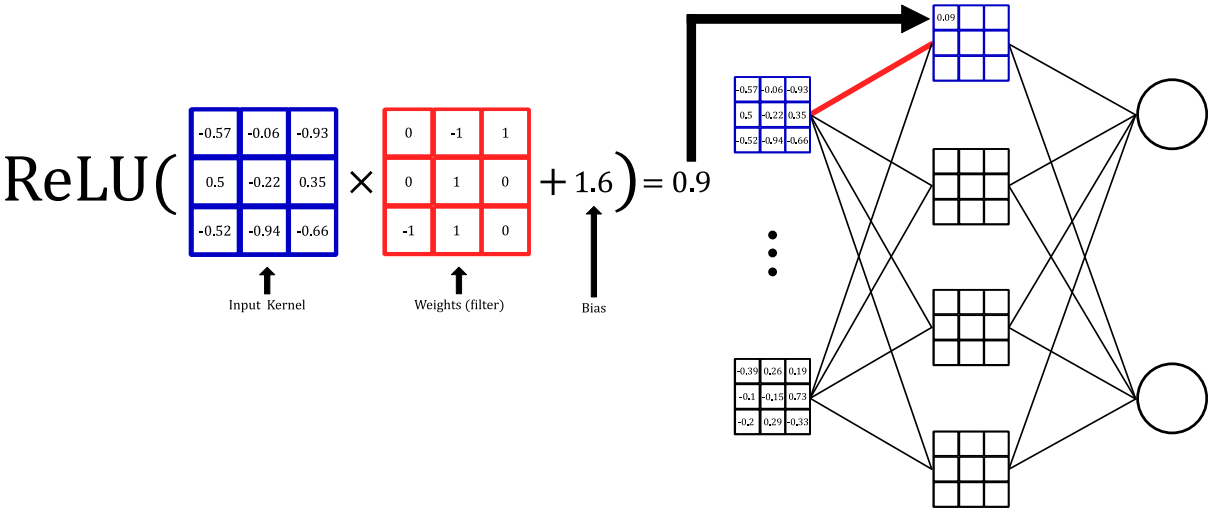


Figure 35. Example of the operations carried out on the input of a CNN.

An operation that is commonly found in CNNs that isn't utilized in regular neural networks is "max pooling". Max pooling reduces the dimensionality of a given input by having a kernel move across the input, and extracting the largest value in the kernel to create an output (Figure 36).

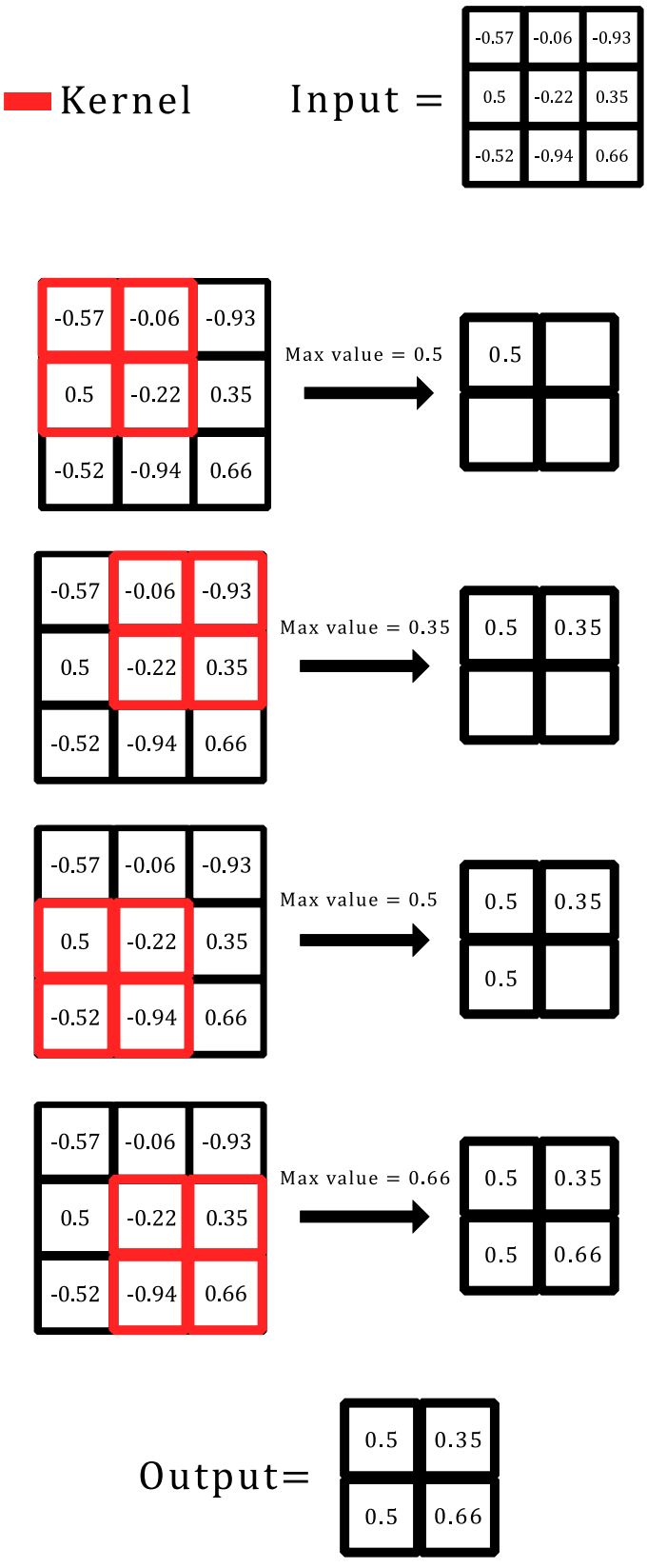


Figure 36. Example of the Max pooling operation.

When a CNN is fully trained, each filter is in some way a representative "feature" from the training data. Research conducted by engineer at Google sought out to visualize the

features learnt by CNNs. The CNN employed for their research is called GoogLeNet (Szegedy et al., 2014), which was trained with a dataset called ImageNet (Deng et al., 2009). This dataset consists of labelled images of various animals and objects. The two primary visualizations that are of interest in the context of this project are:

1. The visualization of an individual filter.
2. The visualization of an entire hidden layer's activations combined.

To achieve these visualization, a loss function is created to maximize the values of a given filter or hidden layer in a CNN, then stochastic gradient descent is used to adjust the values of the input image (Chollet, 2018). The more times this process is iterated upon, the more the activation/activations are maximized, resulting in a more processed output (Figure 37).

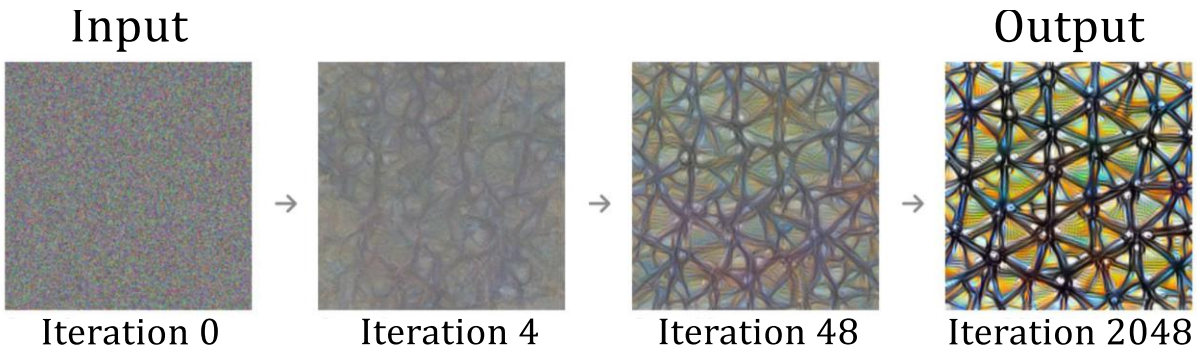


Figure 37. Visualization of a filter using optimization with random noise as an input. (Olah et al., 2017)

The representative features from each layer gradually increase in complexity from the first hidden layer to the last. For instance, the first layer may extract features such as edges and lines (Figure 38A). The intermediate layers textures, patterns, and complex shapes (Figure 38B-D), and the final hidden layer more complete imagery such as a building or a dogs face (Figure 38E) (Olah et al., 2017).

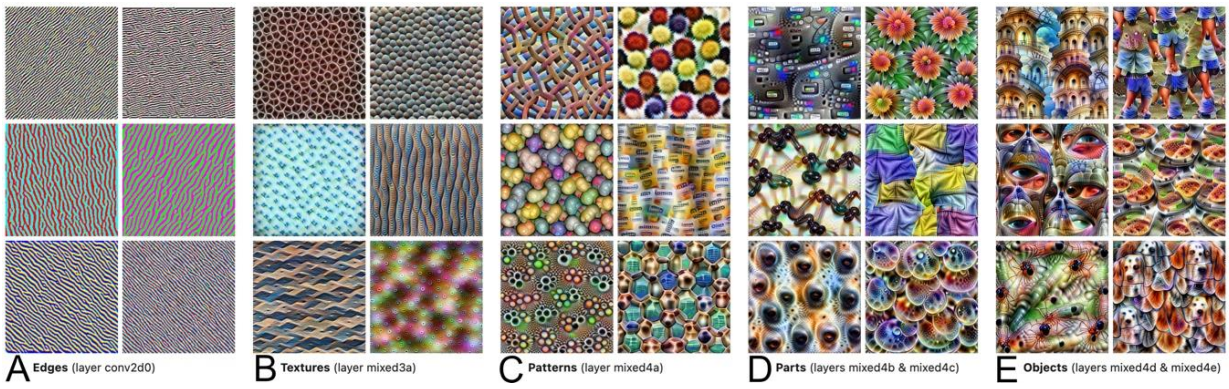


Figure 38. Filters visualization from layers in the GoogLeNet model trained with ImageNet dataset. This demonstrates how the model has a hierarchical perception of the data. (Olah et al., 2017)

This hierarchical build-up of features occurs because of the max pooling operation. As the image is down sampled after each convolutional layer, a hidden layer's kernel will be able to "cover more area" of its receiving input (Figure 39).

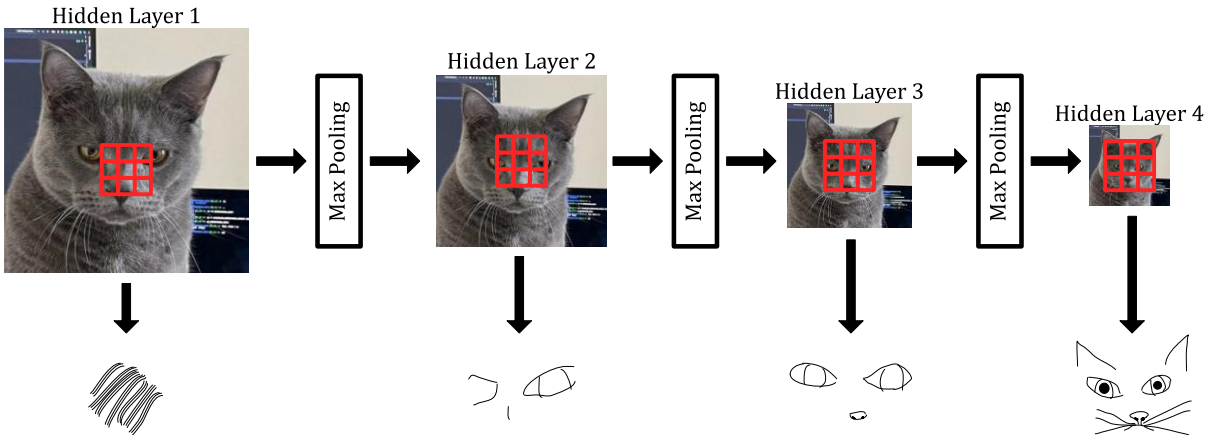


Figure 39. Demonstration of how max pool results in hierarchical feature learning.

As mentioned previously, all the activations of a hidden layer combined can also be visualized. This results in an image processing effect that is popularly referred to as "DeepDream" (Figure 40).

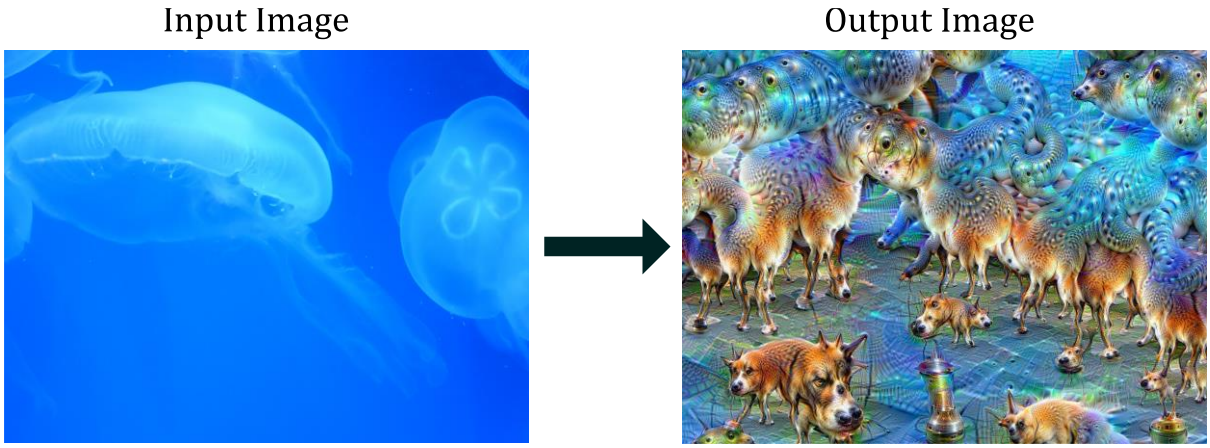


Figure 40. Example of a processed using the DeepDream effect, visualizing a layer's output with many optimization iterations (Thoma, n.d.).

Any convolutional layer from a CNN trained to classify images can be used for this kind of image processing effect, and since CNN's extract features in a hierarchical manner, which layer is selected has a vastly different resulting output. (Figure 41).

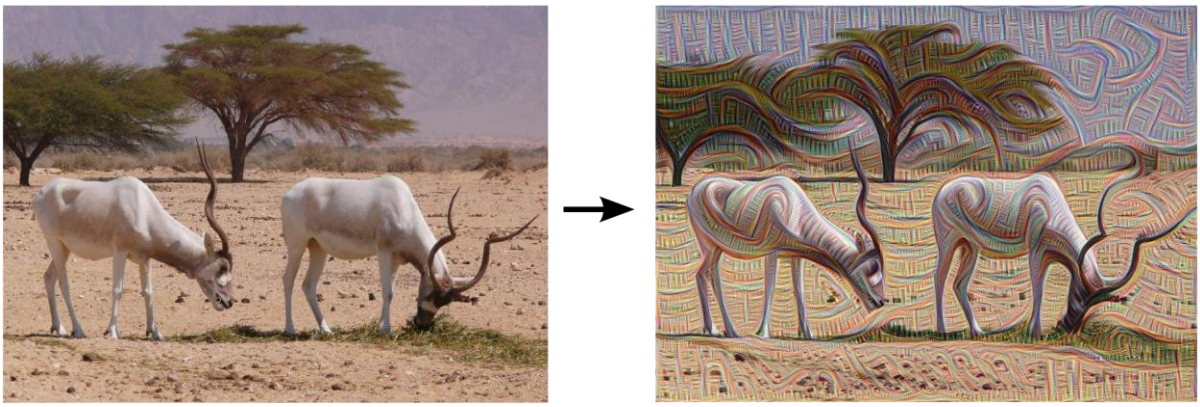


Figure 41. Example of an image processed using the DeepDream technique. The filters being optimized are from an early layer in the network. Left: Original photo by Zachi Evenor. Right: processed by Günther Noack, Software Engineer.

