Topic: Impact of Configurable Parameters on Convolutional Neural Networks Performance in Speech Recognition

RQ: "How do configurable parameters such as network depth, size, and activation functions impact the performance of CNNs in transcribing speech accurately?"

Wordcount: 3997

Table Of Contents

1. Introduction:

In recent years, the application of convolutional neural networks (CNNs) to speech recognition has garnered significant attention within the research community. Foundational studies by Abdel-Hamid et al. (2014)[1] introduced CNN architectures specifically designed to extract robust features from raw audio signals—demonstrating that convolutional filters could effectively capture local time-frequency patterns and filter out background noise. Building on this, Sainath et al. (2015)[2] experimented with deeper network architectures and larger filter banks, showing that these modifications could lead to notable improvements in transcription performance under varied acoustic conditions. In contrast, Li et al. (2018)[3] presented a counterclaim, arguing that merely increasing network depth or filter size does not necessarily yield proportional gains in transcription accuracy—suggesting that integrating temporal dynamics, such as those captured by recurrent layers, might be crucial for real-world applications.

Despite these advances, there remains a notable gap in the literature regarding a systematic investigation of how specific configurable parameters—such as network depth, filter sizes, and activation functions—influence transcription accuracy under diverse real-world conditions. This study seeks to address this gap by rigorously evaluating the impact of these parameters on the performance of CNN-based speech recognition systems. The study aims to evaluate the configurable parameters of CNNs regarding their ability to transcribe speech accurately.

Thus, this study will explore **how configurable parameters such as depth of the network, size of the network, and types of activation function affect CNNs in accurately transcribing speech.** By exploring such parameters, the research seeks to optimize CNNs for better audio handling, greatly benefiting software companies, developers, and, most importantly, end users.

2. Background

2.1 Speech Recognition

Speech recognition, also called Automatic Speech Recognition (ASR), converts spoken language into text or other machine-readable forms.[4]Well-known examples include real-time captions on

[1] https://ieeexplore.ieee.org/document/6857341

[2] https://www.researchgate.net/publication/308872979_Convolutional_Long_Short-Term_Memory_fully_connected_Deep_Neural_Networks

[3] https://arxiv.org/abs/1808.09522

[4] https://nordvpn.com/blog/what-is-speech-recognition/

TikTok or Instagram, podcast transcriptions on Spotify, and automated meeting transcriptions in Zoom.[5]

2.2 Neural Networks

Neural Networks (NNs) are a class of machine learning models to recognize patterns. It is a type of deep learning algorithm and it falls under a subset of machine learning.[6] It consists of interconnected nodes or neurons in a layered structure that resembles the human brain. The basic structure includes an input layer, one or more hidden layers, and an output layer. It continuously uses its nodes to learn from its mistakes and find the optimal combination that gives the lowest error.[7] The number of layers creates a *neural network depth*.

Each neuron has an associated weight to it by summing up the previous layer's weights. Neural networks are trained on data and make more accurate predictions on the data by adjusting their weights by setting constant values to them. These constant values are known as bias and are adapted over time. They learn from the data by changing their weight and bias based on their predictions, improving accuracy over time. They learn through a process known as backpropagation. It adjusts the weights and biases based on the error and then propagates the error through the network and the weights are updated using an optimization algorithm like gradient descent.[8]

2.2.1 Activation Functions

Then, they process the input through an activation function. Essentially, the activation function is a mathematical function that transforms the input. It adds complexity to the node, making the summation non-linear.

This allows the model to analyze a wide range of data and introduces complexity into the neural network, which allows it to comprehend complicated spatial patterns within data. The most common activation functions used in CNNs are ReLU, Leaky ReLU, and Tanh.

[5] https://www.assemblyai.com/blog/what-is-asr/

[6] https://www.ibm.com/topics/convolutional-neural-networks

[7] https://aws.amazon.com/what-is/neural-network/

[8] www.packtpub.com/product/python-machine-learning/9781783555130)

The neural network formula is important to understand as it explains how *any* neuron in a neural network (including those in Convolutional Neural Networks) computes its output. It is crucial to comprehend this for the research question as CNNs are built on the foundation of neural network nodes.

A node within a neural network can be expressed as equation 1:

$$a = \sigma(z) \ldots(1)$$

- ➔ a is the output of the neuron
- ➔ σ is the activation function applied to the weighted sum.
- ➔ z is the weighted sum of inputs plus the bias term.

The activation function acts as a mathematical function that transforms the input Z. Input Z can be expressed as equation 2:

$$z = \sum_{i=1}^{n} w_i x_i + b \ldots(2)$$

w_i: Weights controlling each input's influence on the neuron.

x_i : The i-th output from the previous layer, serving as input.

b: bias term that shifts the activation threshold, aiding in fitting the data

n: the total number of inputs from previous layers.

Therefore, if combining this equation with our initial, it can represent the node function as equation (3) (*Deep Learning Book*)[9]:

$$a = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right) \ldots(3)$$

Again, each node is within a layer, and a layer is within the larger neural network. Building upon this understanding of neural networks, I delved into a specialized type known as Convolutional Neural Networks (CNNs). CNNs are particularly effective in processing temporal data like speech signals, making them a focal point of my research.

[9] https://www.deeplearningbook.org/

2.3 Convolutional Neural Networks

Traditional NNs were created to analyze and predict tabular data. However, they do not show prominent results in audio data. Audio signals are complex with rich patterns and temporal sequences which need specialized models to decipher.

Convolutional Neural Networks (CNNs) employ a layering process to address the limitations of traditional NNs in terms of temporal data.

- ➔ Convolutional layer
- ➔ Pooling layer
- ➔ Fully-connected (FC) layer[10]

2.3.1 Convolutional Layer

The convolutional layer applies local filters to regions of the sequential data. The filters move across the dataset and attempt to detect patterns such as frequency, pitches, and temporal dynamics.

The Filter equation can be represented as equation (4) (5) (*Deep Learning Book)*[11]*:*

$$z[n] = \sum_{i=0}^{k-1} x[n \bullet s + i - p]w[i] + b \ldots (4)$$

$$a[n] = \sigma(z[n]) \ldots(5)$$

- x is the raw audio signal (a 1D array of samples),
- w is the convolution kernel of length K,
- b is the bias term,
- s is the stride, determining how far the filter moves at each step.
- p is the padding applied to the input (which helps control the output size), and
- σ is the activation function (such as ReLU) that introduces non-linearity.

10 https://www.ibm.com/topics/convolutional-neural-networks#:~:text=Convolutional%20neural%20networks%20are%20distinguished,Pooling%20layer

11 https://www.deeplearningbook.org/

See the Appendix for an example calculation. This equation shows how filter operations, stride, and padding extract features from raw audio, affecting the network's effective depth and layer interactions. Activation functions transform these features, linking CNN size and configuration to transcription performance. Convolution isolates speech from noise and, when effectively capturing relevant patterns, boosts accuracy. That is why understanding these concepts is critical, as CNNs rely on such filtering for robust audio processing.

Filtering techniques maintain a spatial hierarchy, letting the system distinguish features from background audio. For instance, it can separate important sounds (speech) from noise in an audio-based application.

Abstracting patterns reduce parameters, producing a smaller activation map than the input sequence. Sharing filter weights across time steps further cuts parameters and effectively captures temporal features. These properties highlight the significance of CNNs for efficient and accurate speech transcription.

2.3.2 Pooling layer

Pooling layers work by dividing the 1D into smaller regions from the convolutional layer and summarizing them into a single value. This process, often using a method called "max pooling" (which selects the highest value in that region), reduces the overall size of the data being processed. By doing this, pooling layers help the network focus on the most important features of the input—such as key patterns in an audio signal—while ignoring small, less important details like background noise.

Pooling layers help prevent overfitting by reducing the detail the network must learn, allowing it to generalize better when accurately transcribing speech.

2.3.3 Fully-Connected Layer

The fully connected layer flattens and joins all the activation maps from the filters and pooling. It essentially just amalgamates all the maps into one. Then, it feeds the data into a traditional neural network and performs a classification or regression.

2.3.4 Summary Of CNNs

Convolutional Neural Networks (CNNs) address the complexity of audio signals. By applying filters with strides and padding, convolution layers isolate speech from background noise and capture temporal structures. Pooling condenses these representations, preventing overfitting and enabling robust generalization. Finally, the fully connected layer aggregates all extracted features for the final classification. This make CNNs particularly effective for speech transcription tasks.

