1 The Neural Network

1.1 An Introduction to Neural Networks

The task of algorithmically matching sound files to musical instrument classes in exceedingly difficult through conventional computer programming techniques. Because of the complexity of sound waves, brain's interpretation them, the challenge arises as to how to build some sort of program that could function at a level above general procedural or explicit instructional rules. Rather than *hard-coding* a set of conditions or parameters for a classification program, we seek an architecture that allows for a computer to *learn* and change and update itself as it is presented with more data. Such an algorithm exists in the form of a *Neural Network* [2, 3, 6].

A neural network is a mathematical function that seeks to produce an approximation F^*, of some usually unknown function F . For audio recognition, F is some function, or series of functions, that occurs in the brain which allows for a listener to take the sensation of audio, and map it to a known musical instrument. For a neural network, a model must construct an approximate function F^* using a set of parameters Θ, which allows the model to map a series of inputs, x (called *features*) to a series of outputs, y (called *predictions*) [3, 4, 16].

Former YouTube video classification team lead, and current machine learning consultant, Aurelien Geron writes about the relationship between biological brains and mathematical neural networks [1]:

> Birds inspired us to fly, burdock plants inspired velcro and nature has inspired many other inventions. It seems only logical, then, to look to the brain's architecture for inspiration on how to build an intelligent machine.

The result of such an analogy is a computer program that is reminiscent of the brain.

Over the course of their lives, humans will learn to map sounds to sources almost effortlessly. Given an example, and the appropriate label, humans can do this with very reasonable accuracy over a wide array of sounds [12, 17]. We can simplify the idea of humans recognizing sound as some function or operation that occurs within the brain, accepting an input sound, and produces an output label. Similarly, a neural network can be constructed, presented with *features* of multiple, labeled sound waves and learns a set of parameters Θ that allows for the mapping to a source.

For this particular project, a neural network has been constructed to perform a *classification* task. A classification task involves mapping the input data, x to one of k possible *classes*, each represented by an integer, 0 through $k - 1$. Each of the k classes represents a particular musical instrument that could have produced the sound-wave in the audio file.

1.2 The Structure of a Neural Network

A Neural Network is simply a model of a mathematical function, composed of several smaller mathematical functions called *layers* [3, 8]. Each layer represents an operation that takes some real input, typically an array of real double-precision floating-point numbers, and returns a modified array of new double-precision floating-point numbers. The exact nature of this operation can be very different depending on the layer type or where it sits within the network. It is this process of transforming inputs successively in a particular order until an output is attained [1, 8]. This output encodes the models final "decision" given a unique input.

Other than inspiration from the brain, Neural Networks are name *networks* because of their nested composition nature. The model can be represented by a linear or acyclic computational graph which successively maps exactly how the repeated composition is structured. In a *feed-forward network*, information is passes successively in one direction. For example functions could be chained together such as [3]

$$F(x) = f^{(L-1)}\big[f^{(L-2)}\big[...f^{(1)}\big[f^{(0)}\big[x\big]\big]\big]...\big] \tag{1}$$

Where each function $f^{(l)}$ represents a layer of the network model. The number of layers in a neural network is referred to as the network *depth*. The dimensionality of each layer is referred to as the network *width* [1, 8].

A network model that contains L unique layers is said to be an L-Layer Neural Network, with each layer usually indexed by a superscipt, 0 through $L-1$. Layer 0 is said to be the *input layer* and layer $L-1$ is said to be the *output layer*. The function that represents a layer (l) is given by

$$f^{(l)} : x \in \mathbb{R} \rightarrow y \in \mathbb{R} \tag{2}$$

The value of x can also be index by layer, we call the array $x^{(l)}$ the *activations* of layer l [3, 8].

The model is recursive by nature, the activations from one layer, $l-1$, are used to directly produce the activations of the next successive layer l. Thus eqn. (2) can be alternatively written as:

$$f^{(l)} : x^{(l-1)} \in \mathbb{R} \rightarrow x^{(l)} \in \mathbb{R} \tag{3}$$

The array of activations, $x^{(0)}$, is the raw input given the neural network, most commonly called *features*. Conversely, the activations $x^{(L-1)}$ are commonly called the network *output* [1, 4, 8].