2.3 Can you hear the drums, Fernando?

Like images, the use of CNN's has been widely adopted for solving audio classification problems such as speech recognition, music recommendation systems, and environmental sound classification. Various datasets have become benchmarking standards for new model architectures such as Google's speech command dataset (Warden, 2017) for speech recognition, and the UrbanSound8K dataset (Salamon et al., 2014) for urban environmental sounds. new state-of-the-art CNN based architectures are emerging at a frequent rate. For example the current state-of-the-art CNN trained with

the Urban8K dataset is a model called MCAAM-Net (Das et al., n.d.), which boasts an accuracy of 99.60%.

Despite the success of CNNs for audio classification, the amount of research that's gone into exploring the features learnt by models trained on audio specific data pales in comparison to models trained on image data. One of the few attempts at researching the features learnt by a CNN on audio data attempted to create the DeepDream effect with raw audio data (Ardila et al., n.d.).

The authors employ a six-layer CNN to predict a 100-dimension collaborative filtering track embeddings (100-dimensional vector) (van den Oord et al., 2013) from 30-second clips of music audio as input. Collaborative filtering is a broad concept that is widely used in recommendation systems, for example Spotify's artist recommendation system (Gupta, 2019).

For example, a simple approach of implementing collaborative filtering for a recommendation system to obtain these embeddings, would involve creating a sparse matrix that represents a "rating" value towards a target. The sparce matrix described in the audio DeepDream research contains users' music listening history. For an example, this will be interpreted as the number of listens towards a particular artist (Figure 42). Each user and artist are represented by a vector (which are randomly initialized), these can be n dimensions.

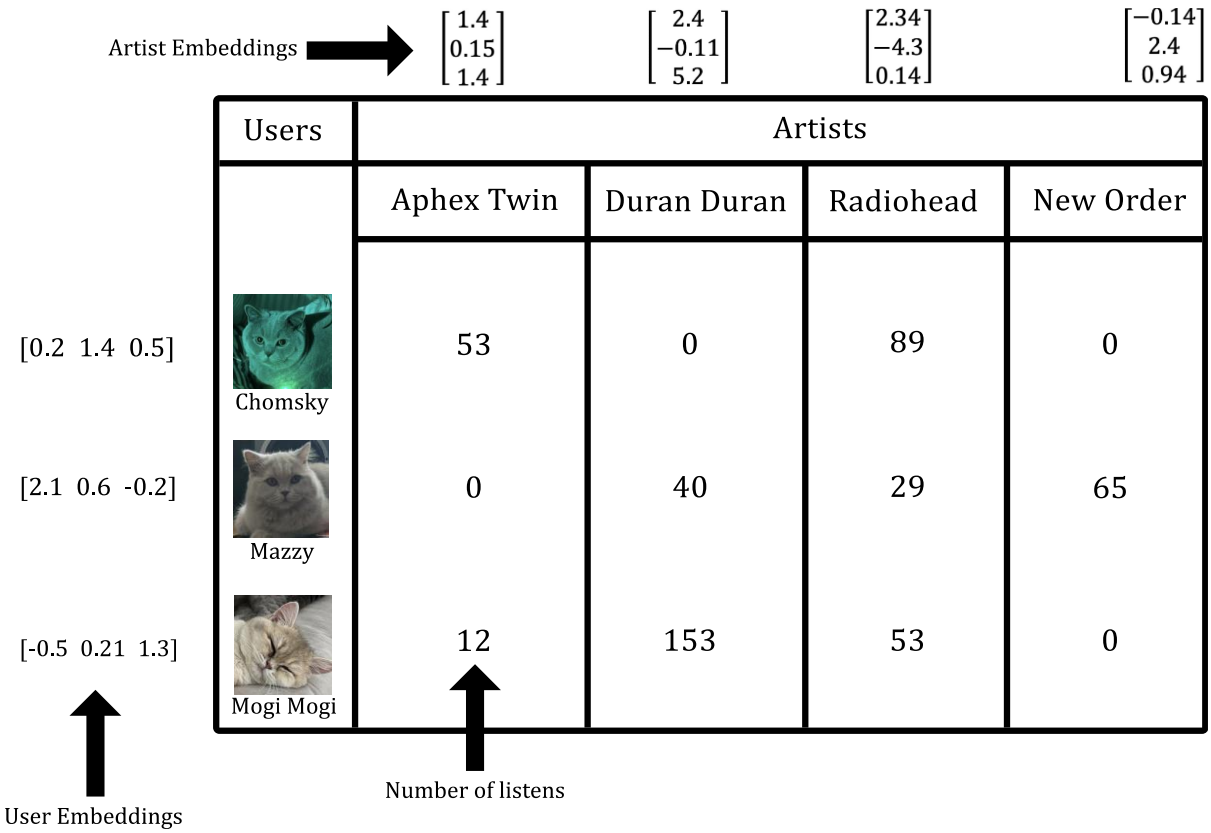


Figure 42. A visual representation of a sparse matrix for a music recommendation system.

The purpose is to train a model to predict the number of listens a user contributes towards an artist from this data. The embeddings that represent the users and artists are used to make this prediction (Figure 43).

$$predictions = user\ embeddings \cdot artist\ embeddings$$

$$[0.2 \quad 1.4 \quad 0.5] \cdot \begin{bmatrix} 1.4 \\ 0.15 \\ 1.4 \end{bmatrix} = 1.19$$

		$\begin{bmatrix} 1.4 \\ 0.15 \\ 1.4 \end{bmatrix}$	$\begin{bmatrix} 2.4 \\ -0.11 \\ 5.2 \end{bmatrix}$	$\begin{bmatrix} 2.34 \\ -4.3 \\ 0.14 \end{bmatrix}$	$\begin{bmatrix} -0.14 \\ 2.4 \\ 0.94 \end{bmatrix}$
	Users	Artists			
		Aphex Twin	Duran Duran	Radiohead	New Order
[0.2 1.4 0.5]	 Chomsky	1.19	2.93	-5.48	3.8
[2.1 0.6 -0.2]	 Mazzy	2.75	3.93	2.31	0.96
[-0.5 0.21 1.3]	 Mogi Mogi	1.15	5.54	-1.89	1.8

Figure 43. All predictions are calculated using randomly initialized embeddings.

In the same way that weights and biases in neural networks need to be adjusted for a desired output, the embeddings need to be adjusted in a direction as to elicit a better prediction. A loss function can be defined for this problem, which then can be optimized with SGD.

The embeddings described in the audio DeepDream research are for individual tracks rather than artists, and as mentioned previously, these were used as the target label for 30 seconds of raw audio as input (Figure 44).

Song: Music for 18 Musicians
Label: [12 84 99 90 4]

Song: I Won't Hurt You
Label: [45 17 74 13 99]

Song: It's Been Eight Years
Label: [34 14 52 15 9]

Song: Gigantic
Label: [95 26 31 3 34]

Figure 44. A visual representation of embeddings representing a song.

Since the dataset used 30 seconds of songs for training, the features the model will be learning will be representative of all the timbral qualities that goes into making song. The intention behind this was to have the model "dream" a piece of music using the DeepDream algorithm on a selected layer from noise or silence as input. However, the results are noisy and do not have any musicality.

Entire clips of songs would have a complex composition of elements that make up the features of a given recording. This leaves little to no room for subtle signal processing to an input audio signal. Therefore, the implementation that has been carried out in this research focuses the dataset to more simple recordings.

3. Implementation

3.1 Putting Together a Dataset.

Choosing a training dataset was crucial to the project as it would be central to what features a model would learn, and in turn the features the model would amplify onto an input signal. It was decided that the desired features a model would learn would need to be musical in some capacity as opposed to abstract sounds (such as construction sounds).

After some considerations two datasets were chosen (Table 1). The NSynth dataset (Engel et al., 2017), and a selection of the Philharmonia free sample library (Philharmonia Orchestra, n.d.).

NSynth Dataset

Instrument	Number of samples
Bass	68,955
Brass	13,830
Flute	9,423
Guitar	35,423
Keyboard	54,991
Mallet	35,066
Organ	36,577
Reed	14,866
String	20,594
Synth Lead	5,501
Vocal	10,753
Total	**305,979**

Philharmonia Dataset

Instrument	Number of samples
Banjo	145
Bass clarinet	1709
Bassoon	1730
Cello	2104
Clarinet	2306
Contrabassoon	2150
Cor Anglais	3656
Double Bass	1965
Flute	2183
Guitar	459
Mandolin	199
Oboe	1131
Saxophone	1716
Viola	2021
Violin	2310
Total	**25,784**

Table 1. NSynth and Philharmonia datasets overview.

These two datasets feature recordings of various isolated orchestral instrument. This would mean the features that the network needs to learn to classify the instruments correctly are the features that make up an instrument's timbre.

The timber of an instrument is how we, as humans, distinguish instruments apart from one another. Timbre is not particularly easy to quantify, and cannot be represented with a single variable, but rather a combination of abstract variables (Erickson, 1975). For example, the way an instrument is "struck" can be considered one of these abstract variables. For instance, a piano cannot "swell" in a note the way a violin could.

Each instrument class in the NSynth dataset were also organized into subclasses of "acoustic", "electronic", and "synthetic" (Table 2).

Family	Acoustic	Electronic	Synthetic	Total
Bass	200	8,387	60,368	68,955
Brass	13,760	70	0	13,830
Flute	6,572	35	2,816	9,423
Guitar	13,343	16,805	5,275	35,423
Keyboard	8,508	42,645	3,838	54,991
Mallet	27,722	5,581	1,763	35,066
Organ	176	36,401	0	36,577
Reed	14,262	76	528	14,866
String	20,510	84	0	20,594
Synth Lead	0	0	5,501	5,501
Vocal	3,925	140	6,688	10,753
Total	**108,978**	**110,224**	**86,777**	**305,979**

Table 2. NSynth ontology with subcategories.

All the recordings where manually vetted to ensure that the instrument label matched the actual timbral quality of the instrument as it is commonly recognised. This led to omitting entire sub classes for certain instruments. For example, all the "synthetic" guitar samples were removed from the dataset because they did not resemble the typical sound of a guitar. Additionally, the "keyboard" class was renamed to "piano", and samples that did not resemble an acoustic piano were removed. The overall number of samples was reduced after this process (Table 3).

Instrument	Number of samples
Bass	8,587
Brass	13,760
Flute	6,607
Guitar	30,148
Keyboard	54,991
Organ	36,577
Piano	6,600
Reed	14,338
String	20,510
Synth Lead	5,501
Vocal	3,925
Total	**146,553**

Table 3. Number of samples per instrument after omitting undesirable samples.

As shown in Table 1, the two datasets have differing ontologies. The Philharmonia recordings are organized into more specific instrument classes such as “violin”, “viola”, and “cello”. While the NSynth dataset has a broader classification for these instruments, which in this case would be “string”. To merge the two datasets, the recordings from the Philharmonia dataset were moved to better fit the broader classifications of the NSynth ontology (Figure 45). The “banjo” and “mandolin” instrument classes from the Philharmonia dataset were omitted from this merger because they did not fit into any of the NSynth classes. Additionally, the “mallet” class from the NSynth dataset was also omitted. This decision was taken because this was the only class that featured percussive instruments, while all the other classes are melodic. Therefore, it was decided that the features that would be learnt from these recordings would be undesirable.

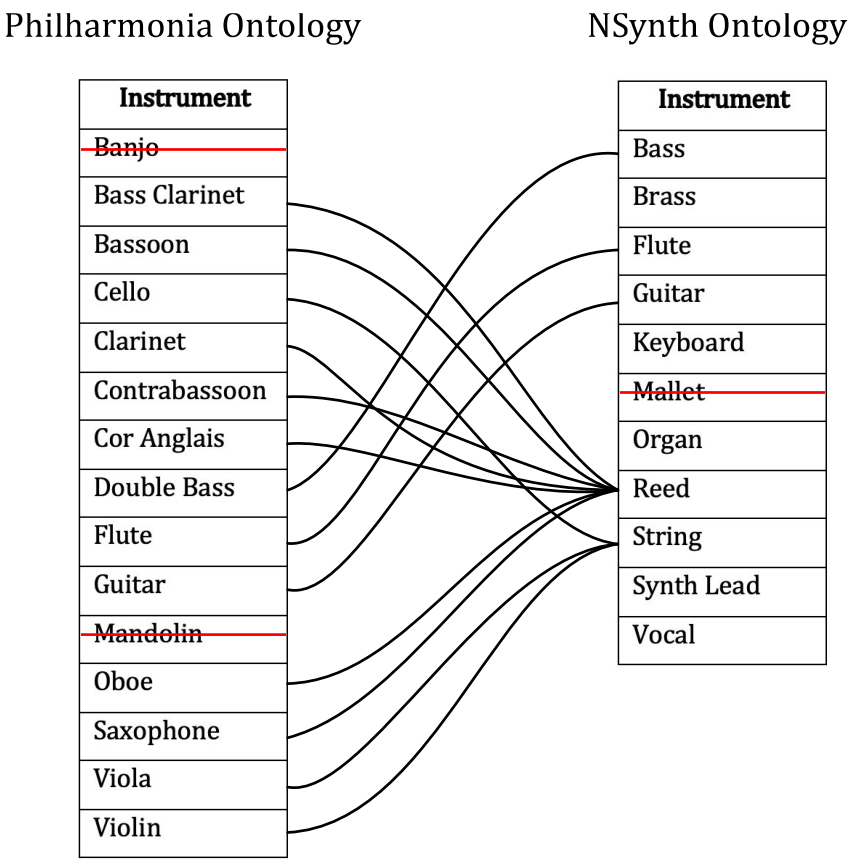


Figure 45. Fitting the Philharmonia ontology into the NSynth ontology.

Both datasets also provided information about each recordings pitch. Some thought went into how this data could be leveraged to improve the outcome of the DeepDream algorithms output, but these were never implemented to scale down the complexity of the project.

3.2 Processing the Dataset.

Audio data is represented as a time series signal, representing an amplitude measurement at each timestep. This amplitude measurement is referred to as a sample, and is measured in decibels (dB). The amplitude measurements are taken at fixed intervals of time at a given rate, this is referred to as the sample rate**.** A common sample rate for audio data is 44,100 Hz. This would result in each second of audio data containing 44,100 samples. For training a neural network, this sample rate can be reduced to speed up the time it takes to train. Therefore, in this study all the recordings in the dataset were reduced to a sample rate of 22,050 Hz.

A Python package called “Librosa” (McFee et al., 2022) was utilized for loading the recordings from the dataset. Librosa offers numerous audio related functionality for audio manipulation and analysis. Once a recording from the dataset is loaded, the time series signal is represented as a Numpy array (Harris et al., 2020). Numpy is a commonly utilized Python package for vector/array data types and operations associated with them.

Each recording in the dataset has a segment of time at the beginning and the end, where the instrument does not sound out a note. These blank segments are redundant because they do not provide any timbral information for classifying the instrument. Therefore, librosa was used to “trim” the beginning, and the end of each recording.

The portion of these segments that were trimmed was determined by selecting all the samples at the start, and the end of the recoding, that were below a determined dB (a threshold dB), and removing them from the signal (Figure 46).