2.4 Configurable Parameters of CNNs for Speech Recognition

CNNs can be optimized to increase word recognition accuracy. Multiple factors can lead to fine-tuning, however, this study is taking the following due to their extensive literature:

- ➔ Depth Of Network
- ➔ Size of the Network
- ➔ Types of Activation Functions

2.4.1 Depth Of Network

The depth of the network refers to the number of layers present in the CNN. In Figure 1, there are 4 layers present. Increasing the depth of the network can increase its complexity and perceptibility for feature extraction.[12] Qian et al. showed that very deep CNN architectures (up to 12 layers) can reduce word error rates in speech recognition by capturing more complex patterns. Referring to my research question, one of the parameters: **'*Depth of Network*'** focuses on understanding the impact of the configurable factors of neural networks to improve the performance of CNNs in transcribing speech accurately. **Therefore, it is a justified factor of exploration within my research question.**

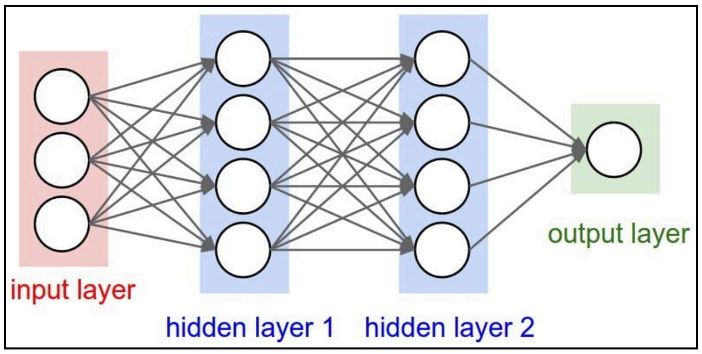


Figure 1: Neural Network Depth of 4 layers

[12] https://ieeexplore.ieee.org/document/7552554

2.4.2 Size Of Network

The size of the network refers to the width of each layer or the number of neurons present in the layers. In Figure 1, hidden layer 1 has a width of 4 neurons. Amodie, et al.'s study showed that increasing the size of the network improved the model's ability to generalize patterns and reduce WER in noisy environments.[13] Thus it may increase accuracy. **To analyze this rigorously,** the study will systematically vary the network size, then **quantitatively measure** the effects on accuracy to measure its impact on speech recognition performance.

2.4.3 Types Of Activation Functions

As said before, activation functions introduce non-linearity to networks, allowing them to interpret complex features. The following explanations justify the reason for selecting ReLU, Leaky ReLU, and TanH functions based on the literature below.

The **ReLU** activation function is prevalent in CNN-based speech recognition because of its **computational efficiency** and ability to **alleviate vanishing gradients** in deeper networks, making training faster and more stable (Krizhevsky et al., 2012)[14]. **Leaky ReLU** addresses the "dying ReLU" issue by allowing a small gradient for negative inputs and thus may better capture subtle features in noisy audio data (Sainath et al., 2015)[15]. **Tanh**, on the other hand, is favoured in certain tasks for its **range of [−1,1][-1,1][−1,1]** and zero-centered output, which can help in data normalization. These three activation functions have been **repeatedly cited** in both **peer-reviewed studies** (e.g., Qian et al. [2016][16] for noise-robust speech recognition) and **industry resources** (e.g., IBM's documentation on deep learning[17]), thereby **validating their relevance** to modern speech recognition pipelines.

[13] https://arxiv.org/abs/1512.02595

[14] https://papers.nips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

[15] https://www.researchgate.net/publication/308872979_Convolutional_Long_Short-Term_Memory_fully_connected_Deep_Neural_Networks

[16] https://linkinghub.elsevier.com/retrieve/pii/S0925231217311980

[17] https://www.ibm.com/think/topics/deep-learning

2.4.3.1 ReLU Functions

The ReLU Function (Rectified Linear Unit) takes all the inputs from the neurons and then adds their weights. Essentially, if there is value x_1 , x_2 , x_3 and weights w_1 , w_2 , w_3 then it sums up: $x_1w_1 + x_2w_2 + x_3\ w_3$. As shown in Figure 2, if the sum is below 0 then the neuron is set to 0. On the other hand, if the sum is above 0, the neuron is equivalent to that. So for $sum\ of\ weights\ =\ x$. This is shown in equation (6).

$$f(x)\ =\ max\ (0,\ x)\ \ldots(6)$$

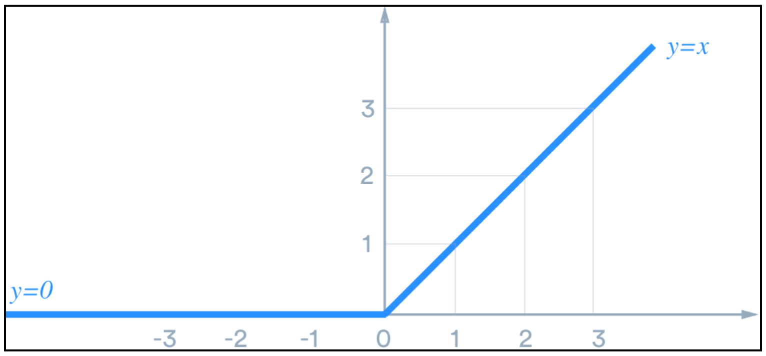


Figure 2: ReLU Function Graph

While it is efficient, as it has to choose between the Sum of weights or 0, it introduces possible flaws, as some neurons may 'die' due to the negative input sum. This may restrict the model's ability to detect possible patterns and complexities.

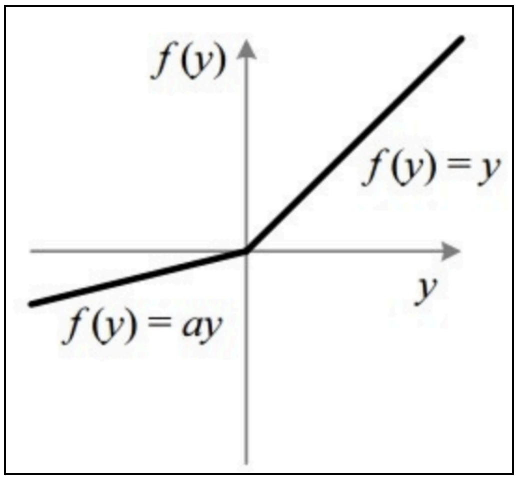


Figure 3: Leaky ReLU Function Graph

2.4.3.2 Leaky ReLU Function

To mitigate the problem above, Nair developed the Leaky ReLU. It is a variation of the ReLU that allows a small, non-zero gradient when the input is negative as shown in Figure 3. This addresses the "dying ReLU" problem where neurons can become inactive and stop learning. The Leaky ReLU function is defined in equation (7):

$$f(x) = x \mid if\ x > 0 \ldots(7)$$

However, $if\ x < 0$ then it is represented as equation (8):

$$f(x) = \alpha x \ldots(8)$$

Where α is a small constant.

2.4.3.3 Tanh Function

The Tanh function maps inputs to a range of -1 to 1 (Figure 4), ensuring symmetrical weight adjustments and preventing extreme imbalances. However, very large or small inputs can saturate at -1 or 1, leading to uneven squashing for unnormalized outputs. This slows learning for extreme values and may hinder efficient convergence.

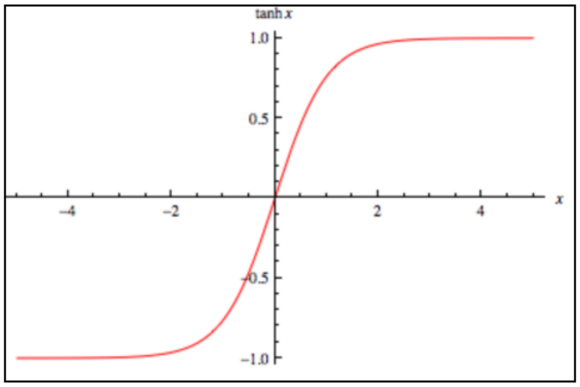


Figure 4: TanH Function

Exploring activation functions, a configurable parameter, is crucial to understanding their direct impact on CNN performance in speech transcription, thereby addressing my research question.

With a comprehensive understanding of CNNs and their configurable parameters, the research was based on the design of an experiment aimed at investigating how these parameters impact speech recognition accuracy.

2.5 Overfitting Data

Overfitting occurs when a model excels on training data but fails to generalize, "memorizing" noise instead of learning meaningful patterns. In speech recognition, a CNN may capture background noise from training audio, impairing its ability to transcribe new samples.

This phenomenon is **directly tied** to the research question of *how configurable parameters such as network depth, size, and activation functions impact the performance of CNNs in transcribing speech accurately*. While deeper and wider networks extract more features, they risk learning random fluctuations over genuine speech patterns. Activation functions, such as ReLU variants, influence overfitting by shaping how negative values and gradients are handled.

3. Methodology:

To investigate how configurable parameters like network depth, size, and activation functions affect CNN performance in speech transcription, an experiment was conducted, varying these factors across 27 model configurations. Using processed data from public datasets, training and validation accuracy were measured. Noting a gap in sound feature extraction research, an experimental setup was adopted to systematically manipulate variables and derive effective results. The code is included in the appendix. Furthermore, this approach facilitated the creation of variables that could be easily manipulated to determine effective results.

3.1 Dataset Used

Google's Speech Commands Dataset[18], a secondary public dataset of 65,000 one-second utterances of 30 words by thousands of speakers, was used. Thus, an unbiased output was ensured, mitigating potential biases arising from accents, locations, and speaking styles. Its background noise variations made it an ideal choice for testing CNN robustness. For this study, the noisiest samples were selected, while the rest were filtered out, aligning with the assessment

[18] https://research.google/blog/launching-the-speech-commands-dataset/

of CNNs' ability to filter out background noise. Consequently, the dataset was reduced to approximately 38,000 audio files.