Algorithm 1 Forward propagation system in a standard deep neural network. Each iteration in the main *for-loop* represents the execution of a layer, and passing the result to the "next" layer function. A practical application of this algorithm should include batches of samples instead of a single sample. I will put this some place better soon! It is here for future placement

Require: Network with L layers - $0, 1, 2, ..., L-2, L-1$
Require: Set of weighting parameters - $W^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of bias parameters - $b^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of layer activation function - $a^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Input sample - $x^{(0)}$
  $Z = \{\} \leftarrow$ array to hold pre-activations
  $X = \{\} \leftarrow$ array to hold final activations
  for $l = 1, 2, ..., L-2, L-1$ **do**
    Compute pre-nonlinear activation
    $z^{(l)} \leftarrow W^{(l)} x^{(l-1)} + b^{(l)}$
    Apply activation function
    $x^{(l)} \leftarrow a^{(l)}[z^{(l)}]$
    Store values for later use in optimizer
    $Z \leftarrow Z.add(z^{(l)})$
    $X \leftarrow X.add(x^{(l)})$
  end for
  Network final prediction is last object in 'act' array.
  $y^* \leftarrow x^{(L-1)} = X[-1]$
  Return the pre-nonlinear activations and final-nonlinear activations
  return Z, X

1.3 Layers Used in Classification Neural Network

As stated previously, a neural network is composed of a series of functions that are called successively to transform features (an input) into a prediction (an output). As shown in eqn. (3), each function feeds directly into the next as to form a sort of computational graph [3].

Typically, a layer function can be divided into two portions: (i.) a Linear transformation, with a bias addition, and (ii.) an element-wise activation transformation. This activation function typically is a non-linear function which allows for the modeling of increasingly complex decision-boundaries. Step (i.) is usually in the form of a matrix-vector equation:

$$z^{(l)} = W^{(l)} x^{(l-1)} + b^{(l)} \tag{4}$$

Where $W^{(l)}$ is the *weighting-matrix* for layer l, $b^{(l)}$ is the *bias-vector* for layer l, $z^{(l)}$ are the *linear-activations* for layer l and $x^{(l-1)}$ is the final activations for layer $l-1$. Step (ii.) is usually given by some *activation function*:

$$x^{(l)} = \sigma^{(l)}\left[z^{(l)}\right] \tag{5}$$

Where $x^{(l)}$ is the final activations for layer l and $z^{(l)}$ is given in equation (4). $\sigma^{(l)}$ is some activation function, which is often use to enable the modeling of more complex-decision boundaries.

The combination of eqn. (4) and eqn. (5), for a layer l , make up the function represented by that layer, as in eqn. (3). Below, we discuss and describe the types of layer functions that are used to produce the classification model in this project.

1.3.1 Dense Layer

The Linear Dense Layer, often just called a *Dense* Layer, or *Fully-Connected* layer for short, was one of the earliest function types used in artificial neural network models [2, 8]. They are among the most commonly used layer types. A dense layer is composed of a layer of *artifical neurons*, each of which holds a numerical value within it, called the *activation* of that neuron. This idea was developed back from McCulloch and Pitts' work [9], and was expanded upon by Frank Rosenblatt in 1957 [1].

We model a layer of neurons as a vector of floating-point numbers. Typically, it is required that the array be one-dimensional. Suppose a layer (l) contains n artificial neurons. We denote the array that hold those activations as $x^{(l)}$ and is given by:

$$\vec{x}^{(l)} = \Big[x_0, x_1, x_2, \ldots, x_{n-2}, x_{n-1}\Big]^T \tag{6}$$

The activation of each entry is given by a linear-combination of activations from the previous layer, as outlined in eqn.(4) and eqn.(5).

Suppose the layer $l-1$ contains m neurons. Then the weighting-matrix, $W^{(l)}$ has shape $m \times n$, the bias-vector $b^{(l)}$ has shape $m \times 1$. We can denote the function in a similar manner to eqn.(3) as:

$$f^{(l)} : x^{(l-1)} \in \mathbb{R}^{(m\times 1)} \rightarrow x^{(l)} \in \mathbb{R}^{(n\times 1)} \tag{7}$$

Thus for a dense layer l, the exact values of each activation is given by [1, 8]:

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}^{(l)} = \sigma^{(l)} \left(\begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,m-1} \\ w_{1,0} & w_{1,1} & \dots & w_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n-1,0} & w_{n-1,1} & \dots & w_{n-1,m-1} \end{bmatrix}^{(l)} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-1} \end{bmatrix}^{(l-1)} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}^{(l)} \right) \tag{8}$$

or more compactly:

$$x^{(l)} = \sigma^{(l)} \Big(W^{(l)} x^{(l-1)} + b^{(l)} \Big) \tag{9}$$

Eqn. (9) is generally referred to as the *dense layer feed-forward equation* [3].

1.3.2 2-Dimensional Convolution Layer

Convolution layers emerged from studying the brain's visual cortex, and have been used from image recognition related tasks for around 40 years. [1, 8]). This layer get's it's name from it's implementation of the mathematical *convolution* function. The 2D discrete-convolution of function or 2D array $A[i, j]$ and $B[i, j]$ is given by [3]:

$$C[i, j] = (A * B)[i, j] = \sum_u \sum_v A[i, j]B[i - u, j - v] \tag{10}$$

Note that convolution is commutative: $(A * B) = (B * A)$. We implement a convolutional layer $f^{(l)}$), by creating a series of K weighting matrices, called *filters*, each of size $m \times n$, where $m, n \in \mathbb{Z}$. Often we choose $m = n$ and call the shape the *convolution kernel size* [8, 3].