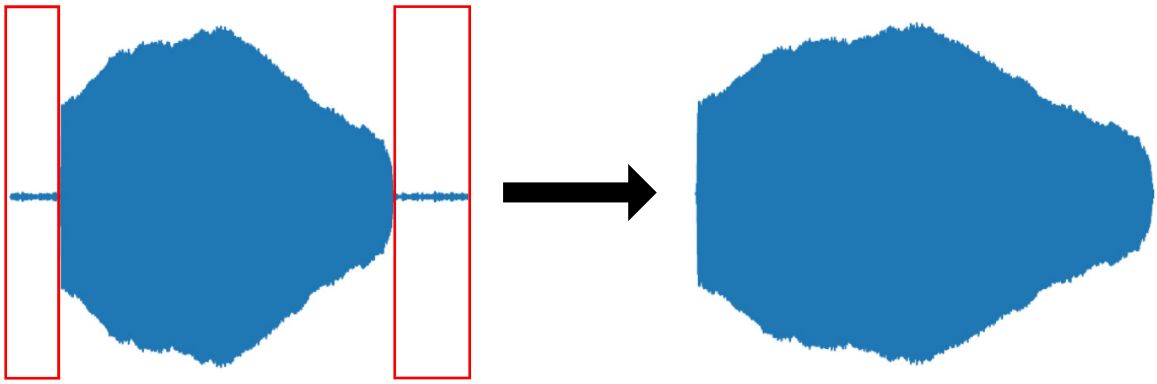


Figure 46. Visualization of trimming a recording where instrument isn't sounding a note.

Each recording in the dataset varies in duration, this is undesirable when training neural networks because it will result in all the data being differently shaped matrices. This was addressed by segmenting each recording into one second segments (Figure 47), this would make each segment’s matrix shape $1 \times 2{,}050$.

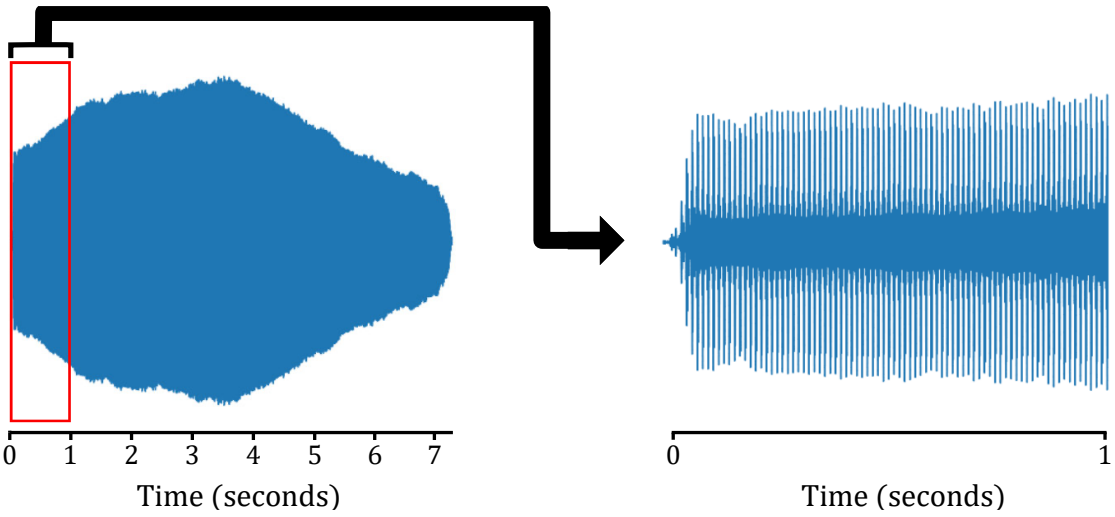


Figure 47. First one second segment from recording.

The red window from Figure 47 represents one second. It can move to another starting position by any amount (in samples). this is referred to as the hop amount. A greater hop amount would create fewer slices compared to a smaller hop amount (Figure 48).

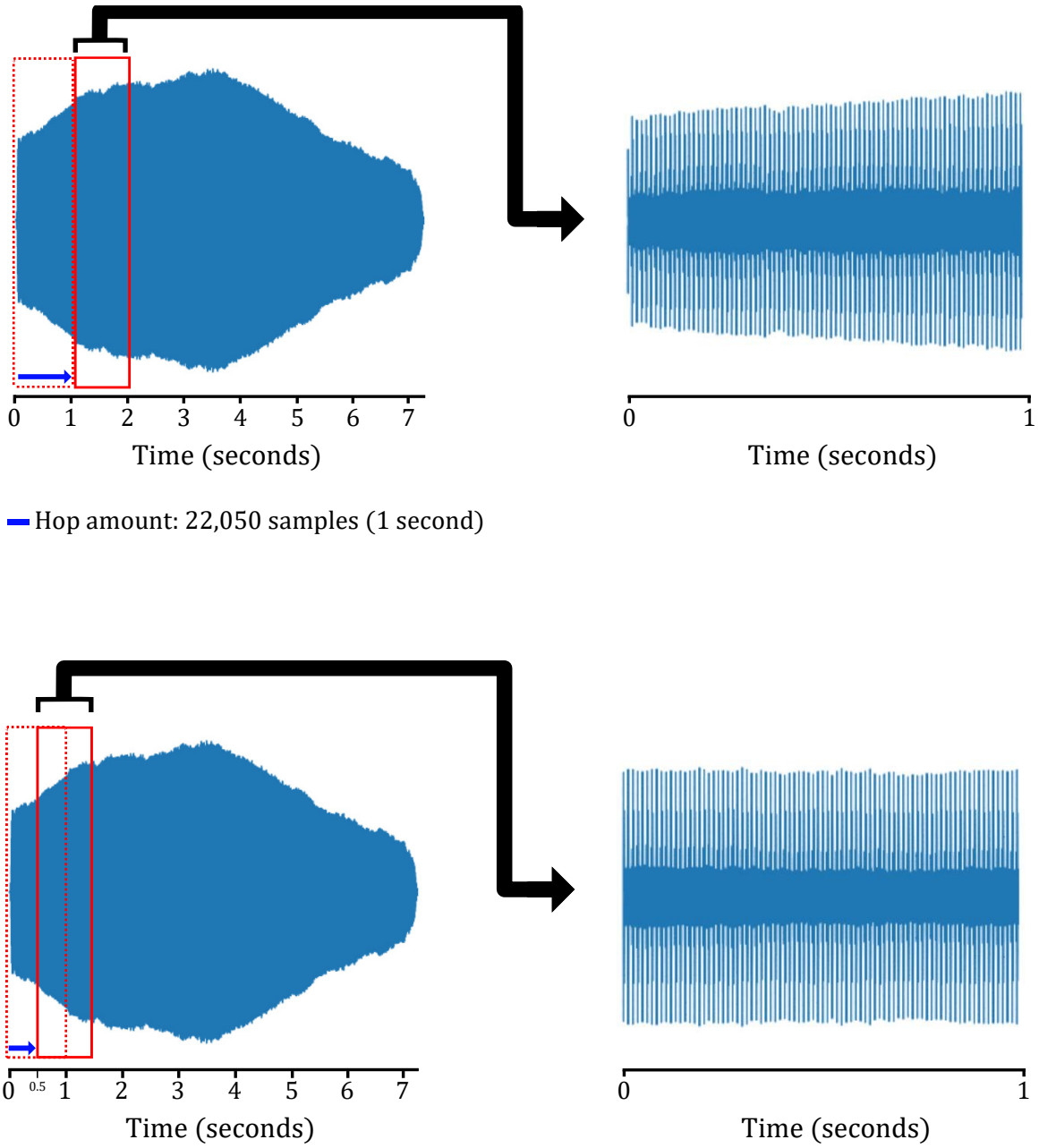


Figure 48. Comparison of a one second hop amount, and a 0.5 second hop amount.

The ability to set the hop amount was used to create more segments for underrepresented instruments in the dataset. This is further discussed in 3.3 Data Augmentation.

The last segment of the signal has the potential of being a shorter duration than one second. This was addressed by leveraging Librosa functionality to apply "zero padding" to the end of the segment. This involves adding samples at the end of the signal to make

up the difference in time, to have the final segment be one second in duration. Each of these samples is an amplitude measurement of 0 dB (Figure 49). This is so that the learning algorithm is not biased (Sinha et al., 2020).

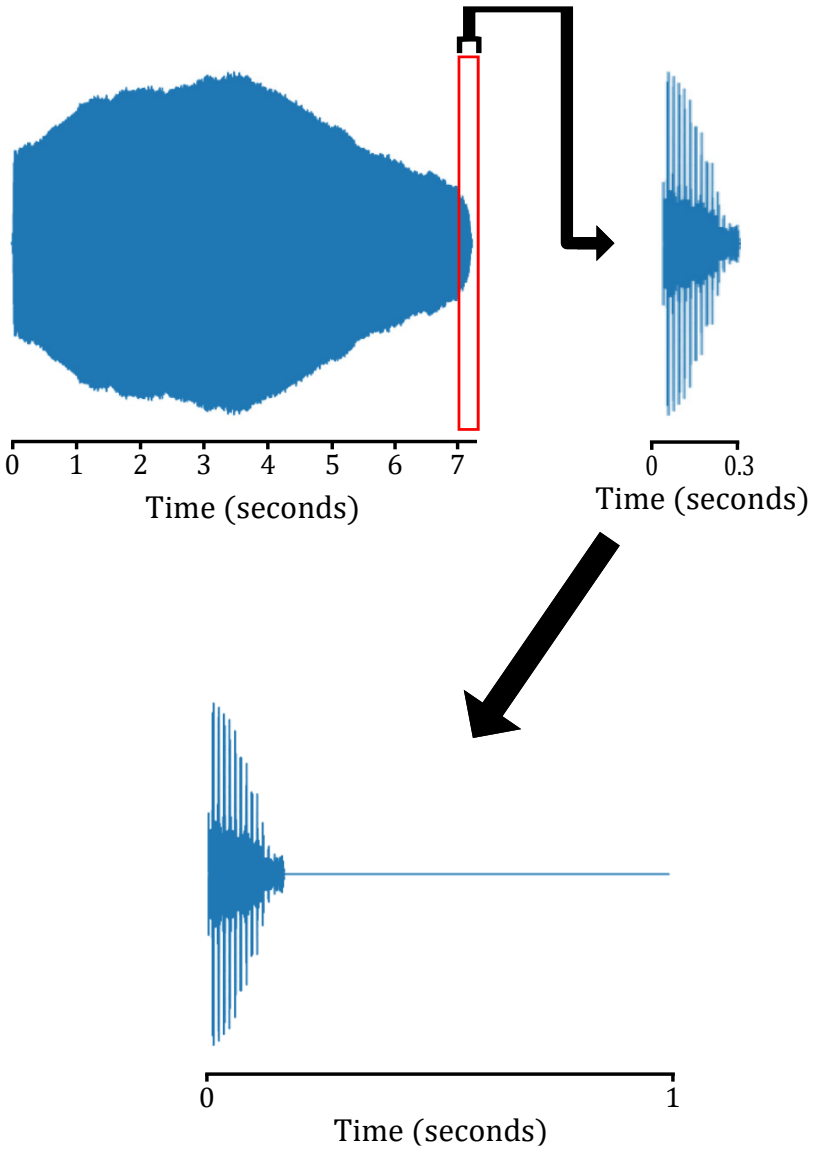


Figure 49. Apply zero padding to the final segment.

3.3 Data Augmentation.

The data distribution of the instruments was examined after all the recordings were segmented. This was done to determine if the dataset represents the classes equally. This would ensure that the learning algorithm is not bias towards more represented instrument.

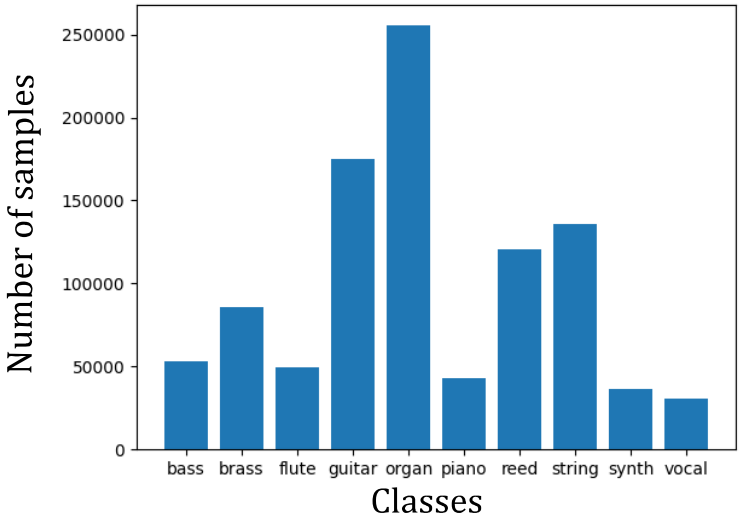


Figure 50. Bar chart shows distribution of samples.

Figure 50 shows that the classes are not equally represented in the dataset. This was resolved by employing two audio data augmentation methods on the pre-segmented dataset. Time stretching, and dynamically setting the “hop amount" (time shifting) that was previously mentioned in Figure 48.

Time stretching involves shortening or extending the duration of the signal without affecting its pitch (Paralkar, 2020). Librosa’s time stretching functionality was used for the bass, brass, flute, piano, synth, and vocal classes as these are the most underrepresented in the dataset. Each recording in these classes was time stretched by a factor of 0.5, resulting in a signal double the duration it originally was (Figure 51).

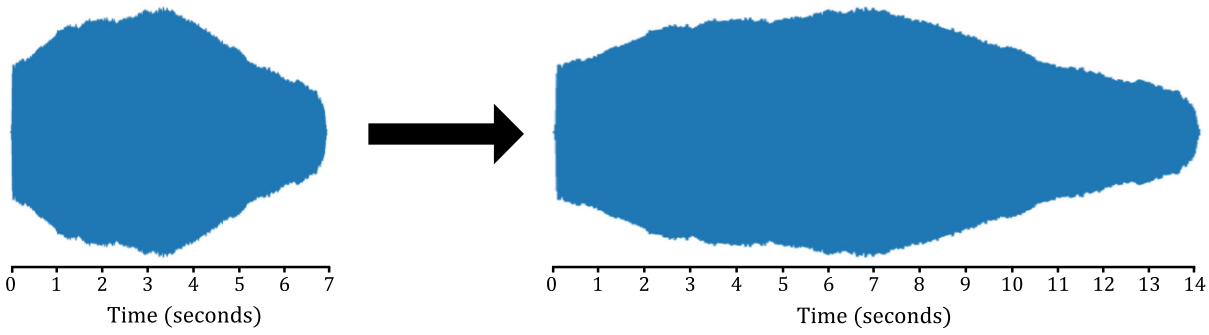


Figure 51. Example of an audio signal being time stretched to be twice its original duration.

After the most underrepresented classes were time stretched, the entire dataset was once again segmented into one second segments with a hop amount determined by the number of samples in each class. For this, the K-means clustering algorithm found in a

Python package called “sci-kit learn” (Pedregosa et al., 2011) was employed to create four clusters of instruments, based on the number of recordings each class has associated with it. Then the hop amount was set by the cluster the instrument belonged to (Figure 52).