3.2 Processing The Dataset

1. **Dataset of Audio Files Containing 30 Spoken Words/Classes**
   30 unique words enable multi-class CNN evaluation, providing a robust benchmark for varying network depths and activation functions.

2. **Audio Files Loaded with Librosa at 16 kHz**
   Standardizing to 16 kHz ensures uniform data and preserves critical speech frequencies for effective feature extraction.

3. **Numerical Labels (0–29)**
   Encoding words as labels (0–29) simplifies multi-class classification, aiding analysis of how depth and activation impact convergence.

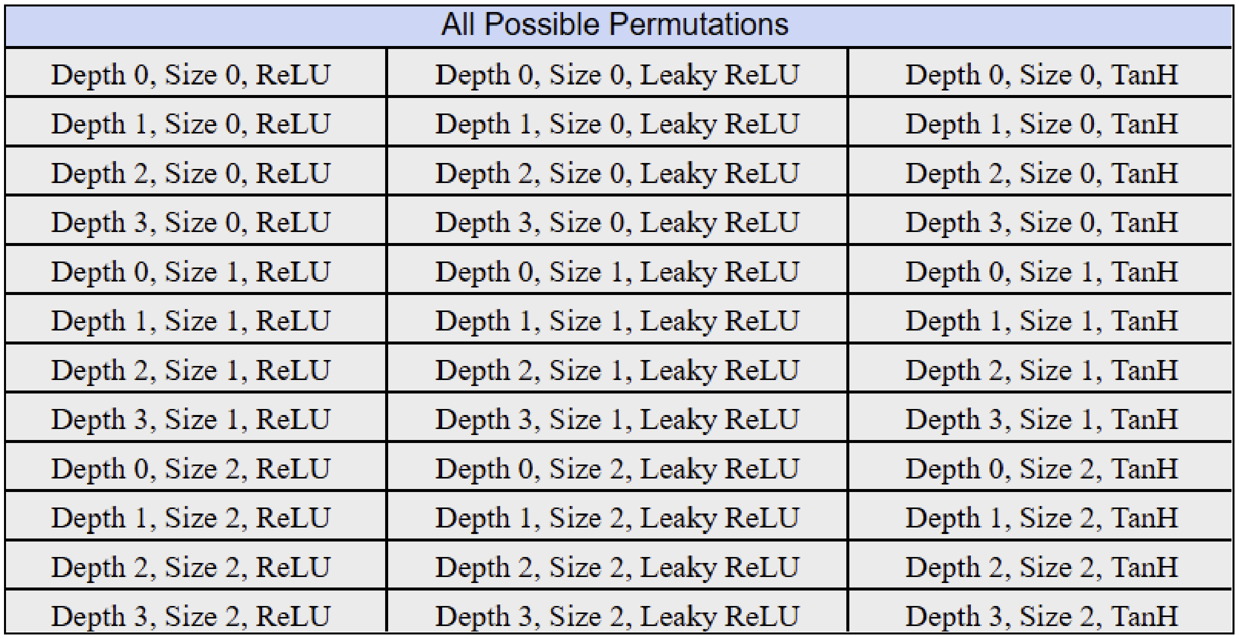
4. **Data Reshaped to (Samples, 16000, 1)**
   This fixed shape standardizes input for consistent CNN comparisons and optimizes temporal feature extraction from single-channel audio.

5. **90/10 Training/Testing Split**
   A large training set (90%) captures speech variability, while the 10% test set evaluates generalization and prevents overfitting under different configurations.

By structuring the dataset effectively, this preprocessing ensures that CNN architectures with varying depths, sizes, and activation functions receive a consistent and well-prepared input. This consistency is essential to accurately measure the impact of each configurable parameter on speech recognition performance.

3.3 Model Architecture.

Since this research evaluates the effect of 3 distinct parameters that can affect the performance of a CNN, an algorithm that creates 3 variations for each instance was formed. Thus, the setup will have a total of 3*3*3=27 configurations of CNN models as seen in Figure 5.



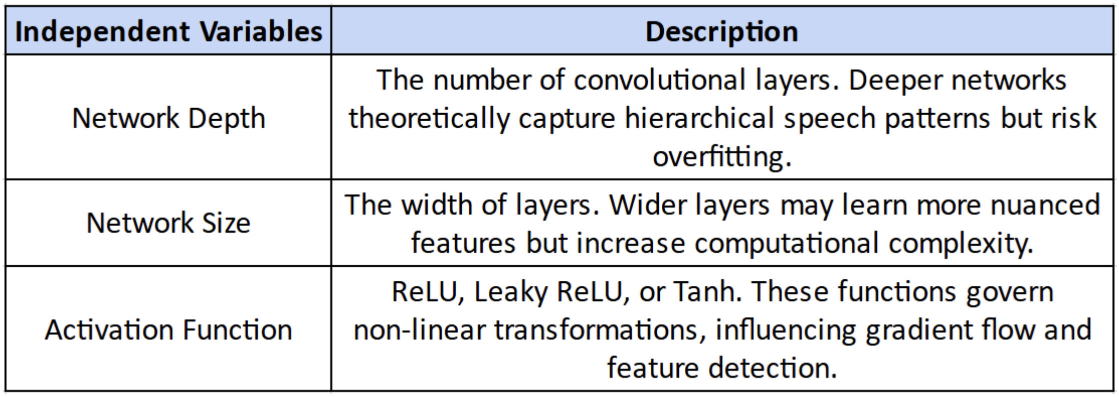
All Possible Permutations		
Depth 0, Size 0, ReLU	Depth 0, Size 0, Leaky ReLU	Depth 0, Size 0, TanH
Depth 1, Size 0, ReLU	Depth 1, Size 0, Leaky ReLU	Depth 1, Size 0, TanH
Depth 2, Size 0, ReLU	Depth 2, Size 0, Leaky ReLU	Depth 2, Size 0, TanH
Depth 3, Size 0, ReLU	Depth 3, Size 0, Leaky ReLU	Depth 3, Size 0, TanH
Depth 0, Size 1, ReLU	Depth 0, Size 1, Leaky ReLU	Depth 0, Size 1, TanH
Depth 1, Size 1, ReLU	Depth 1, Size 1, Leaky ReLU	Depth 1, Size 1, TanH
Depth 2, Size 1, ReLU	Depth 2, Size 1, Leaky ReLU	Depth 2, Size 1, TanH
Depth 3, Size 1, ReLU	Depth 3, Size 1, Leaky ReLU	Depth 3, Size 1, TanH
Depth 0, Size 2, ReLU	Depth 0, Size 2, Leaky ReLU	Depth 0, Size 2, TanH
Depth 1, Size 2, ReLU	Depth 1, Size 2, Leaky ReLU	Depth 1, Size 2, TanH
Depth 2, Size 2, ReLU	Depth 2, Size 2, Leaky ReLU	Depth 2, Size 2, TanH
Depth 3, Size 2, ReLU	Depth 3, Size 2, Leaky ReLU	Depth 3, Size 2, TanH

Figure 5: All Permutations of models

Each factor: depth, size, and filter was assigned a particular architecture that formed the overarching network. The appendix contains the different variations in size, depth, and filter. Tensorflow was employed in order to carry out this experiment with 10 epochs. A possible limitation of 10 epochs was a result of hardware limitations.

These variations join together to form 27 distinct models. This is done through the Cartesian product, which combines all the layers to form all possible permutations. Itertools was used to algorithmically generate and assess each model. After each iteration, the model's performance is evaluated using a chosen metric (e.g., accuracy). Moreover, each model used only one type of activation function(ReLU, Leaky ReLU, or TanH) to ensure consistency throughout the process. However, this may give one-dimensional results as most real-world neural networks are a combination of distinct activation functions. Nonetheless, this process had to be followed as performing an incremental (only the filter layer has a different activation function) led to inconsistencies during testing. For example, the ReLU function does not use values below 0. Thus, it leads to Leaky ReLU (activation functions that use values below 0) to have a nullified effect.

3.5 Independent Variables



Independent Variables	Description
Network Depth	The number of convolutional layers. Deeper networks theoretically capture hierarchical speech patterns but risk overfitting.
Network Size	The width of layers. Wider layers may learn more nuanced features but increase computational complexity.
Activation Function	ReLU, Leaky ReLU, or Tanh. These functions govern non-linear transformations, influencing gradient flow and feature detection.

Figure 6: Independent Variables and Descriptions

The independent variables in Figure 6 were systematically varied to isolate their effects on transcription accuracy, directly addressing the research question. These parameters were selected because they directly define a CNN's capacity to extract temporal features from audio signals as highlighted in the background research.

3.6 Dependent Variable

The dependent variable is transcription accuracy, measured as the proportion of correctly classified speech samples in the test set.