In the case of a 2D input array, $x^{(l-1)} \in \mathbb{R}^{(N\times M)}$, the convolutions filters "step" through the input data and repeatedly compute the element-wise product of each $m \times n$ weighting kernel, and the local activation of the $x^{(l-1)}$ array. The step size in each of the 2-dimension is known as the *stride*. Each of the K filters is used to generate a $m \times n$ feature map from the input activation. For an appropriately trained network, some filters may be dedicated to detecting vertical lines, horizontal lines, sharp edges, areas of mostly one color, etc. [8, 1].

In general, for an $N \times M$ input, with kernel size $m \times n$, with stride size 1×1, and K feature maps.

$$f^{(l)} : x^{(l-1)} \in \mathbb{R}^{(N\times M)} \rightarrow x^{(l)} \in \mathbb{R}^{([N-n+1]\times[M-m+1]\times K)} \tag{11}$$

As an example, consider a single filter map over input $x \in \mathbb{R}^{(4\times 3)}$ and filter map $W \in \mathbb{R}^{(2\times 2)}$ as shown in fig. (**??**):

a	b	c	d
e	f	g	h
i	j	k	l

(a) Input Activations

w	x
y	z

(b) Convolution Filter

aw + bx + ey + fz	bw + cx + fy + gz	cw + dx + gy + hz
ew + fx + iy + jz	fw + gx + jy + kz	gw + hx + ky + lz

(c) Convolution Result

Figure 1: The result of convolving an input (a) with an filter map (b) is a new set of activations (c). This Image was adapted from Goodfellow, pg. 325 [3]

Note that formal implementations include a bias vector and an activation function.

For 2D convolution over input $x^{(l-1)}$ with feature map $W_k^{(l)}$, producing activations $x^{(l)}$, we compute the activations $x_k^{(l)}$ as [3]

$$x_k^{(l)} = \sigma^{(l)}\left[\left(\sum_{u=0}^{m-1}\sum_{v=0}^{n-1} x^{(l-1)}(i,j)W_k^{(l)}(i-u,j-v)\right) + b_k^{(l)}\right] \tag{12}$$

Please check my math here! This operation repeats for each of the K feature maps. Each maps has it's own $n \times m$ weighting matrix and appropriately shaped bias .

The convolution layers allows for several advantages. Among these are (i) *sparse-connectivity*: not every single activation (pixel) is connect to every single output pixel, so it is more computationally efficient, (ii) *positional invariance*: key features can be identified regardless of where they are in the layer

1.3.3 2-Dimensional Maximum Pooling Layer

A Maximum Pooling layers simply returns the maximum neuron activation in a group of neurons. In the case of 2D Max Pooling, we choose a kernel size to be $m \times n$, just like in the 2D Convolution layer, and extract the maximum value in each window, and then step along accrdoing to the chosen stride size [8, 3].As an example, consider again figure (**??**). Using the 2×2 kernel on the 3×4 input, each box would then contain the maximum value of the input activations. We detail this in fig. (**??**):

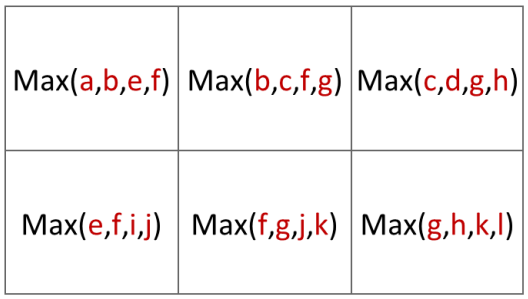

Figure 2: The result of 2D maximum-pooling an input array. This image was adapted from Loy, pg. 126 [8]

Pooling layers, such as maximum pooling, average pooling, or similar are typically placed after a layer or a set of layers of convolution. The purpose of this arrangement is to reduce the number of weights, thereby dropping the complexity of the model and ensuring only features with large activation values are preserved [1, 8, 3].

1.3.4 1-Dimensional Flattening Layer

A flattening layer is used to compress an array with two or more dimensions down into a single dimension. For a flattening layer l, multidimensional activations in layer $l-1$ are compressed down into a single dimensional array. We can use function notation to express this as:

$$f^{(l)} : x^{(l-1)} \in \mathbb{R}^{(M \times N \times ....)} \rightarrow x^{(l)} \in \mathbb{R}^{(MN... \times 1)} \tag{13}$$

The numerical value of each activation is left unchanged. For a layer with activation shape$N \times M$, the resulting activations are reshaped into $NM \times 1$ as shown in eqn (13).