Cluster 1		Cluster 2	
Bass Flute Piano Synth Vocal	Hop amount = 5,512 samples	Reed Brass	Hop amount = 7,350 samples
Cluster 3		**Cluster 4**	
String	Hop amount = 11,025 samples	Guitar Organ	Hop amount = 22,050 samples

Figure 52. Resulting clusters from K-means clustering based on number of samples per instrument.

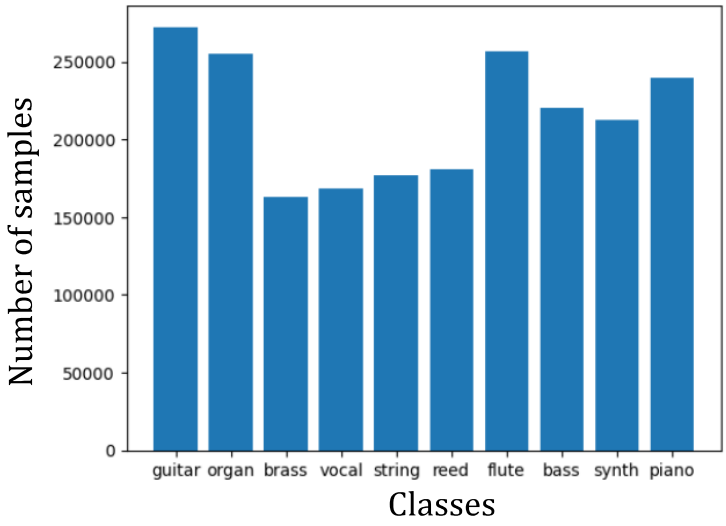


Figure 53. Instrument distribution after data augmentation.

Figure 53 shows that a significant number of samples were created for the underrepresented classes with the data augmentation techniques employed. To fully even out the distribution, functionality was implemented to load the dataset at a set number of samples per instrument, this is further elaborated on in section 3.6

3.4 Encoding the Dataset.

With Librosa’s audio analysis functionality, each of the one second segments were encoded into Mel spectrograms. A Mel spectrogram is a two-dimensional feature map

that represents the frequencies of the signal over time. The $x - axis$ represents time, The $y - axis$ represents the frequencies, and the color map represents the amplitude (Figure 54).

Encoding audio data into Mel spectrograms has several benefits over using raw audio data when training CNNs, one of which is significantly faster training. This is because a Fourier transform is calculated on overlapping windowed segments of the signal. In this process, each segment becomes a compressed representation of time, greatly reducing the number of datapoints (Figure 55).

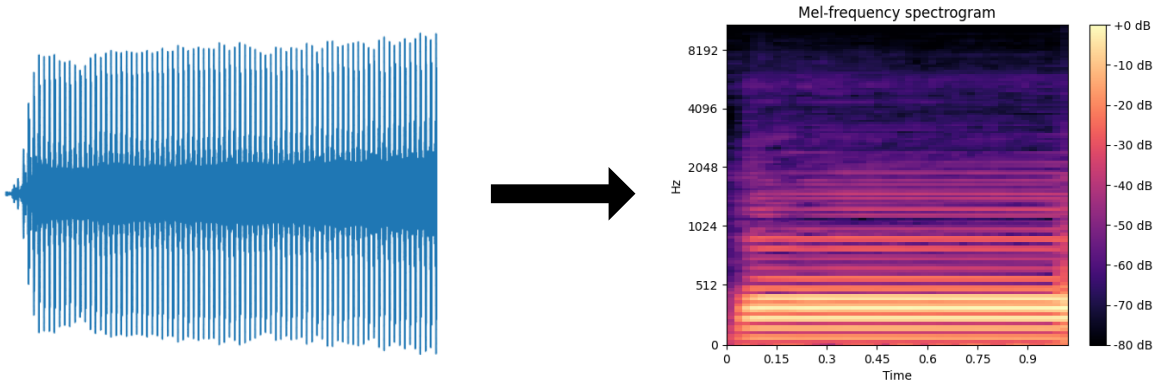


Figure 54. Audio signal converted into a Mel Spectrogram.

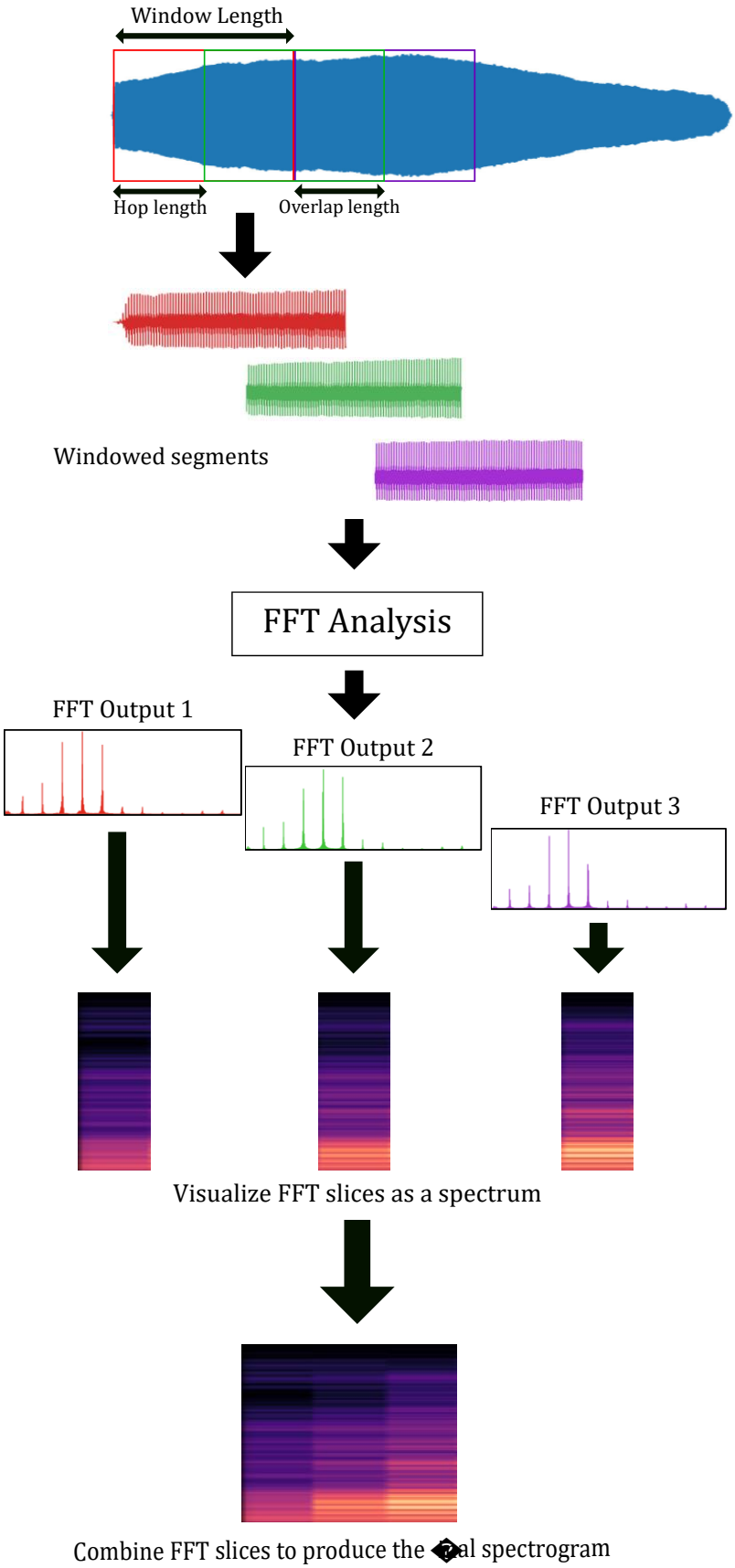


*Figure 55. Steps taken to transform the audio signal into a Mel spectrogram (*Short-Time FFT - MATLAB - MathWorks United Kingdom, *n.d.).*

The resulting frequencies calculated from the FFT are then converted to the Mel scale. The Mel scale is considered to be a more suitable representation of how pitch is perceived (Pedersen, 1965) (Figure 56).

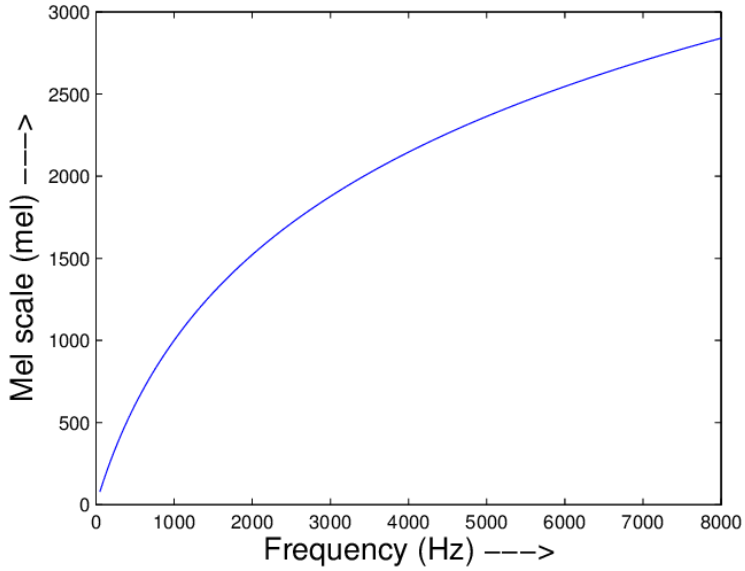


Figure 56. Plot of Mel scale compared to linearly spaced Frequencies (G. & Ramakrishnan, 2007).

Given that the decision to train models using Mel spectrograms was taken, the models learnt spectral features from the data. Therefore, the feature amplification algorithms were performed on input Mel spectrograms as opposed to raw audio data, and the result of the algorithms was a processed Mel spectrogram.

The resulting Mel spectrogram is then converted back into an audio signal. This is achieved by reconstructing the time series audio signal by approximating the Short Time Fourier transform magnitude (Akansu & Haddad, 2001). An unfortunate consequence of encoding audio data into a Mel spectrogram, is that the phase of the signal is not preserved. Functionality from Librosa was used to calculate an approximation of the phase with an algorithm called Griffin-Lim (Sharma et al., 2020).

3.5 Designing and building models.

The Python package Keras (Chollet & others, 2015) with Tensorflow (Martín Abadi et al., 2015) were employed in this project. Tensorflow comes with tools and functionality for building various types of neural networks such as CNNs, and Keras is often paired with

Tensorflow to provide a high-level application programming interface (API) for creating, training, and evaluating Tensorflow models.

When designing models', numerous decisions can be made for determining the architecture of a model. For example, the number of hidden layers, the number of filters for each hidden layer, the optimizers learning rate, and various other parameters. These are referred to as "hyper parameters", and they can be thought of as a vector in a space (Figure 57).

$$\begin{bmatrix} number\ of\ layers \\ number\ of\ units \\ Learning\ rate \end{bmatrix} \longrightarrow \begin{bmatrix} 6 \\ 32 \\ 0.01 \end{bmatrix}$$

Figure 57. Vector representation of hyper-parameters

To accommodate with the dynamic nature of designing models, functions which took hyper parameters as an input, and constructed a model accordingly were created. This allowed for a very large variety of models that could be build, trained, and tested with ease. In the code base these functions are referred to as "model builders". One of the model builders constructs a model that is inspired by the VGG16 (Simonyan & Zisserman, 2015) model architecture (Figure 58).

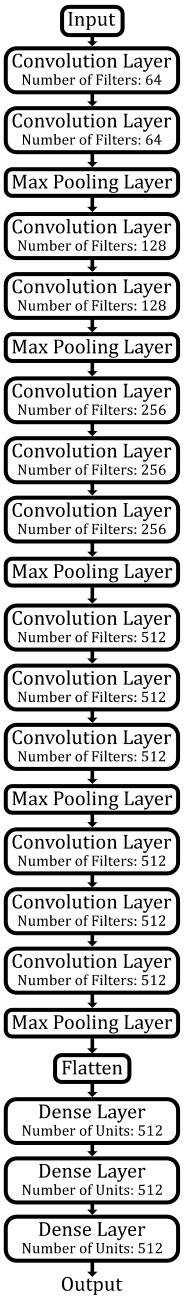


Figure 58. VGG16 model architecture (Bansal, 2020). The number of filters gradually increases in powers of two form the input layer to the output layer.

VGG16 was once a state-of-the-art model for image recognition on the ImageNet dataset. The implementation in this study differs to VGG16 in two ways. It can have a lesser or greater number of convolutional layers compared to the original, and it also has a dropout layer after each convolutional layer.

Dropout layers are used to combat over-fitting. They work by randomly selected units in a layer, and setting their output to be a zero. This effectively "turns off" those units. The

dropout layers are only part of models during the training process. It is not desirable to randomly turn off units when asking the model to make a prediction.

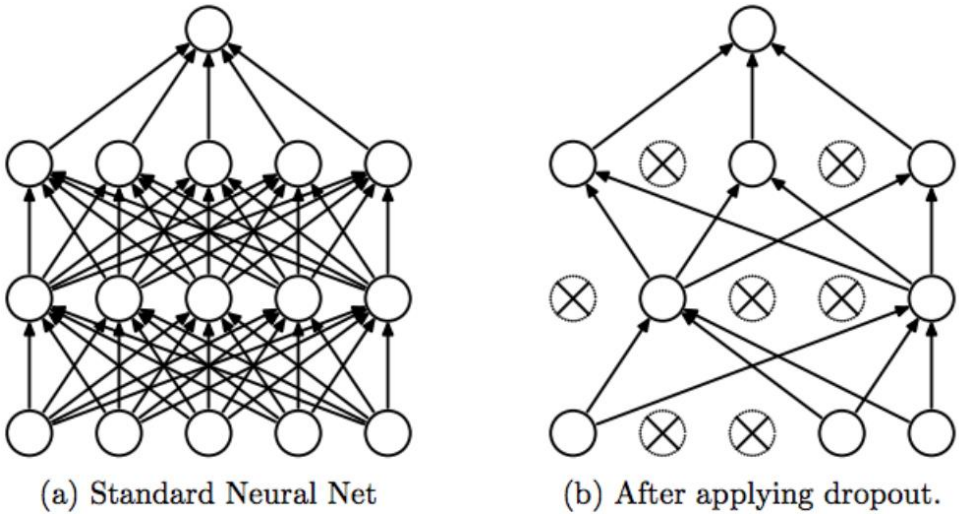


Figure 59. Visualization of the effects dropout layers have on a neural network. (Maklin, 2019)

There are several other model builders which all have different architectures that differ from one another.

3.6 Go Big Data, or Go Home.

Given the size of the dataset, a very large amount of random-access memory (RAM) would be required to load the entire dataset at once. This problem was overcome by creating what the Tensorflow ecosystem referred to as a "data generator". A data generator simply handles the loading of data in batches during the training process, thus making the RAM consumption a fixed amount (Youssef, n.d.).