For the Speech Commands Dataset (30 isolated words), accuracy is the most interpretable metric. It directly answers the research question's focus on performance. While metrics like loss or Word Error Rate (WER) are useful for free-form speech, accuracy is ideal for closed-vocabulary classification tasks.

3.7 Hypothesis

A. Depth Of Network

The depth of the network is likely going to be a large predictor of the accuracy of each model. The more depth a model has, the enhanced its ability will be to extract complex patterns as it will be able to capture more elements. This is because deeper networks can capture more detailed patterns and hierarchical representations of the audio signals

B. Size of the Network

Increasing the network size, specifically the number of neurons in each layer, will improve the model's performance by allowing it to capture more nuanced features from the audio data. However, the performance gains may plateau or degrade beyond a certain point due to overfitting. It may occur as the machine learning model considers noise and background noise as actual data and is overtrained on the features. It may lead to the capture of noise within the data.

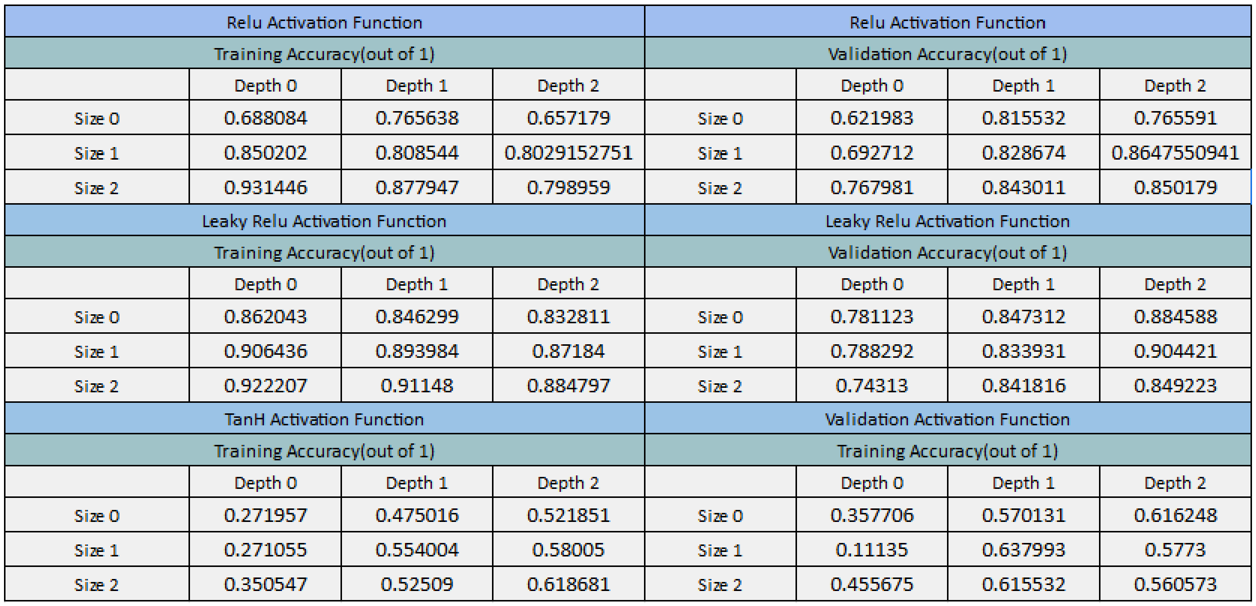
C. Types Of Activation Functions

Different types of Activation functions (ReLU, Leaky ReLU, Tanh) will impact CNN's performance differently. ReLU is expected to perform well due to its simplicity and effectiveness in most deep learning tasks, while Leaky ReLU might slightly improve handling negative values. Tanh might offer better results in noisy environments as a smoother activation function by providing a more normalized output.

4. Experiment Result Analysis and Conclusion

4.1 Table of Training Results:

In this section, the experimental results are analyzed to determine how each configurable parameter influences CNN performance, thereby providing insights into how configurable parameters such as network depth, size, and activation functions impact the accuracy of CNNs in transcribing speech. The table below presents the experiment results, showcasing the training and validation accuracy of each model.



Relu Activation Function				Relu Activation Function			
Training Accuracy(out of 1)				Validation Accuracy(out of 1)			
	Depth 0	Depth 1	Depth 2		Depth 0	Depth 1	Depth 2
Size 0	0.688084	0.765638	0.657179	Size 0	0.621983	0.815532	0.765591
Size 1	0.850202	0.808544	0.8029152751	Size 1	0.692712	0.828674	0.8647550941
Size 2	0.931446	0.877947	0.798959	Size 2	0.767981	0.843011	0.850179
Leaky Relu Activation Function				Leaky Relu Activation Function			
Training Accuracy(out of 1)				Validation Accuracy(out of 1)			
	Depth 0	Depth 1	Depth 2		Depth 0	Depth 1	Depth 2
Size 0	0.862043	0.846299	0.832811	Size 0	0.781123	0.847312	0.884588
Size 1	0.906436	0.893984	0.87184	Size 1	0.788292	0.833931	0.904421
Size 2	0.922207	0.91148	0.884797	Size 2	0.74313	0.841816	0.849223
TanH Activation Function				Validation Activation Function			
Training Accuracy(out of 1)				Training Accuracy(out of 1)			
	Depth 0	Depth 1	Depth 2		Depth 0	Depth 1	Depth 2
Size 0	0.271957	0.475016	0.521851	Size 0	0.357706	0.570131	0.616248
Size 1	0.271055	0.554004	0.58005	Size 1	0.11135	0.637993	0.5773
Size 2	0.350547	0.52509	0.618681	Size 2	0.455675	0.615532	0.560573

Figure 7: Training and Validation Testing Raw Results

4.2 Data Analysis

To better and effectively understand the trends in each CNN's performance, I plotted multiple bar graphs using Matplotlib Python library[19] to display the accuracy of each adjustment.

The first column of the bar graphs in Figure 7 represent the respective activation function used. The second column represents the validation accuracy of the rest of the dataset. The second column will verify the accuracy of the training as it verifies that the model is not overtrained on the data. Each bar represents the accuracy on a scale of 0 to 1 on Epoch 10.

[19] https://matplotlib.org/

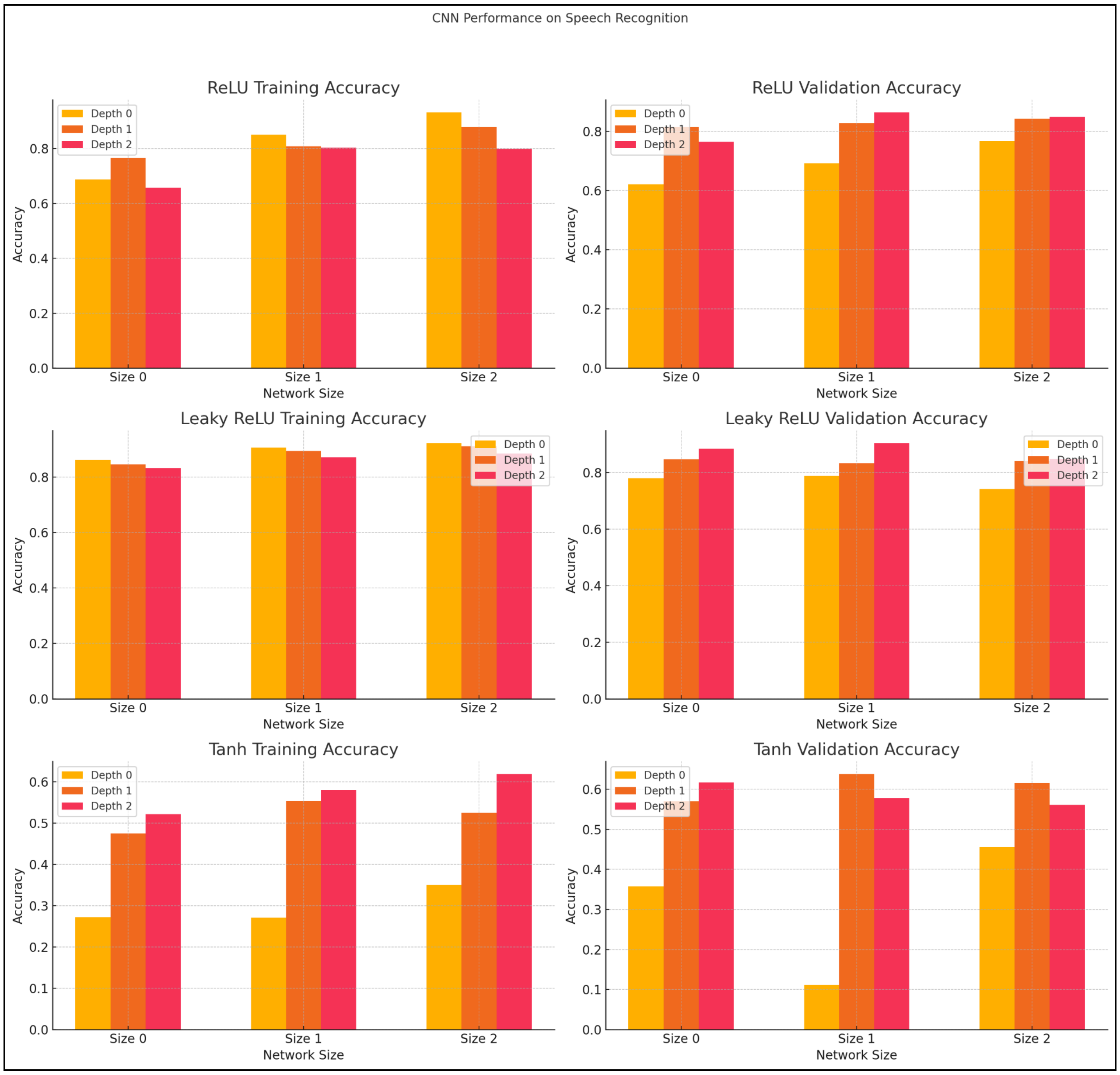


Figure 8: Testing and Training Graphed Results

Now, I will stepwise analyze each feature and its effect on CNN's performance.