Flattening Layer are most commonly used to prepare activations for entry into dense layer or series of dense layers. For example, 2D or 3D images are typically processed with 2D convolution, which may output a 2D or 3D array of activations. Those values are then flattened to 1 dimensions, which can then be passed into dense layers for further processing.

1.3.5 1-Dimensional Concatenation Layer

The 1-Dimensional Concatenation Layer, also called 1D-Concat layer takes separate vectors of activations

1.4 Activation Functions Used in Network Layers

Activation functions are a key parameter in the behavior of neural networks. As discussed in sec. (1.5) a layer function, $f^{(l)}$ is generally composed of a linear transformation, as in eqn. (4), and then an element-wise activation function as in eqn. (5). It is this second step that allows for neural networks to model the incredibly complex decision boundaries that are found in real-world problems [1, 8]. In this section, we detail the activation functions used in out classification neural network.

1.4.1 Rectified Linear Unit

The Rectified Linear Unit (ReLU) activation function acts element-wise on the activations in a given layer. If the activation of a neurons is non-zero or non-negative, the value is untouched, otherwise a 0 is returned. For an input activation x, ReLU is defined by:

$$\text{ReLU}\big[x\big] = \max\big(0, x\big) = \begin{cases} x & \text{if} \quad x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

ReLU is only defined on a real domain. We provide a visualization of the function in fig. (**??**).

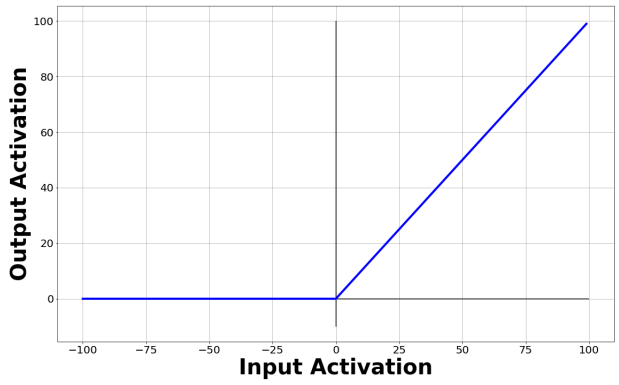

Figure 3: Rectifed Linear Unit (ReLU) activation function

1.4.2 Softmax

1.5 Training a Neural Network

A neural network's purpose is to produce a function F^* that approximates an unknown function F, using a set of parameters, Θ. The model must have a procedure for updating the parameters in Θ to allow for a reasonably close approximation of F. To better understand this, we turn to Tom Mitchell's explanation of a learning algorithm [3, 11]:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if it's performance at tasks in T, as measured by P, improves with experience E.

Without any human intervention, a model must update itself to improve it's performance at a give task a new information is presented to it. To do this, the model must be constructed with a training procedure in mind.

We consider the set of parameters Θ as a concatenation of each applicable layer's weighting matrix and bias vector such that:

$$\Theta = \left\{W^{(0)}, b^{(0)}, W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, ..., W^{(L-2)}, b^{(L-2)}, W^{(L)}, b^{(L-1)}, \right\} \tag{15}$$

Each parameter is floating-point number, which is stored in a weight matrix $W^{(l)}$ or bias vector $b^{(l)}$. Each value for each parameter contributes to the output of the function. Note that $W^{(l)}$ or $b^{(l)}$, itself is a parameter to train, but rather the entries that make up those structures must be trained. In increasingly complicated neural networks, there can be upwards of hundreds of thousands,or even millions of elements in θ making a neural network a functions that exist is a very high dimensional parameter-space [1, 3].

1.5.1 The Cost Function

Suppose we pass a training sample, given by the feature-vector $x^{(0)}$, with an expected outcome given by the vector y, into the neural network. After the network finishes processing, it's output activations are given by the vector $x^{(L-1)}$, also noted at y^*. For a reasonably trained model, we expect the vector y^* to be "similar" to y, indicating that with the provided data, the network has made a valid prediction. Conversely, for an untrained network, y^* is not likely to be "similar" to y at all, indicating that the network has made a poor prediction.

To quantify the difference between the model's prediction, y^* and the expected output, y, we introduce a *cost function*, $J(y^*, y)$, to the network [3, 4]. The cost function, also called a *loss* or *objective* function, compares the neural networks output, y^* and the expected output y. It returns a single value, usually a floating-point number which measures the *poorness* of the prediction. A high cost value indicates a *poor* prediction, and a low cost indicates a reasonable prediction.

$J(y^*, y)$ can take many forms and is often dependent on particular task or data set provided [4]. For this k-classes classification task, we choose to use a *Categorical Crossentropy* (CXE) cost function. For a single predictions sample, it is defined [3]:

$$\mathrm{CXE}\big[y^*, y\big] = -\sum_{i=0}^{k-1} y_i^* \log_{10}(y_i) \tag{16}$$

In practice, this operation is performed over a batch of samples and an average is computed and used a *batch loss* [3].