The data generator implementation for this project stores all the relevant information about the data in a Pandas (McKinney, 2010) "data frame" (Table 4). Pandas is a Python package that provides data structures (one of which is the aforementioned "data frame") that can be visualized in a tabular manner, as well as numerous data analysis and manipulation tools.

index_number	path	instrument	pitch	instrument_label	pitch_label
1424528	/mnt/datasets/serialized_dataset/reed/reed_C#_012878_segment_6.npy	reed	C#	[0. 0. 0. 0. 0. 1. 0. 0. 0.]	[0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
1184151	/mnt/datasets/serialized_dataset/guitar/guitar_E_021464_stretched_segment_5.npy	guitar	E	[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]	[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
383790	/mnt/datasets/serialized_dataset/brass/brass_D#_000835_segment_10.npy	brass	D#	[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]	[0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
294259	/mnt/datasets/serialized_dataset/brass/brass_A#_000359_segment_0.npy	brass	A#	[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]	[0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
1494896	/mnt/datasets/serialized_dataset/brass/brass_C_008526_segment_9.npy	brass	C	[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]	[0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
179069	/mnt/datasets/serialized_dataset/brass/brass_F#_003926_segment_2.npy	brass	F#	[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]	[0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
644195	/mnt/datasets/serialized_dataset/reed/reed_C_002849_segment_7.npy	reed	C	[0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]	[0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
953813	/mnt/datasets/serialized_dataset/organ/organ_F#_003729_segment_2.npy	organ	F#	[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]	[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]

Table 4. Example of a select number of data samples stored in a Pandas data frame.

The process of creating these data frames involves iterating over the file structure in which the dataset is located, and obtaining the full system path to the individual samples. The file name of each sample contains the instrument class to which it belongs to, as well as its pitch data. This information is extracted using regular expression (commonly referred to as “regex”), a language that is used for searching through text data based on pattern matching. A one-hot encoded vector representation of the instrument, and the pitch data is also created for each sample, and stored in the data frame.

Functionality was also implemented to set the number of samples per instrument to be stored in the data frames. This allowed for the option to use a small amount of data, which was beneficial in testing the systems created for training model. It also had the added benefit of having a totally even data distribution between all the classes.

Once a data frame is populated, the data generator is then responsible for handling the way the data gets loaded into memory during the training loop, or when evaluating a trained model. In this case all the data was previously stored as a Numpy array (hence the “.npy” file extension), Numpy in turn provides functionally to read and load the data stored in these files. Figure 60 illustrates the flow of training, and evaluating a model with data generators.

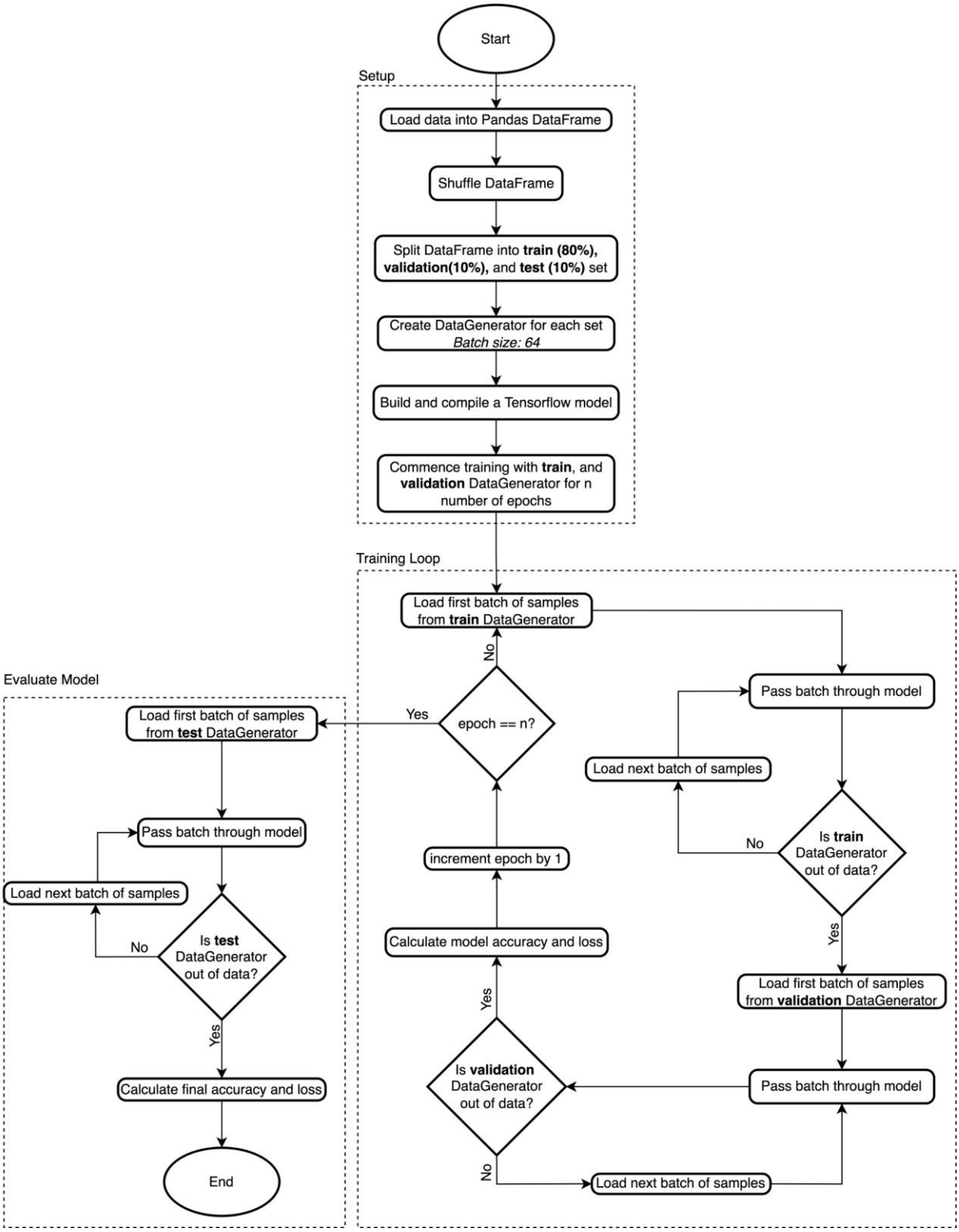


Figure 60. Use of data generators during the training process of a model.

3.7 Hyper-Parameter Optimization, Logs, and the Training Loop.

Depending on the model, the number of combinations the hyper-parameters could be arranged in can become very large, making it difficult to find parameters for an optimal model. Therefore, search algorithms can be employed to automate this process. In this case "Bayesian optimization" was chosen. A Python package called "BayesianOptimization" (Nogueira, 2014) was used for implementation. Bayesian optimization works by finding hyper-parameters within a determined bounds to maximizing the output of a "black box" function (Kenworthy, 2020). In the context of this study, a function was created that was responsible for:

1. Building and compiling a model.
2. Training the model.
3. Loading the model at its optimal epoch.
4. Evaluating the model on the test set.
5. Returning the accuracy score.

By returning the accuracy score, the Bayesian optimizer is searching for hyper-parameters that maximizes the accuracy of the model. The Bayesian optimizer requires a determined number of iterations it should optimize for (i.e., number of models to train). The greater the number of iterations, the better an understanding the optimizer will have of the vector space the search is occurring within. The optimizer will eventually home in onto an area in the space with optimal hyper-parameters for the model being trained.

Since the optimizer will be training numerous models, a system was implemented to keep various logs during this training process. These logs include:

- The hyper-parameters chosen.
- The model architecture.
- The model evaluation.
- Model checkpoints (so trained models can easily be loaded).
- The model learning history.

The models hyper-parameters that are chosen by the optimizer are logged so that the model can be reconstructed if needs be, these were stored in a comma separated value (CSV) format (Figure 5)

model_ID	dropout_amount	learning_rate	num_classes	num_first_conv_blocks	num_fourth_conv_blocks	num_second_conv_blocks	num_third_conv_blocks
3	0.1868955193048339	9.507143064099161e-05	15.0	5.789267873576293	2.248149123539492	2.247956162689621	1.4646688973455957
4	0.43222189674169265	6.011150117432088e-05	15.0	1.1646759543664196	8.759278817295954	7.659541126403374	2.6987128854262092

Table 5. Example of hyper-parameter logs for a custom model builder.

The model evaluation was also stored in a CSV formal (Table 6).

model_ID	loss	accuracy
0	0.2238350361585617	0.9390528202056885
1	0.28409552574157715	0.9221661686897278
3	0.1496812403202057	0.9489518404006958
4	2.738002300262451	0.1418866515159607

Table 6. Example of model evaluation logs.

The models' learning histories were stored using a Python package called Aim (Sogomonian et al., 2019), this package stores all the metrics that can be recorded during the training loops, such as loss and accuracy, and graphs out these metrics on a browser (Figure 61).

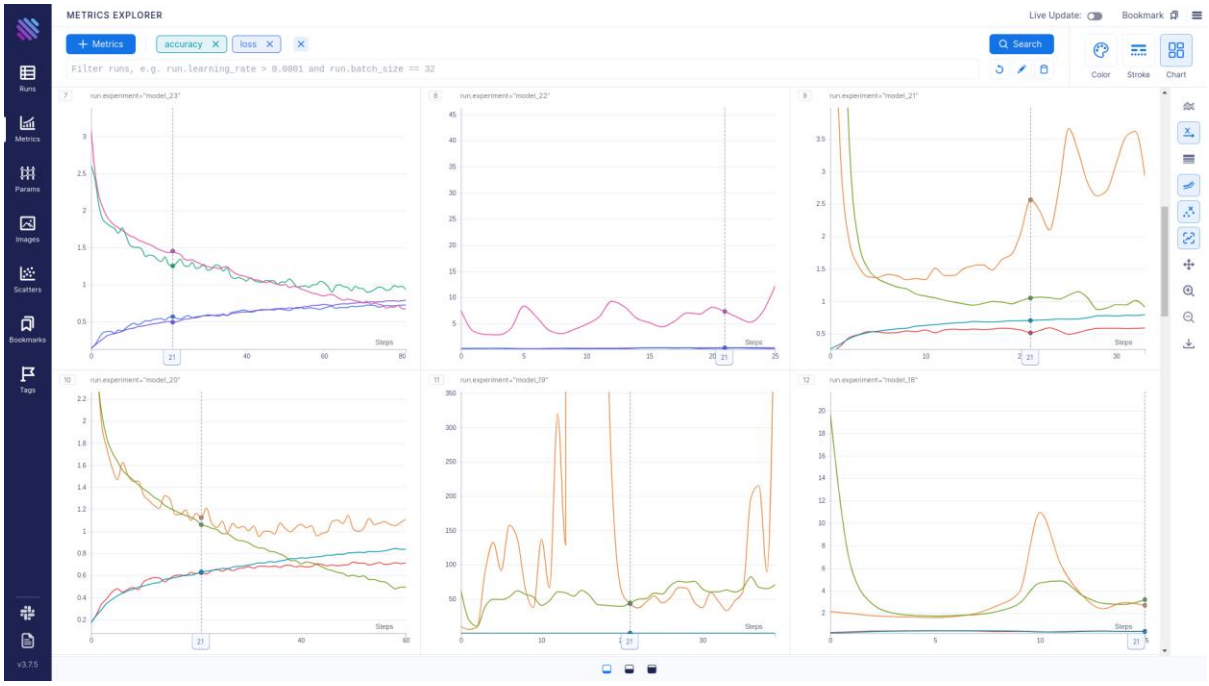


Figure 61. Training history of models graphed on a browser using Aim.

Once a run of training is complete, all logs can easily be examined to compare different models to aid with selection of a model that has the highest accuracy. Figure 62 illustrates the flow of the training process with the Bayesian optimization and the logging system.

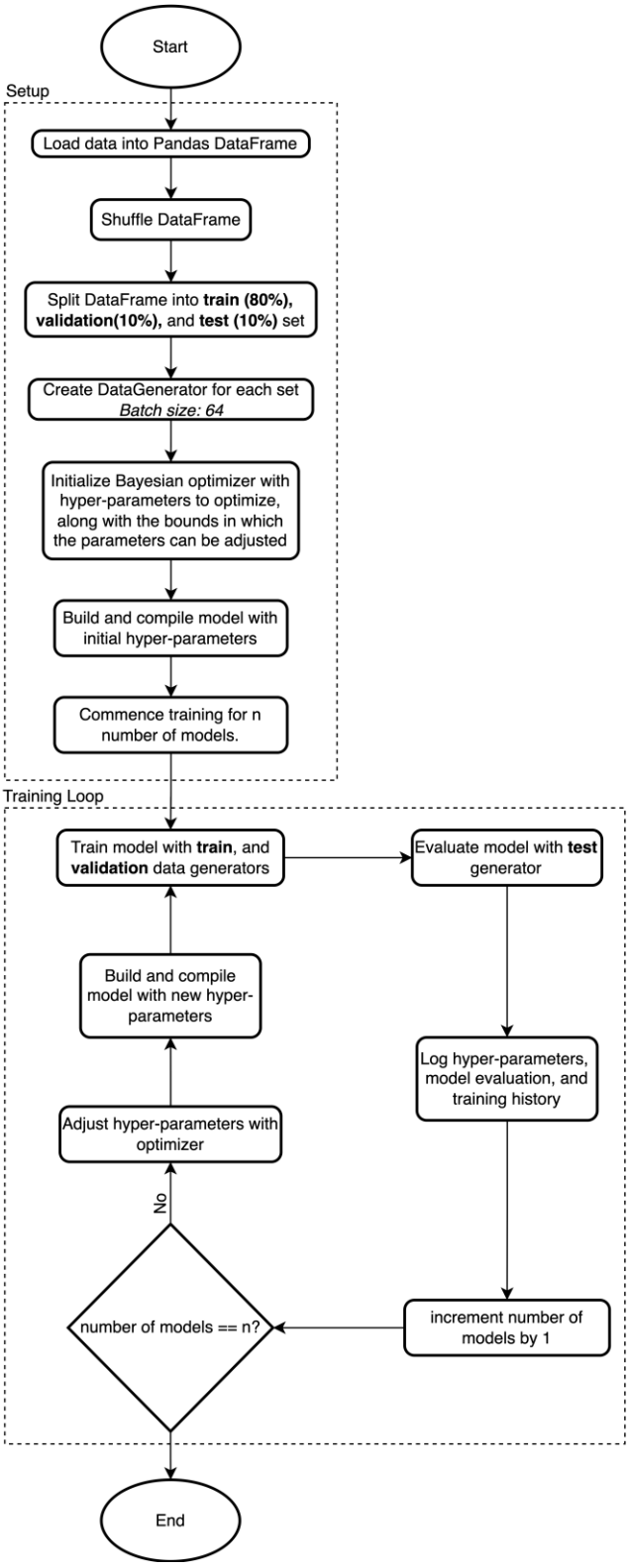


Figure 62. Training loop with hyper-parameter optimization.

3.7 Amplifying Filter Activations.

Once the training process was done, a model was selected to explore the capabilities of processing audio using its filters. The first method carried out involves selecting a filter in the model, and passing through a Mel spectrogram into it, to obtain its output (also referred to as the "activation") (Figure 63).