4.2.1 Depth Analysis

Observing figure 7 and 8, both ReLU and Leaky ReLU benefit from increased network depth, as seen by the higher validation accuracy 0.765591 (Size 0, Depth 2) vs. 0.621983 (Size 0, depth 0) for ReLU, and validation accuracy 0.781123 (Depth 0, Size 0) vs. 0.884588 (Depth 0, Size 2) for Leaky ReLU, supporting hypothesis A. However, deeper ReLU networks risk overfitting, as evidenced by the drop in training accuracy from 0.765638 to 0.657179 (Size 0, Depth 1 to 2).

This trend aligns with Abdel-Hamid et al. (2014)[20] and Simonyan and Zisserman (2014)[21], who both underscore the importance of balancing network depth with proper regularization.

4.2.2 Size Analysis

Increasing the neuron count boosts accuracy, supporting hypothesis B that larger networks capture more features. For instance, in the ReLU configuration at Depth 0, the training accuracy rises from 0.688084 (Size 0, Depth 0) to 0.931446 (Size 2, Depth 0), with a corresponding improvement in validation accuracy (0.621983 to 0.767981). However, best results occur at moderate sizes, as overly large networks start to overfit. He et al. (2016)[22] contradict this view by demonstrating that with the right techniques, increasing network size can continue to enhance performance without substantial overfitting. Therefore, according to his research, the methodology could have been altered by experimenting with weight initialisation and batch normalisation to handle the overfitting.

4.2.3 Activation Function Analysis

ReLU delivers strong relative training performance but shows improvements with additional layers. Leaky ReLU, on the other hand, handles smaller configurations effectively (training accuracy 0.862043 at Size 0, Depth 0 compared with 0.68804 of ReLU) and scales well, mitigating the dying neuron issue and offering better generalization. Moreover, Leaky ReLU often outperforms ReLU in validation accuracy (0.904421 vs. 0.864756). In contrast, Tanh consistently underperforms and shows relatively unpredictable results(with a peak training accuracy of only 0.618681 at Size 2, Depth 2), which is in line with Xu et al. (2015)[23]'s findings regarding its limitations in complex, noisy environments. My findings contradict hypothesis C. However, earlier work by LeCun et al. (1998)[24] contradicts this trend by showing that Tanh can be quite effective when the network is properly tuned. Therefore, possibly looking deeper into TanH activation functions could be future research.

4.3 Optimal Combinations and Trade-offs

For ReLU, a configuration of Depth 2 and Size 1 achieves the highest validation accuracy (0.864755), striking a balance between learning capacity and generalization. Although a more complex configuration (Depth 2, Size 2) offers higher training accuracy, it compromises

[20] https://ieeexplore.ieee.org/document/6857341
[21] https://arxiv.org/abs/1409.1556
[22]
https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf
[23] https://arxiv.org/abs/1502.03044
[24] https://ieeexplore.ieee.org/document/726791

validation performance. Leaky ReLU shows a slight advantage, with its best performance at Depth 2 and Size 1 (0.904421), though similar trade-offs exist with increased complexity. TanH performs best at a moderate setting (Depth 1, Size 1) with a validation accuracy of 0.637993, but higher configurations lead to vanishing gradients, further limiting its effectiveness.

Overall, while ReLU and Leaky ReLU both perform robustly across various configurations, Leaky ReLU's improved consistency and resilience against neuron inactivation make it more favourable in scenarios where maintaining high activation is critical.

5. Limitations and Possible Improvements

5.1 Hardware Limitations

The experiment lacked GPU acceleration, restricting complexity, experiment count, and efficiency. Using a GPU or distributed system could improve training speed, allowing more epochs for better optimization. Additionally, K-fold validation would enhance reliability, as only a single 10-epoch model was used.

5.2 Dataset Scope

This dataset had one-word utterances. However, natural speech over longer words is a much more complex phenomenon that has to process sequential data. This involves experimenting with recurrent neural networks, as CNNs do not perform well with long natural language alone. Therefore, the scope of the research question would increase.

6. Evaluation and Future Research

6.1 Hybrid Activation Functions

It would be interesting to observe the interaction of multiple activation functions. This would represent real-world CNN. Thus, balancing the individual strengths of each function.

6.2 Overfitting and Methods To Reduce

Overfitting was a common problem with datasets of complexity. If a study were done as a practical experiment looking at the effects of increasing size and depth and its effect on overtraining on CNNs with speech transcription, it could provide us with unique insights.

Moreover, the study could see the effect of implementing regularization techniques such as batch normalization, early stopping, or L2 regularization more aggressively, which could help mitigate overfitting.

7 Conclusion

This investigation examined how CNN configurations—depth, size, and activation functions—affect speech transcription accuracy. While my hypothesis emphasized network depth, it overlooked overtraining risks. Increased complexity reduced generalization, leading to overfitting, similar to the effects of larger network sizes. Activation functions proved crucial in training. ReLU and Leaky ReLU outperformed TanH, which struggled due to the vanishing gradient problem. Optimally, Leaky ReLU with size 1 and depth 2 achieved a validation accuracy of 0.904421. Though accuracy generally increased, inconsistencies prevented a clear trend.

Through this investigation, I deepened my understanding of CNNs and their role in speech recognition. This journey not only answered my research question but also ignited a desire to further explore machine learning applications in human communication.

8. Citations

Amodei, D., Anubhai, R., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Chen, J., Chrzanowski, M., Coates, A., Diamos, G., Elsen, E., Engel, J., Fan, L., Fougner, C., Han, T., Hannun, A., Jun, B., LeGresley, P., Lin, L., . . . Zhu, Z. (2015, December 8). *Deep Speech 2: End-to-End speech recognition in English and Mandarin*. arXiv.org. https://arxiv.org/abs/1512.02595

Deep learning. (n.d.). https://www.deeplearningbook.org/

Einorytė, A., & Einorytė, A. (2025, January 28). *What is speech recognition, and how does it work?* NordVPN. https://nordvpn.com/blog/what-is-speech-recognition/

Foster, K. (2024, August 20). *What is ASR? An Overview of Automatic Speech Recognition*. News, Tutorials, AI Research. https://www.assemblyai.com/blog/what-is-asr/

Ibm. (2024a, December 19). Convolutional Neural Networks. *What are convolutional neural networks?* https://www.ibm.com/topics/convolutional-neural-networks

Ibm. (2024b, December 19). Convolutional Neural Networks. *What are Convultional Neural Networks?* https://www.ibm.com/topics/convolutional-neural-networks#:~:text=Convolutional%20neural%20networks%20%20are%20distinguished,Pooling%20layer

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), 84–90. https://doi.org/10.1145/3065386

Launching the Speech Commands dataset. (n.d.). https://research.google/blog/launching-the-speech-commands-dataset/

Outline of the convolutional layer. (2017). Research Gate. https://www.researchgate.net/figure/Outline-of-the-convolutional-layer_fig1_323792694

Raschka, S. (2015, September 23). *Python Machine Learning | Data | Print*. Packt. https://www.packtpub.com/product/python-machine-learning/9781783555130

Very deep convolutional neural networks for noise robust speech recognition. (2016, December 1). IEEE Journals & Magazine | IEEE Xplore. https://ieeexplore.ieee.org/document/7552554

What is a Neural Network? - Artificial Neural Network Explained - AWS. (n.d.). Amazon Web Services, Inc. https://aws.amazon.com/what-is/neural-network/

Convolutional, Long Short-Term Memory, fully connected Deep Neural Networks. (2015, April 1). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/7178838/

Convolutional neural networks for speech recognition. (2014, October 1). IEEE Journals & Magazine | IEEE Xplore. https://ieeexplore.ieee.org/document/6857341

Gradient-based learning applied to document recognition. (1998, November 1). IEEE Journals & Magazine | IEEE Xplore. https://ieeexplore.ieee.org/document/726791

He, K., Zhang, X., Ren, S., & Microsoft Research. (n.d.). Deep residual learning for image recognition. In *Microsoft Research* [Journal-article]. https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). *ImageNet Classification with Deep Convolutional Neural Networks*. https://papers.nips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

Li, J., Liu, C., & Gong, Y. (2018, August 28). *Layer Trajectory LSTM*. arXiv.org. https://arxiv.org/abs/1808.09522

Qian, S., & Lia, H. (n.d.). Adaptive activation functions in convolutional neural networks. *ELSEVIER*, *272*, 204–212. https://doi.org/10.1016/j.neucom.2017.06.070

Simonyan, K., & Zisserman, A. (2014, September 4). *Very deep convolutional networks for Large-Scale image recognition*. arXiv.org. https://arxiv.org/abs/1409.1556

Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., Zemel, R., & Bengio, Y. (2015, February 10). *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*. arXiv.org. https://arxiv.org/abs/1502.03044

Matplotlib — Visualization With Python. (n.d.). https://matplotlib.org/

9. Appendix

Example calculation of a 1D convolution layer.