1.5.2 Gradient Based Learning

We have developed a method for quantifying the difference between a given output y^* and an expected output y with the inclusion of a cost function. For a trained neural network, we expect that samples consistently produce a low cost value, particularly on previous unseen inputs, which indicates that the model has appropriately *generalized*. The process of training the model is to choose the parameters in Θ that allows for this behavior of the cost function. We then treat the training process of a neural network as a higher dimensional *optimization* problem.

A neural network uses *indirect optimization*, which contrasts from pure optimization. Deep Learning expert Ian Goodfellow described this [3]:

> In most learning scenarios, we care about some performance measure P,... We reduce a different cost function, $J(\theta)$ in the hope that doing so will also imprvce P.

By choosing an appropriate cost function, and selecting the parameters Θ that minimize the average cost over a data set, we also assume that doing so minimized the error in the performance P.

The cost of a sample is dependent on training labels y and the network output y^*. The output is given by the final layer activation $x^{(L-1)}$, which in turn are produced by the previous layer and so forth, as demonstrated in eqn. (3), This recursive nature combined with the dimensionality of the parameter object Θ makes an analytical solution to the optimization impractical [1, 3, 4]. We instead optimize with a gradient-based method.

We can reduce the cost given a set of paramaters, $J(\Theta)$, by moving each element in Θ with small steps in the direction of the negative partial derivative of each parameter. In the case of the high-dimensional Θ object, we compute the partial derivative with respect to each weight and bias in the form of the *gradient vector*. We denote the gradient of J with respect to the parameters in Θ as:

$$\nabla_\Theta\big[J\big] = \Big[\frac{\partial J}{\partial W^{(0)}}, \frac{\partial J}{\partial b^{(0)}}, \frac{\partial J}{\partial W^{(1)}}, \frac{\partial J}{\partial b^{(1)}}, \ldots, \frac{\partial J}{\partial W^{(L-1)}}, \frac{\partial J}{\partial b^{(L-1)}}\Big]^T \tag{17}$$

Where $\frac{\partial J}{\partial W^{(l)}}$ is taking the partial derivative of each element in the $W^{(l)}$ matrix, and preserves the shape.

Due to the nested composition of the network output in eqn. (1), and subsequently the cost itself, we must use the chain rule of calculus to work backwards through the neural network to compute the partial derivative with respect to each parameter in Θ. The process of working backwards to compute each element in the gradient vector is called *back-propagation* [1]. Below we detail an algorithm in pseudo-code to back-propagate through a fully-connected nueral network, such as the multilayer perceptron outlines in sec.(1.2)

Algorithm 2 Backwards propagation system, in a standard densely connected deep neural network. Each iteration in the *for-loop* computes the gradient of the cost function J with respect to the weight and bias arrays. Each element in those arrays is then the discrete gradient of that parameter. A practical application of this algorithm should include batches of samples instead of a single sample

Require: Cost/Objective function J and learning rate α
Require: Set of weighting parameters - $W^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of bias parameters - $b^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of layer activation function - $a^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of layer activation function derivatives - $\partial a^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of pre-nonlinear activation - $Z^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of post-nonlinear activation - $X^{(i)}, i \in \{0, 1, ..., L-1\}$
  Execute forward pass in algorithm (1) and compute the gradient of the cost with respect to the final layer activations
  $dx \leftarrow \nabla_{(y^*)} J(y, y^*)$
  Intialize ∇J as output object, should have same shape as Θ
  for $L-1, L-2, ..., 2, 1$ **do**
    Compute gradient w.r.t pre-nonlinear activation portion of layer function
    $dx^{(l)} \leftarrow \nabla_{Z^{(l)}} J = dx^{(l)} \odot \partial a^{(l)}[Z^{(l)}]$
    Compute gradient w.r.t weighting and bias elements
    $db \leftarrow \nabla_{b^{(l)}} J = dx^{(l)}$
    $dW \leftarrow \nabla_{W^{(l)}} J = dx^{(l)} \cdot X^{(l-1)}$
    Add db and dW steps to ∇J object
    $\nabla J = \nabla J.Add(dW, db)$
  end for
  Return gradient w.r.t to each parameter in Θ
  return ∇J

After (i) computing the gradient, we can scale is by a desired learning rate, α and (ii) add it element-wise to the exisitint elements in Θ. By repeating steps (i) and (ii) in sucession, we gradually drive the cost function to produces consistently lower and lower values. This is called *gradient descent* and is the basis for many optimization algorithms. We show the

general update rule on iteration (i) for gradient based learning over a batch of m samples in eqn. (18).