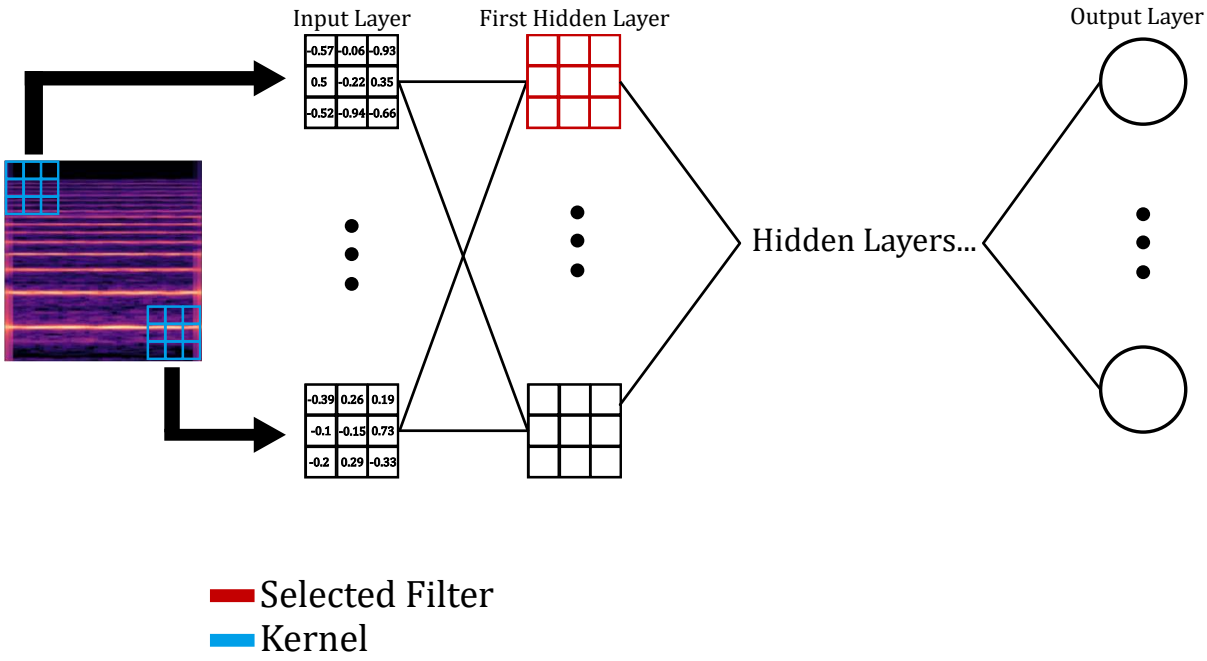


Figure 63. Selection of a filter in a trained model. Any filter from any hidden layer can be selected.

Next a loss function is used to maximize the filters activation. The loss function used in this case is the mean of the activation. The gradient of the loss with respect to the input is calculated, and stochastic gradient ascent (opposite of SGD, maximizes a loss as opposed to minimizes a loss) is used to adjust the values of the input Mel spectrogram, effectively maximizing the activation onto the input. This process can be iterated upon n time, to further maximizing the activation (Figure 64).

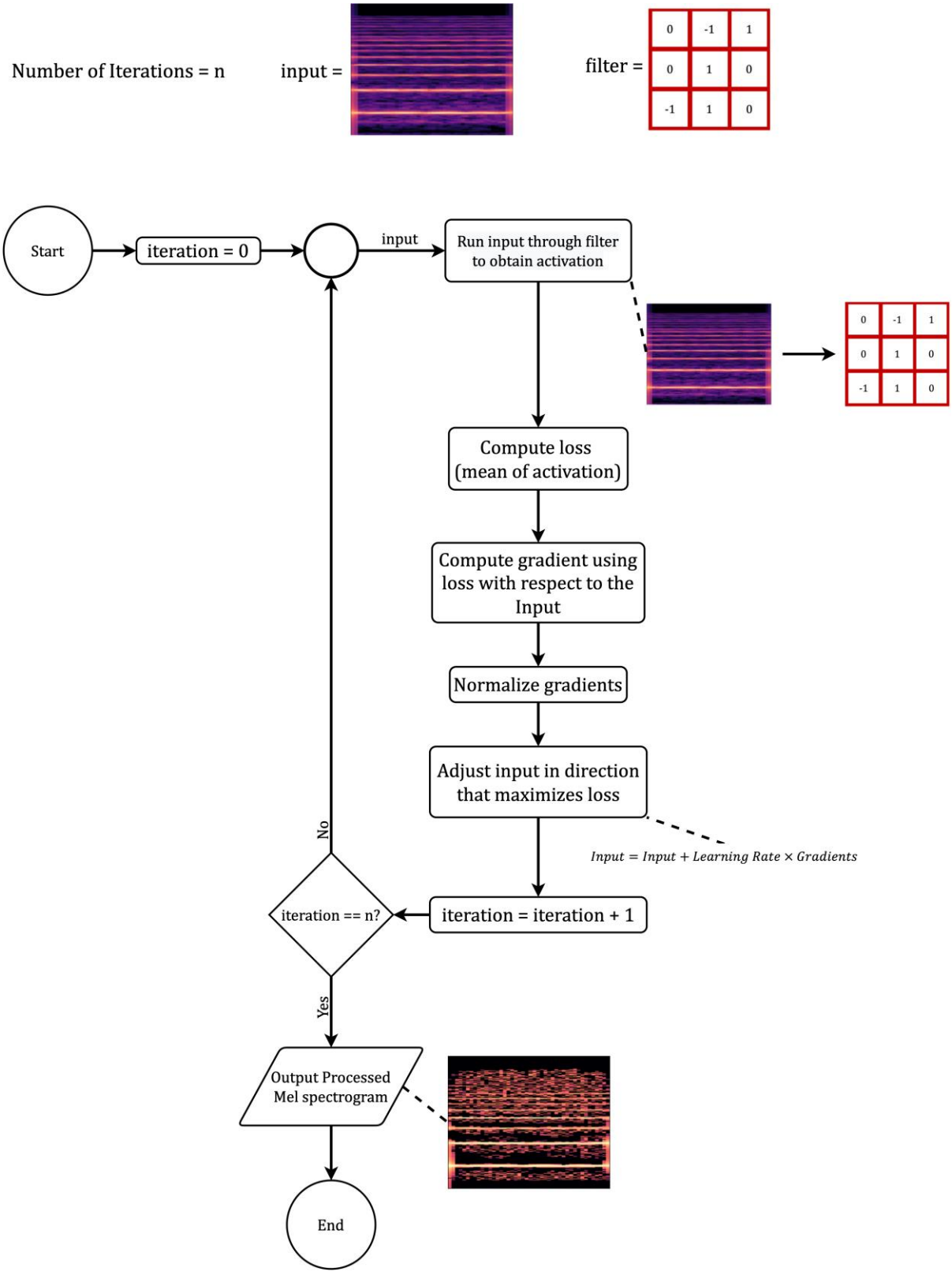


Figure 64. Steps taken to process and input Mel spectrogram using an individual filter from a trained CNN.

A Similar algorithm is employed to maximize all the activations of a hidden layer. First a hidden layer is selected (Figure 65).

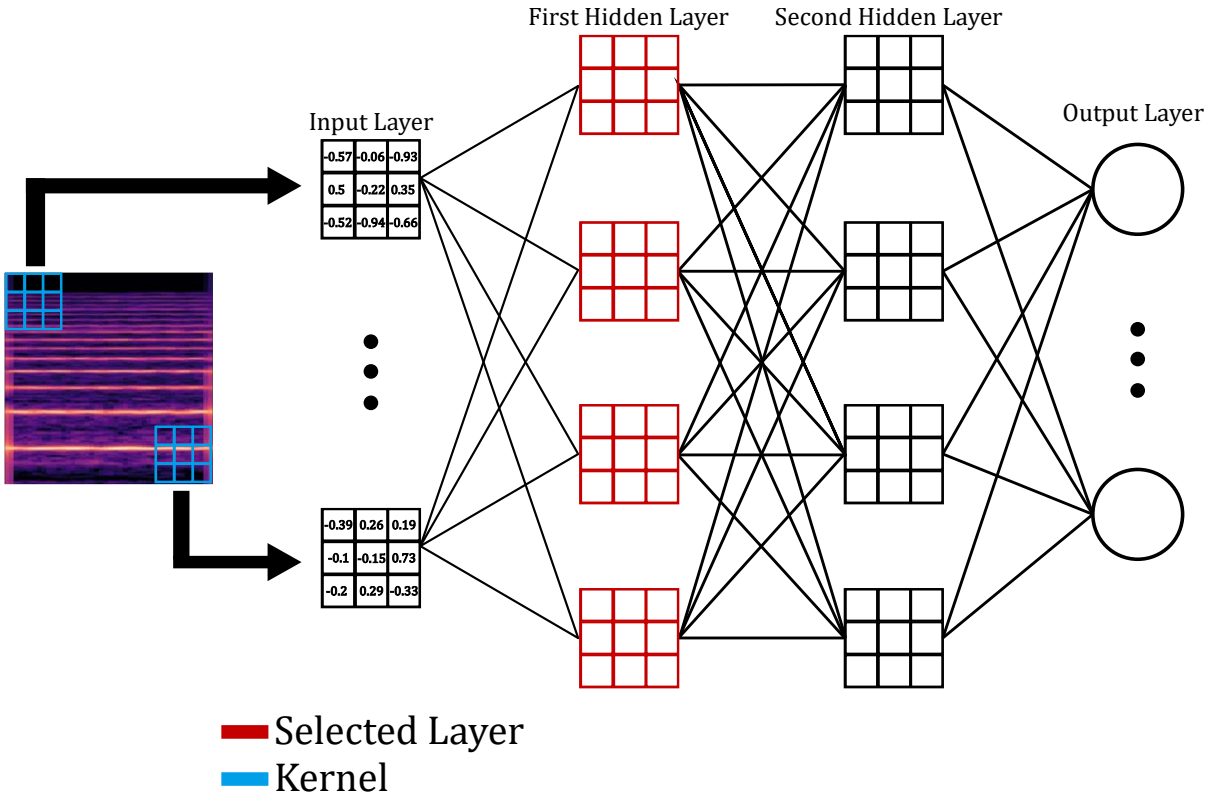


Figure 65. Visualization of selecting all the filters from an entire layer.

Then the Mel spectrogram is passed through all the filters to obtain the activations from the layer, and a loss is calculated. In this case the loss is defined by calculating the mean for each activation, then summing the resulting means together. Then just like the previous algorithm, the gradient of the loss with respect to the input is calculated, followed by gradient ascent to maximize the activation onto the input.

4 Testing

The development of this study was partially done using Jupyter notebooks (Kluyver et al., 2016). A Jupyter notebook is a python environment that encourages code to be organized into modular blocks, which can be executed in any order. These notebooks also allow media such as audio and images to be displayed as an output from a code block. Due to this, these notebooks were heavily utilized to run experiments, develop algorithms, and demonstrate the functionally of systems.

4.1 Proof of Concept.

A proof-of-concept test was carried out before any models were trained. For this InceptionV3 (Szegedy et al., 2015) was used to process a Mel spectrogram. InceptionV3 is an image classification model trained on ImageNet. Using a model trained with images to process Mel spectrograms is possible because the matrix representation of images and Mel spectrograms resemble one another (Figure 66).

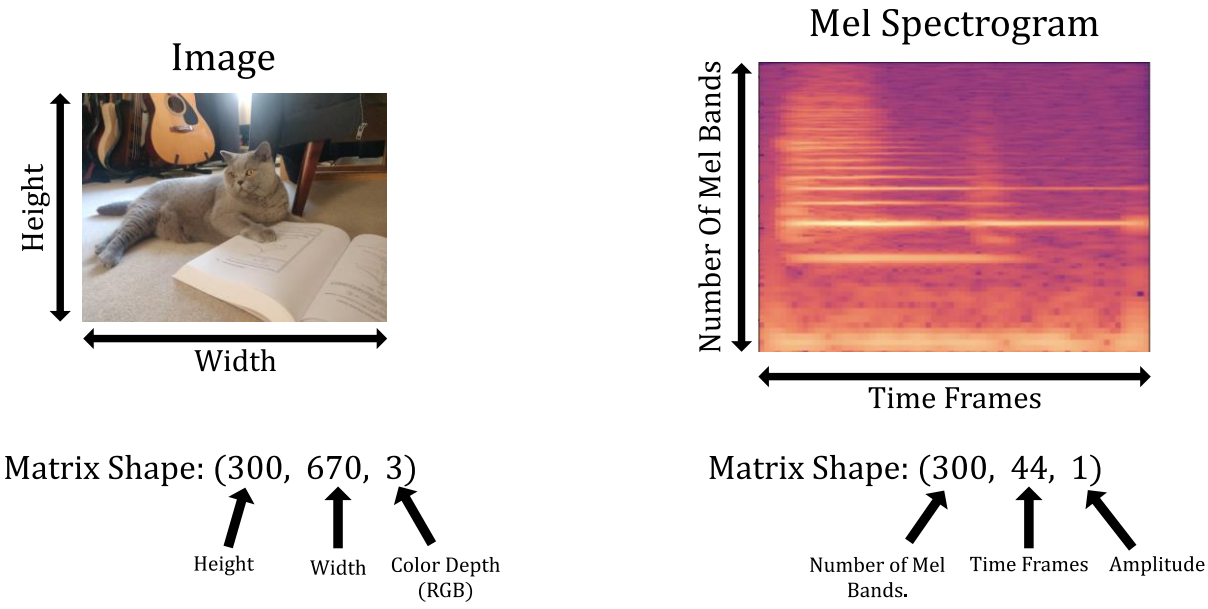


Figure 66. Comparison of image data, and Mel spectrogram.

The difference between the two is that the color depth requires three dimensions, while the amplitude is one dimension. This meant that the Mel spectrogram must be re-shaped before it can be used as an input with InceptionV3. With Numpy functionality, two additional dimensions were created for the amplitude, and since there can only be one dimension of amplitude measurements, the additional two dimensions were filled with zeros.

With the Mel spectrogram re-shaped, the data was passed into InceptionV3, and a layer was chosen to maximize. The result is a processed DeepDream effected Mel spectrogram.

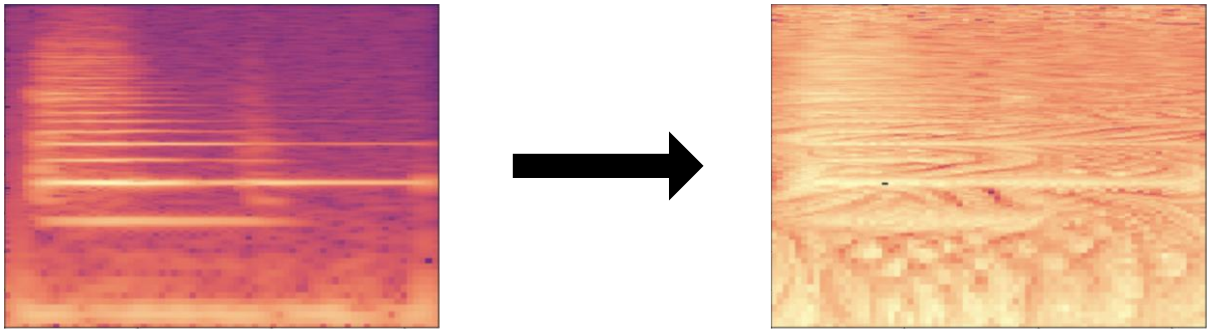


Figure 67. Before and of a Mel spectrogram processed using InceptionV3 trained with ImageNet.

The result was converted into audio, and sonified.

Soundcloud to proof-of-concept examples (Caution, loud audio).

4.2 A Duel Between Mel Spectrogram Encoders.

A python package called Kapre (Choi et al., 2017) was audited for its capabilities to encode raw audio into a Mel spectrogram. Unlike like Librosa, Kapre packaged the Mel spectrogram encoding process as a Keras layer, which gets computed on the graphical processing unit (GPU). This is appealing because this encoding layer can become a part of a model (Figure 68), thus making the input of a model raw audio, eliminating the need to encode the entire dataset before making a start on training models.