Convolution Example

Our Setup

- Input signal (x): [1, 2, 3, 4, 5, 6]
- Kernel (w): [1, 0, –1] (Kernel length: K = 3)
- Bias (b): 0
- Stride (s): 1
- Padding (p): 1
- Activation function: ReLU, where
$$ReLU(z) = max(0, z)$$
- Convolution Equation:
$$z[n] = \sum_{i=0}^{K-1} x[n \cdot s + i - p] \cdot w[i] + b$$
- Activation:
$$a[n] = ReLU(z[n])$$
- Padding: Any access to an index outside of the range of x is treated as 0.

Step 1: Determine the Output Size

For an input length $N = 6$, the output length L is:

$$L = \lfloor \frac{N+2p-K}{s} + 1 \rfloor$$

$$L = \lfloor \frac{6+2-3}{1} + 1 \rfloor$$

$$L = \lfloor \frac{5}{1} + 1 \rfloor = 6$$

Thus, we compute outputs for $n = 0, 1, 2, 3, 4, 5$.

Step 2: Compute $z[n]$ and $a[n]$ for Each n

For $n = 0$

Indices Calculation:

- $i = 0: x[0 \cdot 1 + 0 - 1] = x[-1]$ (out-of-bound, assume 0)
- $i = 1: x[0 \cdot 1 + 1 - 1] = x[0] = 1$
- $i = 2: x[0 \cdot 1 + 2 - 1] = x[1] = 2$

Apply Kernel:

$$z[0] = (0 \times 1) + (1 \times 0) + (2 \times (-1)) + 0 = 0 + 0 - 2 = -2$$

Activation:

$$a[0] = ReLU(-2) = 0$$

For $n = 1$

Indices Calculation:

- $i = 0: x[1 + 0 - 1] = x[0] = 1$
- $i = 1: x[1 + 1 - 1] = x[1] = 2$
- $i = 2: x[1 + 2 - 1] = x[2] = 3$

Apply Kernel:

$$z[1] = (1 \times 1) + (2 \times 0) + (3 \times (-1)) + 0 = 1 + 0 - 3 = -2$$

Activation:

$$a[1] = ReLU(-2) = 0$$

For $n = 2$

Indices Calculation:

- $i = 0: x[2 + 0 - 1] = x[1] = 2$
- $i = 1: x[2 + 1 - 1] = x[2] = 3$
- $i = 2: x[2 + 2 - 1] = x[3] = 4$

Apply Kernel:

$$z[2] = (2 \times 1) + (3 \times 0) + (4 \times (-1)) + 0 = 2 + 0 - 4 = -2$$

Activation:

$$a[2] = ReLU(-2) = 0$$

For $n = 3$

Indices Calculation:

- $i = 0: x[3 + 0 - 1] = x[2] = 3$
- $i = 1: x[3 + 1 - 1] = x[3] = 4$
- $i = 2: x[3 + 2 - 1] = x[4] = 5$

Apply Kernel:

$$z[3] = (3 \times 1) + (4 \times 0) + (5 \times (-1)) + 0 = 3 + 0 - 5 = -2$$

Activation:

$$a[3] = ReLU(-2) = 0$$

For $n = 4$

Indices Calculation:

- $i = 0: x[4 + 0 - 1] = x[3] = 4$
- $i = 1: x[4 + 1 - 1] = x[4] = 5$
- $i = 2: x[4 + 2 - 1] = x[5] = 6$

Apply Kernel:

$$z[4] = (4 \times 1) + (5 \times 0) + (6 \times (-1)) + 0 = 4 + 0 - 6 = -2$$

Activation:

$$a[4] = ReLU(-2) = 0$$

For $n = 5$

Indices Calculation:

- $i = 0: x[5 + 0 - 1] = x[4] = 5$
- $i = 1: x[5 + 1 - 1] = x[5] = 6$
- $i = 2: x[5 + 2 - 1] = x[6]$ (out-of-bound, assume 0)

Apply Kernel:

$$z[5] = (5 \times 1) + (6 \times 0) + (0 \times (-1)) + 0 = 5 + 0 + 0 = 5$$

Activation:

$$a[5] = ReLU(5) = 5$$

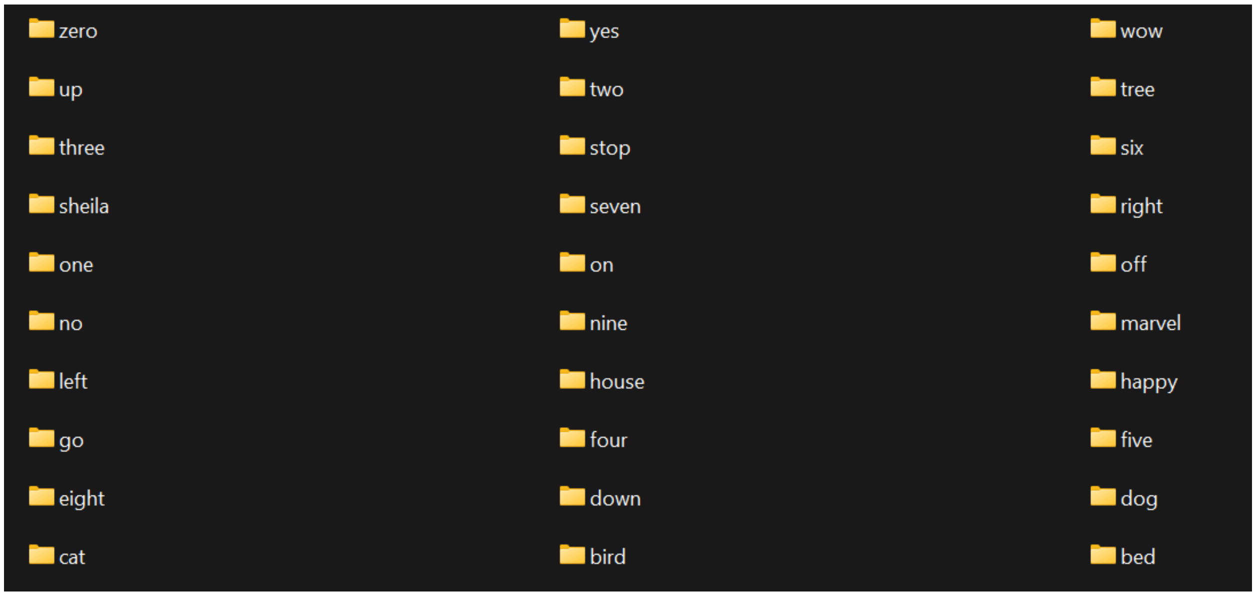
Final Output

After applying the convolution and activation function, the final output array a is:

$$a = [0, 0, 0, 0, 0, 5]$$

This example demonstrates how each output element is computed by sliding the kernel over the padded input, performing element-wise multiplication and summing the results, adding the bias, and finally applying the ReLU activation.

These is the folders that contain the .wav files that I used and is the Raw Data of the machine learning model.