$$\Theta^{(i)} = \Theta^{(i-1)} + (-\alpha)\frac{1}{m}\nabla_\Theta\Big[\sum_{n=1}^{m} J(y^{*(n)}, y^{(n)})\Big] \tag{18}$$

1.5.3 The Optimizer

For this project, we employ an *Adaptive-Moments* optimization algorithm, also called *ADAM* for short. It is a powerful variant of algorithms with an adaptive learning rate. It tracks an exponentially decaying average of past gradients, as well as exponentially decaying average of past squared gradients [1]. This produces a far more aggressive optimizer at a higher computational cost. For a given iteration (i), The ADAM update is given:

$$\begin{aligned}
s^{(i)} &= \rho_1 s^{(i-1)} + (1-\rho_1)\frac{1}{m}\nabla_\Theta\Big[\sum_{n=1}^{m} J(y^{*(n)}, y^{(n)})\Big] \\
r^{(i)} &= \rho_2 r^{(i-1)} + (1-\rho_2)\frac{1}{m}\nabla_\Theta\Big[\sum_{n=1}^{m} J(y^{*(n)}, y^{(n)})\Big] \odot \frac{1}{m}\nabla_\Theta\Big[\sum_{n=1}^{m} J(y^{*(n)}, y^{(n)})\Big] \\
s'^{(i)} &= \frac{s^{(i)}}{1-\rho_1^t} \\
r'^{(i)} &= \frac{r^{(i)}}{1-\rho_2^t} \\
\Theta^{(i)} &= \Theta^{(i-1)} + (-\alpha)\frac{s'^{(i)}}{\sqrt{r'^{(i)}} + \delta}
\end{aligned} \tag{19}$$

ADAM has experimentally shown to be a very powerful and robust optimizer useful for a wide range of tasks. Because of the two decay constants ρ_1 and ρ_2, we an compound and accumulate the values of past gradients to continue to push the cost to lower and lower values, even if the magnitude of the gradient becomes very small [1].

Algorithm 3 Adaptive-Moments (ADAM) optimizer for a neural network

Require: Step size α

Require: Small constant δ for numerical stabilization, usually about 10^{-7}.

Require: constants ρ_1, ρ_2 used got exponential decay rates, usually 0.9 and 0.999 respectively.

Require: Subroutine/function to compute gradient of cost function.

Require: Mini-batch size, m

Require: Stopping criterion S

Initialize moment variables and iteration counter $s = 0$, $r = 0$, $i = 0$

while Stopping Criterion S is **false do**

Extract a mini-batch of m samples from larger data set X. $[x^{(0)}, x^{(1)}, ..., x^{(m-1)}]$ and corresponding target values $[y^{(0)}, y^{(1)}, ..., y^{(m-1)}]$.

Compute numerical gradient estimate of each sample in batch. This can be done with standard back-propagation in algorithm (2) and normalize by batch size m:

$\nabla J \leftarrow \frac{1}{m}\nabla_\Theta \Big[\sum_{n=1}^{m} J(y^{*(n)}, y^{(n)}) \Big]$

Compute first bias moment: $s \leftarrow \rho_1 s + (1 - \rho_1)\nabla J$

Compute second bias moment: $r \leftarrow \rho_2 s + (1 - \rho_2)\nabla J \odot \nabla J$

First bias correction: $s' \leftarrow \frac{s}{1-\rho_1^i}$

Second bias correction: $r' \leftarrow \frac{r}{1-\rho_2^i}$

Compute And Apply update: $\Delta\Theta = (-\alpha)\frac{s'}{\sqrt{r'}+\delta}$

$\Theta = \Theta + \Delta\Theta$

Update Iteration number: $i \leftarrow i + 1$

end while

1.6 Chosen Model Architecture

The success of a neural network algorithm is enormously dependent of the strength of the chosen features, rather than the quantity of available data [1, 5, 7]. We have derived an appropriate set of inputs arrays See Section on Features, and constructed a neural network architecture to compliment these features. From these two input arrays, one being a matrix and the other being a column-vector, we have produced a *multimodal neural network*, that process the two inputs, and combines separate neural networks into a single model to produce a one prediction resulting from both inputs.

The full classification neural network used for this project consists of two distinct entry points. Rather than presenting the network with one set of input data, X, we present the network with two different arrays, X_1 and X_2. Both arrays are a product of the same audio file sound wave, and thus share a common training label, y. Each of the two entry layers leads to it's own branch and the given information is process roughly in parallel. Once a certain number of layer functions have been applied, each arm of the neural network is left with a single dense layer of activations. These two activation vectors are concatenated into a single new dense layer, and then passed through one final layer to generate the model's final output. A visual representation of this architecture can be found in fig. (**??**).