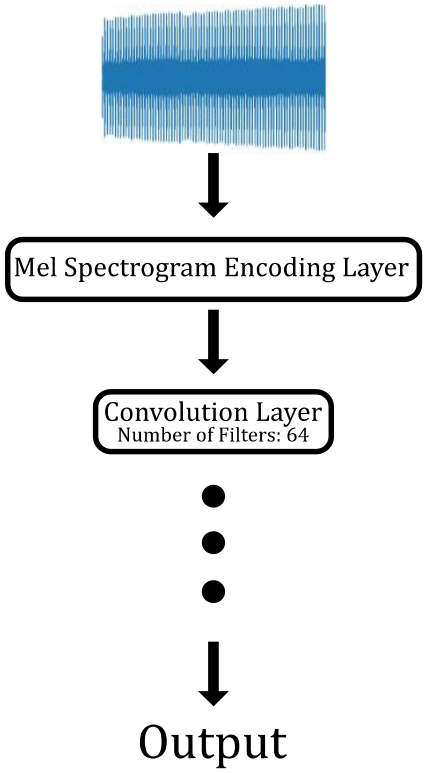


Figure 68. Example of a model with an encoding layer incorporated within a model.

A test comparing the quality of the encoding between Librosa and Kapre was conducted. For this test, all the parameters involved with the Mel spectrogram encoding, such as the length of the FFT window segment, were kept the same between the two encoders. The steps of this experiment are as follows:

1. Load an audio sample (in this case it was a sample of a cello playing out a note for one second).
2. Encode the audio sample into a Mel spectrogram using Librosa.
3. Encode the audio sample into a Mel spectrogram using Kapre.
4. Visualize both Mel spectrograms for comparison.
5. Convert both Mel spectrograms into audio using Librosa's Mel to audio converter.
6. Save converted audio.

The converted audio and spectrograms were then compared (Figure 69).

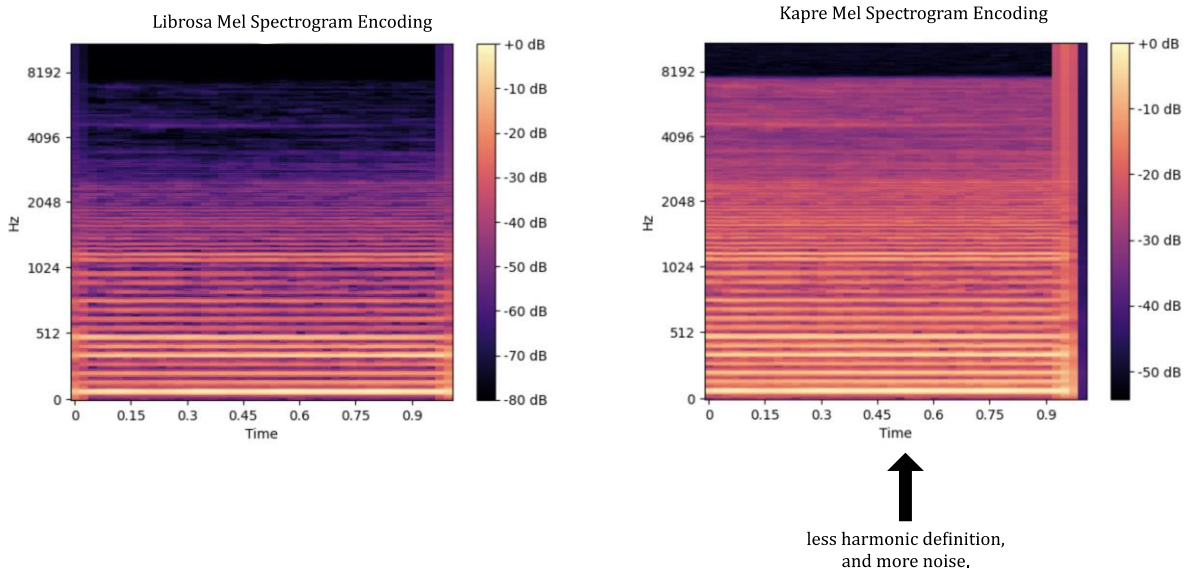


Figure 69. Mel spectrogram comparison between Librosa (left), and Kapre (right).

The resulting audio from the Kapre Mel spectrogram encoder featured unpleasant noise, which were particularly perceived in the upper-mid frequencies. On the other hand, the converted audio from the Librosa encoding closely resembled the original input audio. Therefore, it was decided that Kapre is not to be used.

4.3 Improving Data Loading Times.

The Librosa Mel spectrogram encoder was then implemented as part of the data generator to test the viability of the encoding happening during the training loop. This was implemented using multiple threads to leverage parallel encoders on the CPU. However, a desire to determine if loading already encoded Mel spectrograms would be more performant arose, when the time it took to train models was noticeably long.

Therefore, a test was conducted to time and compare encoding during the train loop, and loading already encoded date. For this , a small portion of the dataset was encoded and stored as Mel spectrograms. Then both scenarios were timed, each scenario involved loading 64 samples. The results were:

Loading raw audio followed by encoding $\approx$ 34 seconds
directly loading Mel spectrograms $\approx <$ 1 second

Given how significant the difference in performance was, it was decided that the data generator would load already encoded Mel spectrograms. Therefore, the entire dataset was encoded into Mel spectrograms, and stored to disk.

4.4 Unit Testing.

Some unit testing was carried out to ensure that the data samples, once loaded, match their label. These tests include:

1. Testing that one-hot-encoding representations of classes for given samples matches the actual classes those samples belong too.
2. Testing that the data frames contained an equal representation for all the classes.
3. Testing that the batches of samples coming from the data generator all match their classes.

5. Evaluation

5.1 Model Training Results.

Numerous runs of training were done with different model builders, and differently sized portions of the datasets. The first runs of training were done with a small portion of the dataset, these served as a test to find appropriate hyper-parameter bounds for the Bayesian optimization to search within.

Model Builder: bayesian_optimization_test_model
Number of Samples For Training: 20,000

model_ID	loss	accuracy
1	20.19195557	0.2358870953
58	2.538319349	0.1683467776
49	2.547129393	0.1668346822
57	2.504528999	0.1658266187
25	2.560255289	0.1653225869

Figure 70. Top five models from a test run of training. Highest accuracy ≈ 24%

One of the runs that was done used a large portion of the dataset.

Model Builder: build_conv2d_model
Number of Samples For Training: 493,989

model_ID	loss	accuracy
17	2.944287539	0.5093950033
1	5.456118584	0.4870499969
16	1.89609766	0.4697549939
11	1.941204071	0.4591499865
2	1.894245863	0.4586000144

Figure 71. Top five models from a training run with a large portion of the dataset. Highest accuracy ≈ 51%

All the models from Figure 71 are over-fitting, and are not achieving an ideal accuracy score. Larger models would have likely fit this amount of data better, but since the hyper-parameter bounds for the Bayesian optimizer were determined on a smaller amount of data, the optimizer did not seek the more optimal parameters. However, instead of adjusting the hyper-parameter bounds to accommodate the search space with more options to build larger models. it was decided that a smaller portion of the dataset is to be used. This is because this run of training took a very long time to complete, and creating even larger models would mean an even longer training duration.

With a smaller portion of the dataset, the same model builder performed significantly better (Figure 72).

Model Builder: build_conv2d_model
Number of Samples For Training: 30,000

model_ID	loss	accuracy
72	0.7641189098	0.8048878312
62	0.8215856552	0.7865585089
76	0.8448687792	0.784555316
78	0.8698278666	0.7781450152
75	0.8975596428	0.7710336447

Figure 72. Top five models from a training run with a large portion of the dataset. Highest accuracy ≈ 80%

The best model from this run of training was one of the models chosen to conduct filter visualization experiments. The model's architecture is illustrated in Figure 73. This model will be referred to as the "main model" going forward.

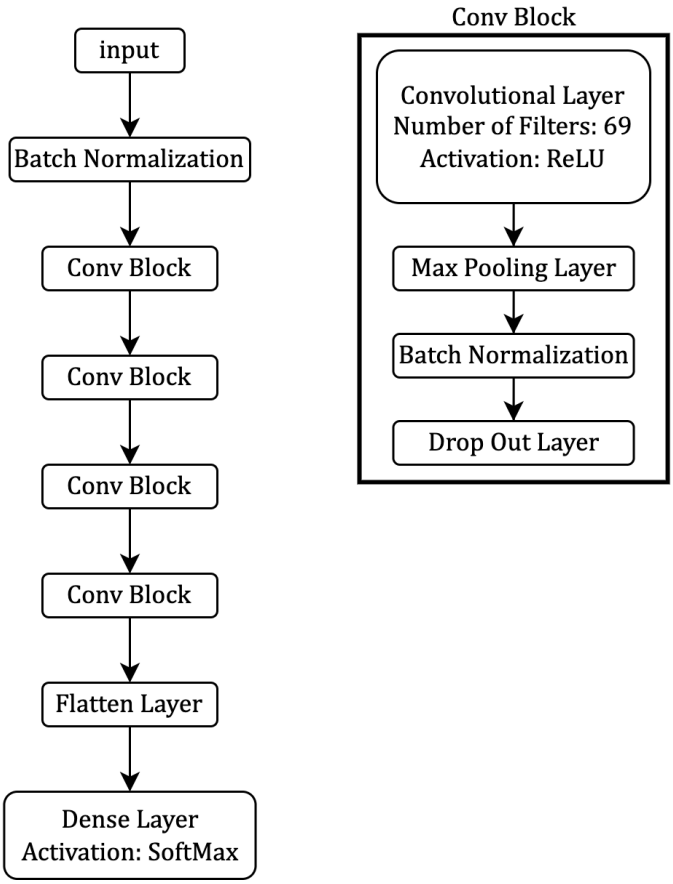


Figure 73. Architecture of the main model.

During testing, a run of training was done using the original Philharmonia dataset with its original ontology preserved. One of the models from this run was also selected because it achieved a high accuracy score (Figure 74). This model will be referred to as the Philharmonia model going forward.

Model Builder: vgg_like_model
Number of Samples For Training: 15,393

model_ID	loss	accuracy
3	0.1496812403	0.9489518404
0	0.2238350362	0.9390528202
6	0.2352690101	0.9250776172
1	0.2840955257	0.9221661687
5	0.3607382774	0.8765528202

Figure 74. Top five models from a test training run using Philharmonia dataset in its original form.

5.2 Sonified Results.

The two algorithms for processing a signal described in section 3.7 Amplifying Filter Activations. were employed in combination with the two models selected to process recordings of isolated instruments. For ease of comparison, all these examples used the same one second sample of a cello briefly bowing a note.

The sample was loaded, encoded, and passed through a portion of the filters from the first layer of the main model, and the Philharmonia model. The algorithm was iterated upon one time. These processed samples seem to introduce certain frequencies that are not in the original signal. There is also added noise to the signal. No discernible patters that would quantify an instruments timbre are noticeable though.

The DeepDream algorithm was next explored. Several layers from the main model, and the Philharmonia model were chosen for this. The algorithm was iterated up 50 times. The processed samples don't seem to offer as much harmonic frequencies as the individual filters do, but rather a lot more noise. Again, no discernible patters that would quantify an instruments timbre are noticeable.

Individual filters from the two models were chosen to process a long piece of audio. For this a recording of "Clair de Lune" (*Clair de Lune (Claude Debussy) by Beta Records*, n.d.) composed by Claude Debussy was chosen.

1. Main Model Calir de Lune.
2. Philharmonia Model Clair de Lune.

Both processed recordings introduce frequencies that are not found in original recording. The main model seems to be "cleaner" compared to the Philharmonia model.

The main model was also chosen to process a break beat (run for one iteration). Each filter here produces a different effect, which can all be described as a "glitchy" effect.

5.3 Mel spectrograms Are Not Images.

As stated previously in section 3.4 Encoding the Dataset., Mel spectrograms were chosen because they can represent audio data in a compress format, thus making the training process a lot quicker. However, 2D CNNs are not able to extract meaningful features of an instrument's timbre from Mel spectrograms the same way they are able to extract features from an image. This is because images and Mel spectrograms convey vastly different information, but the kernel from the filters in the CNN are extracting features from Mel spectrograms as if it is an image.

For instance, kernel's that move across an image are extracting various features such as lines, edges, or cat ears with no spacial awareness. This is fine for images because a cat's ear could be placed anywhere along the dimensions of an image. All that matters is that the kernel can extract the pattern that is associated with a cat's ear, and activate accordingly.

However, Mel spectrograms (of instruments) conveys a lot of information that requires spacial awareness. For example, the $y - axis$ (frequencies) in its entirety could be representative of an instrument's harmonics, which is a crucial determinant factor for identifying an instruments timbre. The $x - axis$ (time) combined with the amplitude is also crucial because it conveys information about the instrument's envelope (the amplitude structure). For example, the change in amplitude over time of a piano aggressively striking a note, and holding it down until it gently fades out is vastly different to a violin that gently swells in a note until it very abruptly stops.

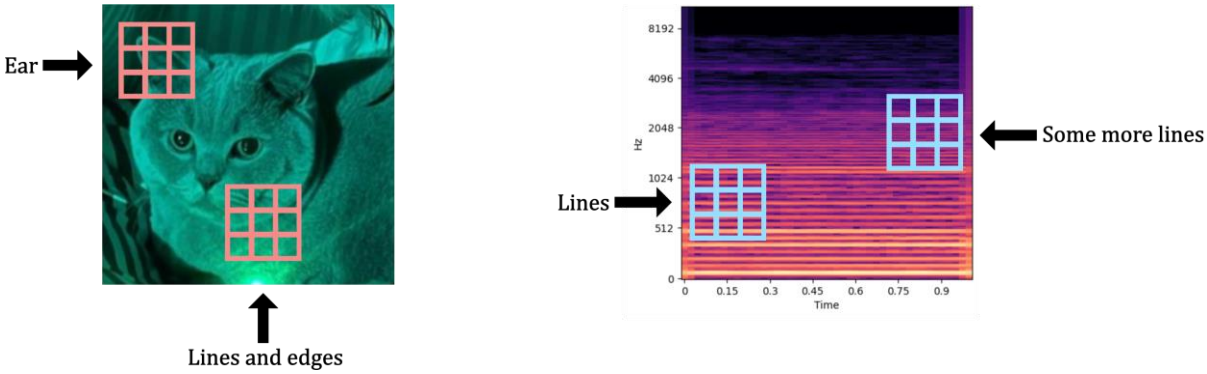


Figure 75. Illustration of how filters don't have spacial awareness, which is crucial for Mel spectrograms of instruments.

Performing the filter amplification algorithm on an input signal exposes how the features the filters learnt are lines as opposed to meaningful information about an instrument's timbre (Figure 76).

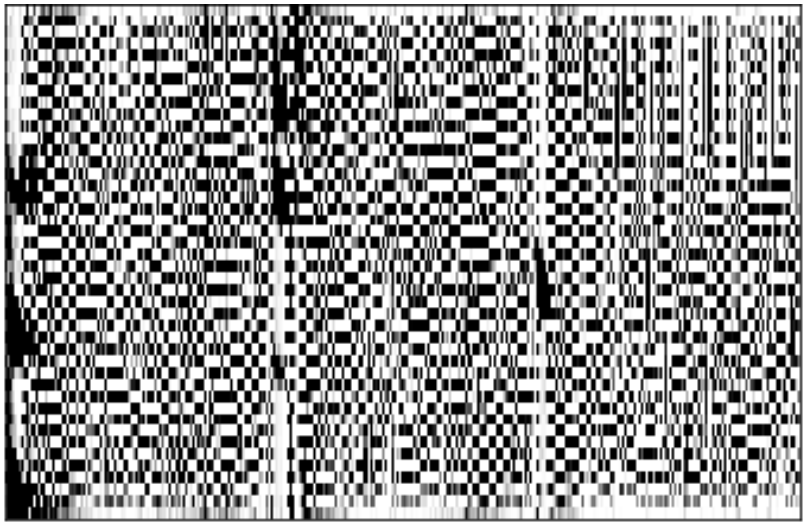


Figure 76. A processed sample of a break beat. Gradient ascent was run for 100 iterations on a filter form the first layer.