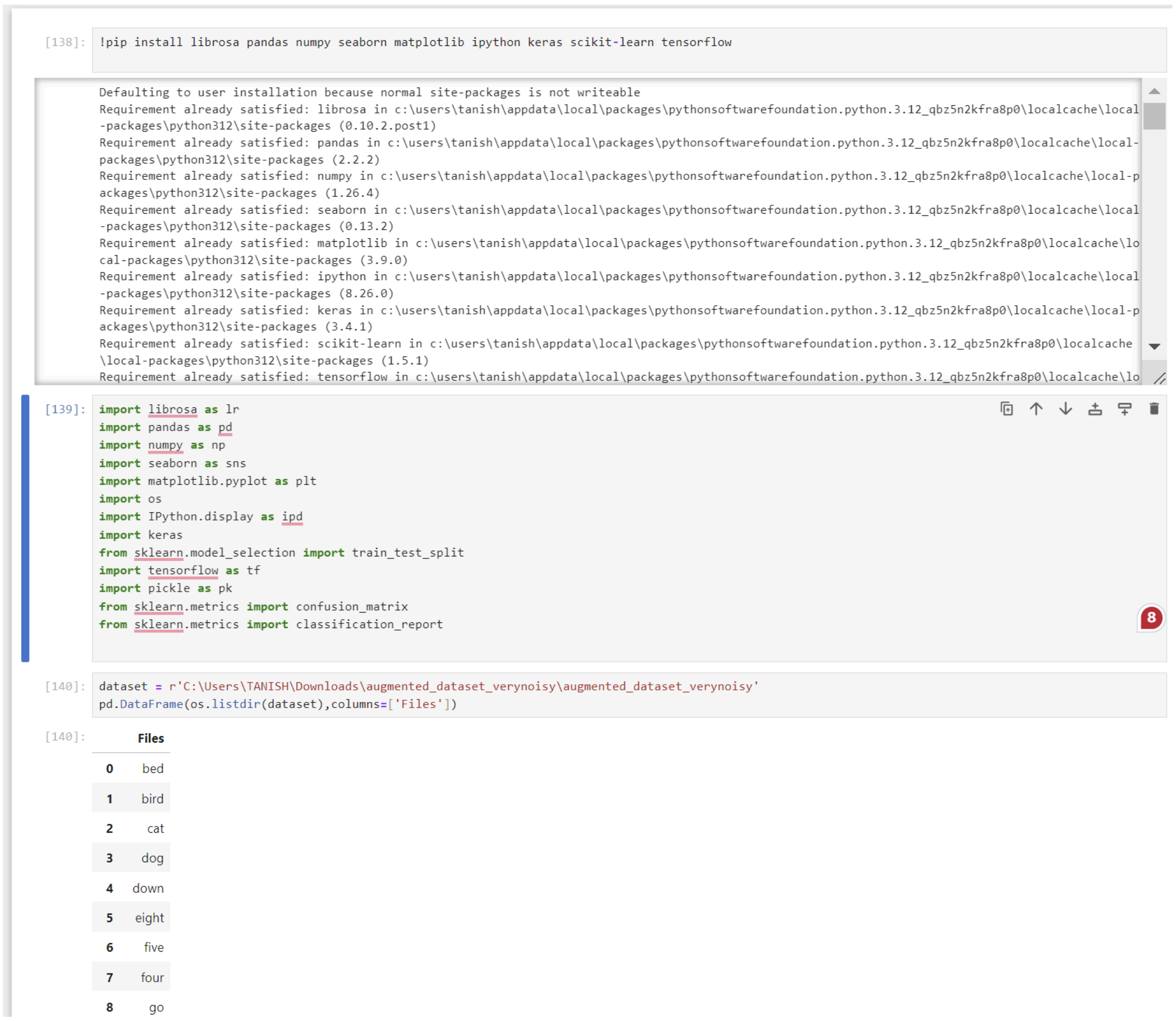
zero	yes	wow
up	two	tree
three	stop	six
sheila	seven	right
one	on	off
no	nine	marvel
left	house	happy
go	four	five
eight	down	dog
cat	bird	bed

These are the layers that I used to test:

```
def generate_layer_configs(activation_function):
    depth_configs = [
        [
            {'type': 'conv', 'filters': 8, 'kernel_size': 13,
'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3}
        ],
        [
            {'type': 'conv', 'filters': 8, 'kernel_size': 13,
'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'conv', 'filters': 16, 'kernel_size': 11,
'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3}
        ],
        [
            {'type': 'conv', 'filters': 8, 'kernel_size': 13,
'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'conv', 'filters': 16, 'kernel_size': 11,
'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'conv', 'filters': 32, 'kernel_size': 9,
'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3}
        ]
    ]

    size_configs = [
        [
            {'type': 'flatten'},
            {'type': 'dense', 'units': 128, 'activation':
activation_function},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'dense', 'units': 64, 'activation':
activation_function},
            {'type': 'dropout', 'rate': 0.3}
        ],
        [
            {'type': 'flatten'},
            {'type': 'dense', 'units': 256, 'activation':
activation_function},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'dense', 'units': 128, 'activation':
activation_function},
            {'type': 'dropout', 'rate': 0.3}
        ],
        [
            {'type': 'flatten'},
            {'type': 'dense', 'units': 512, 'activation':
activation_function},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'dense', 'units': 256, 'activation':
activation_function},
            {'type': 'dropout', 'rate': 0.3}
        ]
    ]

    filter_configs = [
        [
            {'type': 'conv', 'filters': 8, 'kernel_size': 13,
'activation': activation_function},
            {'type': 'conv', 'filters': 16, 'kernel_size': 11,
'activation': activation_function}
        ]
    ]
    return depth_configs, size_configs, filter_configs


##The Filter Config is ReLU, Leaky ReLU or Tanh. It is done in the model
building phase in the algorithm.
```

This is the code that I used to process, generate, and evaluate the models. I used it to create my CNNs, feed them Wav data in an array format, and train & evaluate them using specific configurations.

It was training through an i9 12900k without any GPU. Thus, there were limitations to processing as it was solely running through a CPU. I used a Jupyter notebook, and the cell-by-cell format is below.



```
[138]: !pip install librosa pandas numpy seaborn matplotlib ipython keras scikit-learn tensorflow
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: librosa in c:\users\tanish\appdata\local\packages\pythonsoftwarefoundation.python.3.12_qbz5n2kfra8p0\localcache\local
-packages\python312\site-packages (0.10.2.post1)
Requirement already satisfied: pandas in c:\users\tanish\appdata\local\packages\pythonsoftwarefoundation.python.3.12_qbz5n2kfra8p0\localcache\local-
packages\python312\site-packages (2.2.2)
Requirement already satisfied: numpy in c:\users\tanish\appdata\local\packages\pythonsoftwarefoundation.python.3.12_qbz5n2kfra8p0\localcache\local-p
ackages\python312\site-packages (1.26.4)
Requirement already satisfied: seaborn in c:\users\tanish\appdata\local\packages\pythonsoftwarefoundation.python.3.12_qbz5n2kfra8p0\localcache\local
-packages\python312\site-packages (0.13.2)
Requirement already satisfied: matplotlib in c:\users\tanish\appdata\local\packages\pythonsoftwarefoundation.python.3.12_qbz5n2kfra8p0\localcache\lo
cal-packages\python312\site-packages (3.9.0)
Requirement already satisfied: ipython in c:\users\tanish\appdata\local\packages\pythonsoftwarefoundation.python.3.12_qbz5n2kfra8p0\localcache\local
-packages\python312\site-packages (8.26.0)
Requirement already satisfied: keras in c:\users\tanish\appdata\local\packages\pythonsoftwarefoundation.python.3.12_qbz5n2kfra8p0\localcache\local-p
ackages\python312\site-packages (3.4.1)
Requirement already satisfied: scikit-learn in c:\users\tanish\appdata\local\packages\pythonsoftwarefoundation.python.3.12_qbz5n2kfra8p0\localcache
\local-packages\python312\site-packages (1.5.1)
Requirement already satisfied: tensorflow in c:\users\tanish\appdata\local\packages\pythonsoftwarefoundation.python.3.12_qbz5n2kfra8p0\localcache\lo
```

```
[139]: import librosa as lr
       import pandas as pd
       import numpy as np
       import seaborn as sns
       import matplotlib.pyplot as plt
       import os
       import IPython.display as ipd
       import keras
       from sklearn.model_selection import train_test_split
       import tensorflow as tf
       import pickle as pk
       from sklearn.metrics import confusion_matrix
       from sklearn.metrics import classification_report
```

8

```
[140]: dataset = r'C:\Users\TANISH\Downloads\augmented_dataset_verynoisy\augmented_dataset_verynoisy'
       pd.DataFrame(os.listdir(dataset),columns=['Files'])
```

[140]:

	Files
0	bed
1	bird
2	cat
3	dog
4	down
5	eight
6	five
7	four
8	go

8	go
9	happy
10	house
11	left
12	marvel
13	nine
14	no
15	off
16	on
17	one
18	right
19	seven
20	sheila
21	six
22	stop
23	three
24	tree
25	two
26	up
27	wow
28	yes
29	zero

[141]:

```
def count(path):
    size=[]
    for file in os.listdir(path):
        size.append(len(os.listdir(os.path.join(path,file))))
    return pd.DataFrame(size,columns=['Number Of Sample'],index=os.listdir(path))
tr=count(dataset)
tr
```

[141]:

	Number Of Sample
bed	1356
bird	1346
cat	1378
dog	1474
down	1188
eight	1113

five	1092
four	2400
go	960
happy	1481
house	2382
left	1485
marvel	1253
nine	1144
no	957
off	2244
on	2228
one	1276
right	1276
seven	1411
sheila	1463
six	1485
stop	1485
three	1188
tree	1188
two	902
up	1187
wow	957
yes	1244
zero	1306