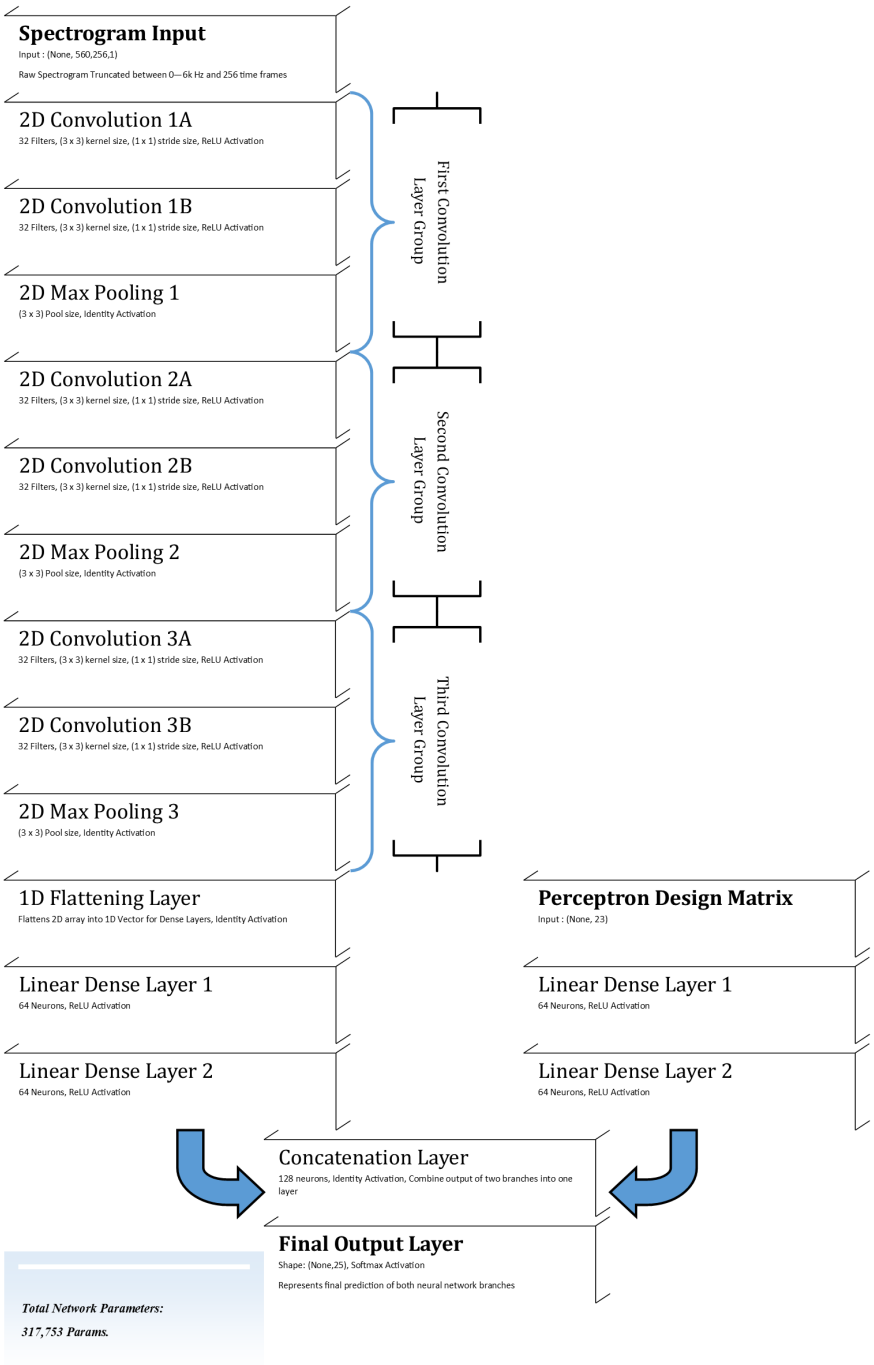

Figure 4: The developed architecture of the audio file classification neural network. The Left branch process an image-like input, the right branch processes a vector-like input. The activations are then merged, and then a single output is produced

1.6.1 The Spectrogram Branch

The spectrogram branch is pictured on the left side of fig. (**??**). A spectrogram is a representation of a sound wave, or signal by representing energy distributions as a function of *time* and *frequency* [17, 12, 5]. Details on the creating of the spectrogram can be found in the *features* section of this document.

The input layer of the spectrogram accepts a 4-Dimensional array. The axes, in order of indexing, represents (i) the size of the *mini-batch* of samples, (ii) the pixel width of each sample, (iii) the pixel height of each sample, and (iv) the number of channels in each sample. As a model hyper-parameter, we have chosen each batch to contain 64 samples. We have also chosen to truncate time and frequency axes (ii & iii) to contain 560 frequency bins, and 256 time-frames. Each image also contains a single channel, which make it gray-scale when visualized. We can denote the 4D shape of the input object into this branch as:

$$X_1 \in \mathbb{R}^{(64\times560\times256\times1)} \tag{20}$$

Any other shape will be rejected by the model, and an error is raised.