A kernel that extends across the entire frequency range could be a possible solution to this problem (Figure 77).

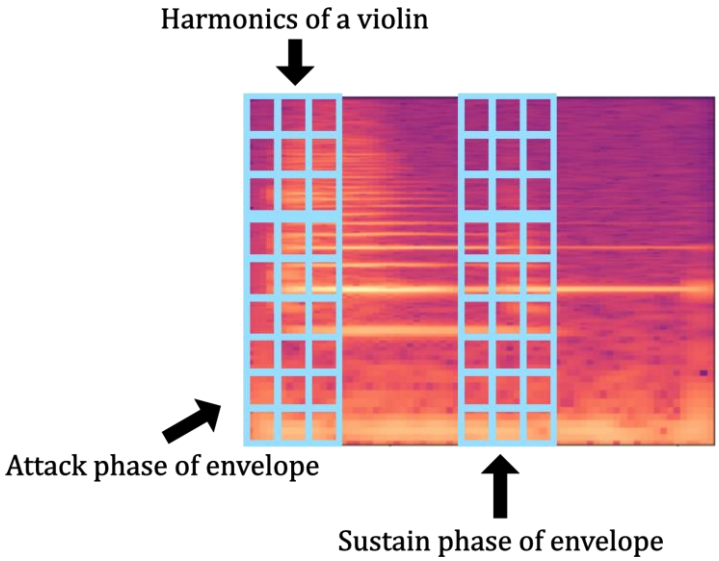


Figure 77. A hypothetical kernel that could be more representative of an instrument's timbre.

Another problem of using Mel spectrograms with CNNs is that max pooling layers shrink the frequency range, and time of the Mel spectrogram (Figure 78), resulting in a significant loss of information which our ears are sensitive too.

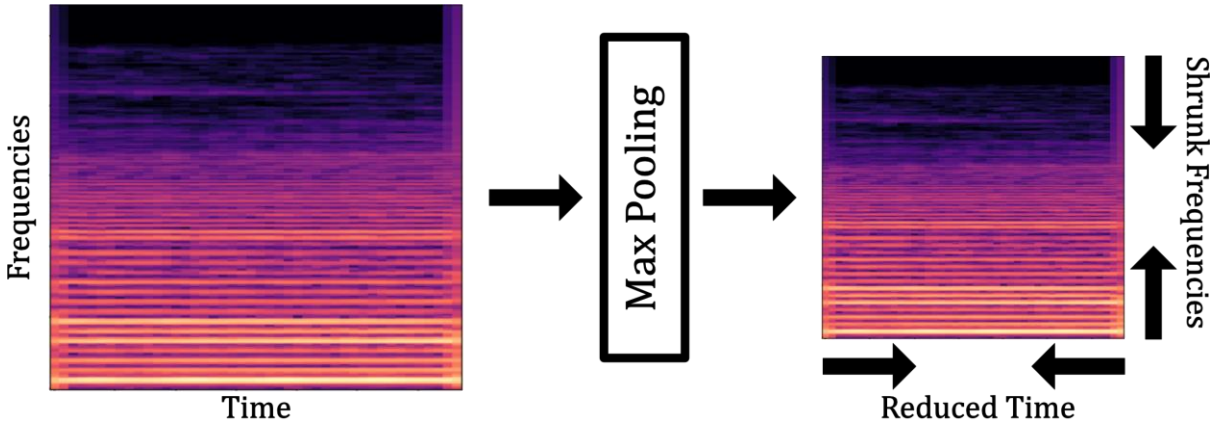


Figure 78. Visualization of the effects max pooling has on a Mel spectrogram.

Max pooling also does not have the effect of extracting every more abstract features in a hierarchical manner demonstrated in Figure 39. Instead, the extraction is very similar to having no max pooling, but at a lower fidelity.

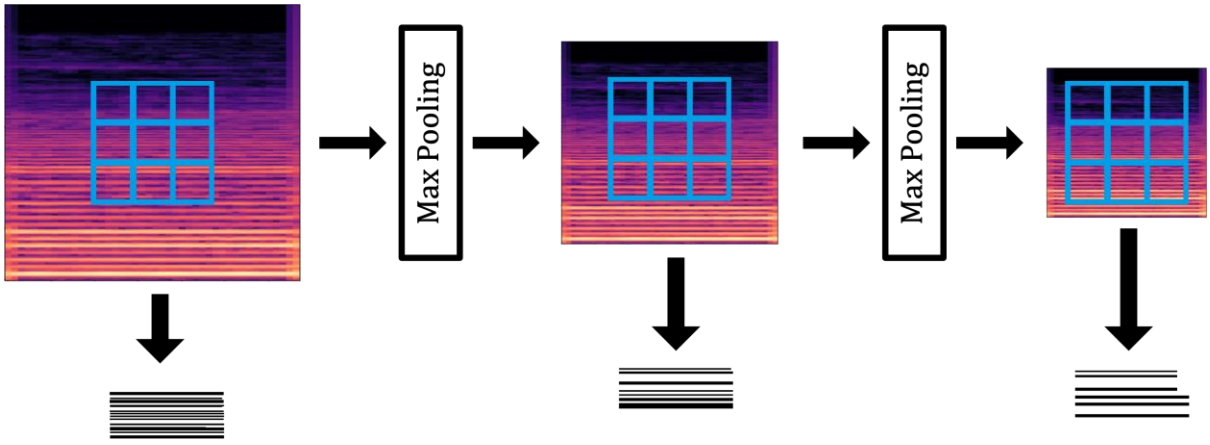


Figure 79. Visualization of how Max Pooling with Mel spectrograms doesn't elicit a hierarchical composition of elements to create more abstract patterns.

6. Conclusion

Throughout the development of this project, a variety of challenges have been realized in the domain of building AI systems for the purpose of audio processing. Some of these challenges include:

1. Finding appropriate dataset.

- When seeking out datasets, very few high-quality datasets were found of musical instruments in the public domain, and the few datasets that do exist contain fewer samples compared to image-based datasets that can be found.

2. Maintaining audio fidelity.
    - For AI systems to be viable audio effect processing units, there cannot be too significant a loss in audio fidelity. Effect units such as reverb, or delay are commonly used in a digital audio workstation (DAW), which are typically processing, and playing back audio at a set sample rate of 44.1kHz, or 48kHz. The model that was built in this project can only processes audio at the significantly reduced sample rate of 22.05kHz, and it is additionally encoded which further reduces the overall number of data points. Despite this significant reduction of data samples, performing the algorithms described in section 3.7 Amplifying Filter Activations. takes a very long time to compute. It is worth noting that these algorithms were implemented in python, which is typically considered a slower language compared to a language like C++, which is generally the programming language of choice for audio programming.

The work conducted in this project has shown how model feature visualization with Mel spectrograms hasn't yielded patterns that are recognizable in the same way the research conducted on images has elicited. However, further research can be conducted. It is of particular interest to explore the algorithms described in section 3.7 Amplifying Filter Activations. with models that are trained using raw audio directly. It is also of interest to explore other types of models that are better suited for learning from time-series data, such as long short-term memory (LSTM) models.

It is also of interest to explore alternatives to the max pooling operation, as to not remove samples from an audio signal. A variety of audio down sampling techniques that don't affect the dimensionality of a signal can be explored for this.

7. References

Akansu, A. N., & Haddad, R. A. (2001). Chapter 5—Time-Frequency Representations. In A. N. Akansu & R. A. Haddad (Eds.), *Multiresolution Signal Decomposition (Second Edition)* (Second Edition, pp. 331–390). Academic Press. https://doi.org/10.1016/B978-012047141-6/50005-7

Ardila, D., Resnick, C., Roberts, A., & Eck, D. (n.d.). *AUDIO DEEPDREAM: OPTIMIZING RAW AUDIO WITH CONVOLUTIONAL NETWORKS.* 3.

Bansal, M. (2020, August 31). Face recognition using Transfer learning and VGG16. *Analytics Vidhya*. https://medium.com/analytics-vidhya/face-recognition-using-transfer-learning-and-vgg16-cf4de57b9154

Choi, K., Joo, D., & Kim, J. (2017). Kapre: On-GPU Audio Preprocessing Layers for a Quick Implementation of Deep Neural Network Models with Keras. *Machine Learning for Music Discovery Workshop at 34th International Conference on Machine Learning*.

Chollet, F. (2018). *Deep Learning with Python*. Manning.

Chollet, F. & others. (2015). *Keras*. https://keras.io

Clair de Lune (Claude Debussy) by Beta Records. (n.d.). Retrieved 6 May 2022, from https://www.soundclick.com/music/songInfo.cfm?songID=14032150

Das, J. K., Chakrabarty, A., & Piran, Md. J. (n.d.). Environmental sound classification using convolution neural networks with different integrated loss functions. *Expert Systems*, *n/a*(n/a). https://doi.org/10.1111/exsy.12804

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 248–255.

Engel, J., Resnick, C., Roberts, A., Dieleman, S., Eck, D., Simonyan, K., & Norouzi, M. (2017). *Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders*.
Erickson, R. (1975). *Sound Structure in Music*. University of California Press.
G., A., & Ramakrishnan, A. G. (2007). *Music and Speech Analysis Using the 'Bach' Scale Filter-Bank*. https://doi.org/10.13140/RG.2.1.4752.9686

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.
Gupta, S. (2019, April 1). *Collaborative Filtering and Embeddings—Part 1*. Medium. https://towardsdatascience.com/collaborative-filtering-and-embeddings-part-1-63b00b9739ce

Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Río, J. F. del, Wiebe, M., Peterson, P., … Oliphant, T. E. (2020).

Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Kenworthy, A. (2020, October 21). *Improving black-box process efficiency using Bayesian Optimization—Balluff Blog*. https://www.innovating-automation.blog/bayesian-optimization/

Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., & Willing, C. (2016). Jupyter Notebooks – a publishing format for reproducible computational workflows. In F. Loizides & B. Schmidt (Eds.), *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (pp. 87–90). IOS Press.

Maklin, C. (2019, July 22). *Dropout Neural Network Layer In Keras Explained*. Medium. https://towardsdatascience.com/machine-learning-part-20-dropout-keras-layers-explained-8c9f6dc4c9ab

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Jia, Y., Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, … Xiaoqiang Zheng. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. https://www.tensorflow.org/

McFee, B., Metsai, A., McVicar, M., Balke, S., Thomé, C., Raffel, C., Zalkow, F., Malek, A., Dana, Lee, K., Nieto, O., Ellis, D., Mason, J., Battenberg, E., Seyfarth, S., Yamamoto, R., viktorandreevichmorozov, Choi, K., Moore, J., … Thassilo. (2022). *librosa/librosa: 0.9.1* (0.9.1) [Computer software]. Zenodo. https://doi.org/10.5281/zenodo.6097378

McKinney, W. (2010). Data Structures for Statistical Computing in Python. In S. van der Walt & J. Millman (Eds.), *Proceedings of the 9th Python in Science Conference* (pp. 56–61). https://doi.org/10.25080/Majora-92bf1922-00a

Nelson, D. (2020, January 10). What is Backpropagation? *Unite.AI*. https://www.unite.ai/what-is-backpropagation/

Nogueira, F. (2014). *Bayesian Optimization: Open source constrained global optimization tool for Python*. https://github.com/fmfn/BayesianOptimization

Olah, C., Mordvintsev, A., & Schubert, L. (2017). Feature Visualization. *Distill*, *2*(11), e7. https://doi.org/10.23915/distill.00007

Paralkar, K. (2020, April 15). Audio Data Augmentation in python. *Medium*. https://medium.com/@keur.plkar/audio-data-augmentation-in-python-a91600613e47

Pedersen, P. (1965). The Mel Scale. *Journal of Music Theory*, *9*(2), 295–308. https://doi.org/10.2307/843164

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., & others. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*(Oct), 2825–2830.

Philharmonia Orchestra. (n.d.). *Philharmonia—Sound Samples*.

Resemble AI. (2022, March 9). How Resemble AI Created Andy Warhol Docu-series Narration Using 3 Minutes of Original Voice Recordings. *Resemble AI*. https://www.resemble.ai/andy-warhol/

Salamon, J., Jacoby, C., & Bello, J. P. (2014). A Dataset and Taxonomy for Urban Sound Research. *Proceedings of the 22nd ACM International Conference on Multimedia*, 1041–1044. https://doi.org/10.1145/2647868.2655045

Sharma, A., Kumar, P., Maddukuri, V., Madamshettib, N., KG, K., Kavurub, S. S. S., Raman, B., & Roy, P. P. (2020). Fast Griffin Lim based Waveform Generation Strategy for Text-to-Speech Synthesis. *ArXiv:2007.05764 [Cs, Eess]*. http://arxiv.org/abs/2007.05764

Short-time FFT - MATLAB - MathWorks United Kingdom. (n.d.). Retrieved 26 April 2022,
from https://uk.mathworks.com/help/dsp/ref/dsp.stft.html

Simonyan, K., & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. *ArXiv:1409.1556 [Cs]*. http://arxiv.org/abs/1409.1556

Sinha, H., Awasthi, V., & Ajmera, P. K. (2020). Audio classification using braided convolutional neural networks. *IET Signal Processing*, *14*(7), 448–454. https://doi.org/10.1049/iet-spr.2019.0381

Sogomonian, G., Arakelyan, G., & er al. (2019). *An easy-to-use & supercharged open-source experiment tracker*. https://aimstack.readthedocs.io/en/stable/

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2014). Going Deeper with Convolutions. *ArXiv:1409.4842 [Cs]*. http://arxiv.org/abs/1409.4842

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2015). Rethinking the Inception Architecture for Computer Vision. *ArXiv:1512.00567 [Cs]*. http://arxiv.org/abs/1512.00567

Thoma, M. (n.d.). *Aurelia-aurita-3-0049.jpg (1280×960)*. Retrieved 6 May 2022, from https://upload.wikimedia.org/wikipedia/commons/6/65/Aurelia-aurita-3-0049.jpg

van den Oord, A., Dieleman, S., & Schrauwen, B. (2013). Deep content-based music recommendation. In C. J. Burges, L. Bottou, M. Welling, Z. Ghahramani, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems* (Vol. 26). Curran Associates, Inc.
https://proceedings.neurips.cc/paper/2013/file/b3ba8f1bee1238a2f37603d90b58898d-Paper.pdf

Warden, P. (2017, August 24). Launching the Speech Commands Dataset. *Google AI Blog*. http://ai.googleblog.com/2017/08/launching-speech-commands-dataset.html

Youssef, M. (n.d.). *Data Generators In Tensorflow*. Retrieved 28 April 2022, from https://mahmoudyusof.github.io/facial-keypoint-detection/data-generator/

8. Appendix

- Code repository GitLab: https://gitlab.doc.gold.ac.uk/aspit002/auditory-deepdream

- Soundcloud with more audio examples: https://soundcloud.com/user-151681972