After the input layer assert the appropriate shape, the array X_1, which is a collection of 64 spectrograms, is passed into the first of three *Convolution Layer Groups*. These layer groups are inspired from the *VGG-16 Neural Network* architecture [citation needed]. Each convolution layer group in composed of three individual layers:

1. A 2-Dimensional Convolution layer, using 32 filters, a 3×3 kernel, a 1×1 step size, and a ReLU activation function,
2. A 2-Dimensional Convolution layer, using 32 filters, a 3×3 kernel, a 1×1 step size, and a ReLU activation function,
3. A 2-Dimensional Maximum Pooling layer using a 3×3 pooling size, and an Identity activation function

The Convolution layers convoluted over the middle two axes of the data (over space and time in the spectrogram).

By grouping layers in this structure, each of the 32 filters passes over a 3×3 area, and then again. The result is

1.6.2 The Perceptron Branch

The Perceptron branch is picture on the right side of fig. (**??**), notice that it is considerably smaller in side and complexity than it's neighbor. Rather than accept an image-like input, the perception simply takes a vector-like input of properties of a larger data set - in this case, each waveform. We call these properties *features* [1, 5, 16]. Details on the creating of these features can be found in the *features* section of this document.

The input layer of the perceptron accepts a 2-Dimensional array. The axes, in order of indexing, represent (i) the size of the *mini-batch* of samples, (ii) the number of features for each sample. We use the same model hyper-paramater of 64 samples per batch, and have developed 23 unique clssification features that have been derived from time-space and frequency-space representations of the audio file data. We can denote the 2D shape if the input object into this branch as:

$$X_2 \in \mathbb{R}^{(64\times 20)} \tag{21}$$

Again, any other shape will be rejected by the model, and an error is raised.

1.6.3 The Final Output Branch

The last layer in eachof the two branches is a ReLu-activated Dense layer containing 64 neurons. The activations are concatenated

References

[1] Geron, Aurelien. *Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly, 2017.

[2] Geron, Aurelien. *Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 2nd ed., O'Reilly, 2019.

[3] Goodfellow, Ian, et al.*Deep Learning*. MIT Press, 2017.

[4] James, Gareth, et al. *An Introduction to Statistical Learning with Applications in R*. Springer, 2017.

[5] Khan, M. Kashif Saeed, and Wasfi G. Al-Khatib. "Machine-Learning Based Classification of Speech and Music." Multimedia Systems, vol. 12, no. 1, 2006, pp. 55–67., doi:10.1007/s00530-006-0034-0.

[6] Levine, Daniel S. *Introduction to Neural and Cognitive Modeling*. 3rd ed., Routledge, 2019.

[7] Liu, Zhu, et al. "Audio Feature Extraction and Analysis for Scene Segmentation and Classification." Journal of VLSI Signal Processing, vol. 20, 1998, pp. 61–79.

[8] Loy, James , *Neural Network Projects with Python*. Packt Publishing, 2019

[9] McCulloch, Warren S., and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity." *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, 1943, pp. 115–133.

[10] Mierswa, Ingo, and Katharina Morik. "Automatic Feature Extraction for Classifying Audio Data." *Machine Learning*, vol. 58, no. 2-3, 2005, pp. 127–149., doi:10.1007/s10994-005-5824-7.

[11] Mitchell, Tom Michael. Machine Learning. 1st ed., McGraw-Hill, 1997.

[12] Olson, Harry E. *Music, Physics and Engineering*. 2nd ed., Dover Publications, 1967.

[13] Peatross, Justin, and Michael Ware. *Physics of Light and Optics*. Brigham Young University, Department of Physics, 2015.

[14] Petrik, Marek. "Introduction to Deep Learning." Machine Learning. 20 April. 2020, Durham, New Hampshire.

[15] Short, K. and Garcia R.A. 2006. "Signal Analysis Using the Complex Spectral Phase Evolution (CSPE) Method." AES: *Audio Engineering Society Convention Paper*.

[16] Virtanen, Tuomas, et al. *Computational Analysis of Sound Scenes and Events*. Springer, 2018.

[17] White, Harvey Elliott, and Donald H. White. *Physics and Music: the Science of Musical Sound*. Dover Publications, Inc., 2019.

[18] Zhang, Tong, and C.-C. Jay Kuo. "Content-Based Classification and Retrieval of Audio." *Advanced Signal Processing Algorithms, Architectures, and Implementations VIII*, 2 Oct. 1998, pp. 432–443., doi:10.1117/12.325703.