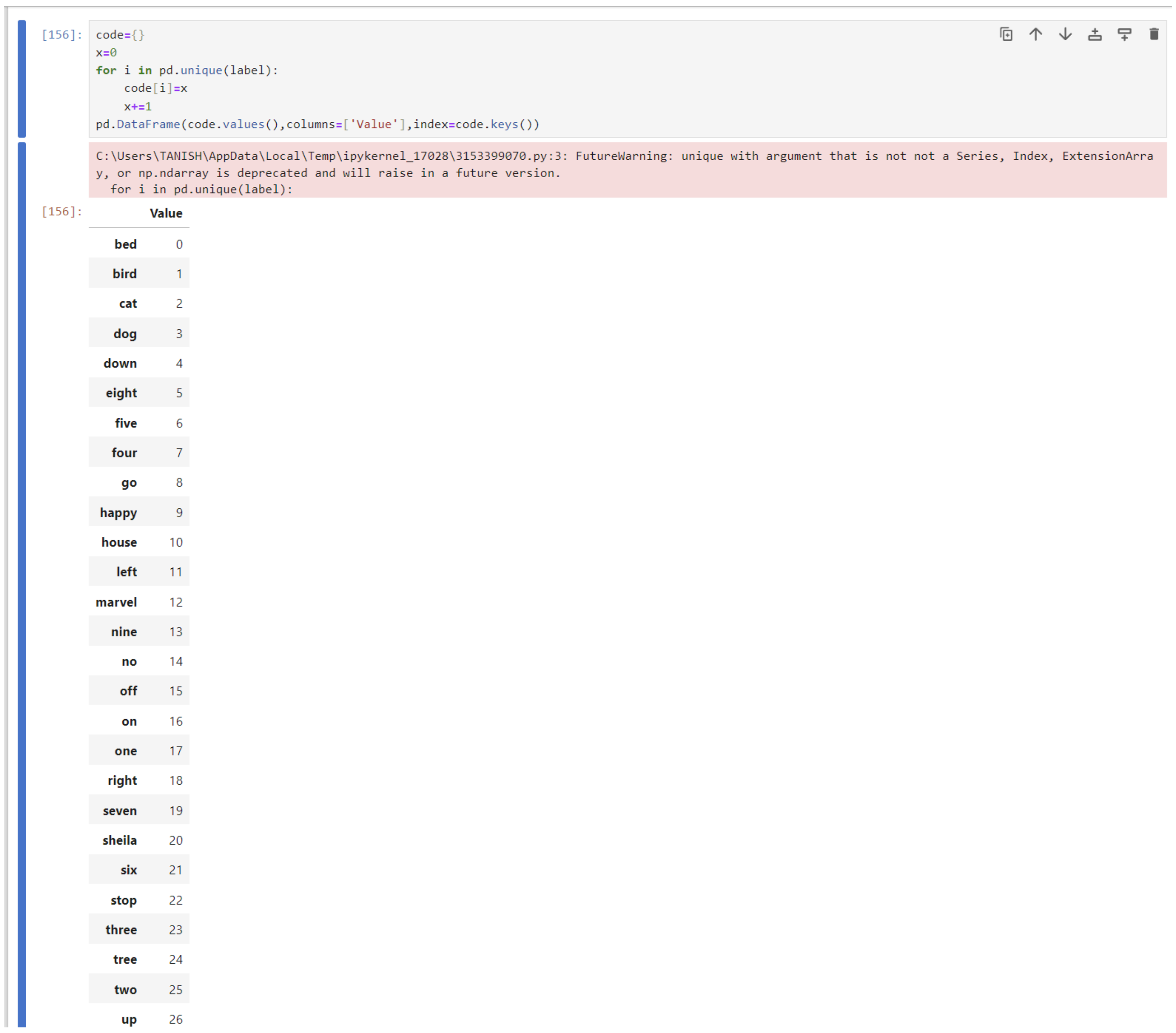
[144]:
```
def load(path):
    data=[]
    label=[]
    sample=[]
    for file in os.listdir(path):
        path_=os.path.join(path,file)
        for fil in os.listdir(path_):
            data_contain,sample_rate=lr.load(os.path.join(path_,fil) ,sr=16000)
            data.append(data_contain)
            sample.append(sample_rate)
            label.append(file)
    return data,label,sample
```

[145]:
```
data,label,sample=load(dataset)
df=pd.DataFrame()
df['Label'],df['sample']=label,sample
df
```

[145]:

	Label	sample
0	bed	16000
1	bed	16000
2	bed	16000
3	bed	16000
4	bed	16000
...	...	...
41844	zero	16000
41845	zero	16000
41846	zero	16000
41847	zero	16000
41848	zero	16000

41849 rows × 2 columns



```
[156]: code={}
       x=0
       for i in pd.unique(label):
           code[i]=x
           x+=1
       pd.DataFrame(code.values(),columns=['Value'],index=code.keys())
```

```
C:\Users\TANISH\AppData\Local\Temp\ipykernel_17028\3153399070.py:3: FutureWarning: unique with argument that is not not a Series, Index, ExtensionArra
y, or np.ndarray is deprecated and will raise in a future version.
  for i in pd.unique(label):
```

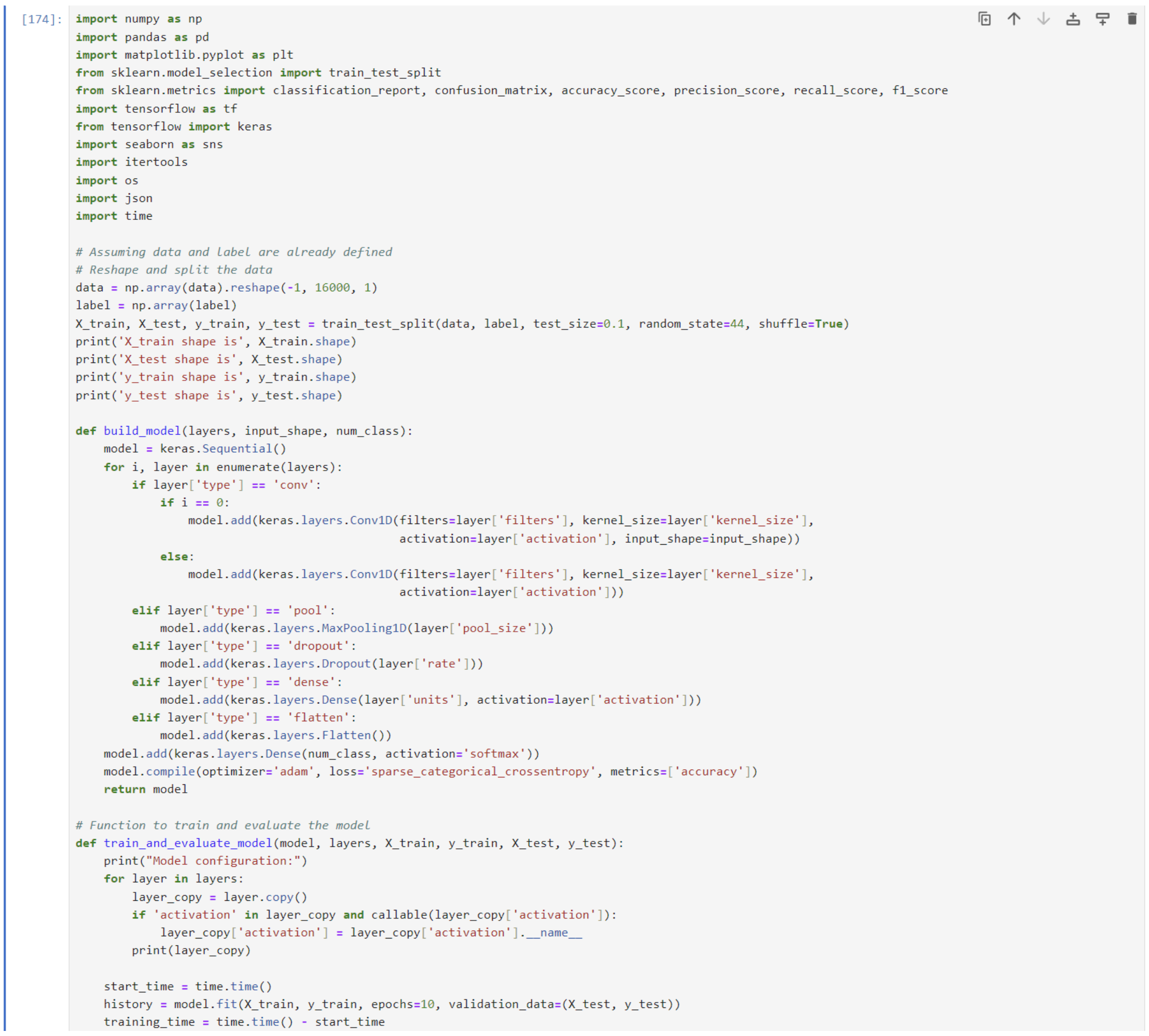
[156]:

	Value
bed	0
bird	1
cat	2
dog	3
down	4
eight	5
five	6
four	7
go	8
happy	9
house	10
left	11
marvel	12
nine	13
no	14
off	15
on	16
one	17
right	18
seven	19
sheila	20
six	21
stop	22
three	23
tree	24
two	25
up	26

```
  six   21
 stop   22
three   23
 tree   24
  two   25
   up   26
  wow   27
  yes   28
 zero   29
```

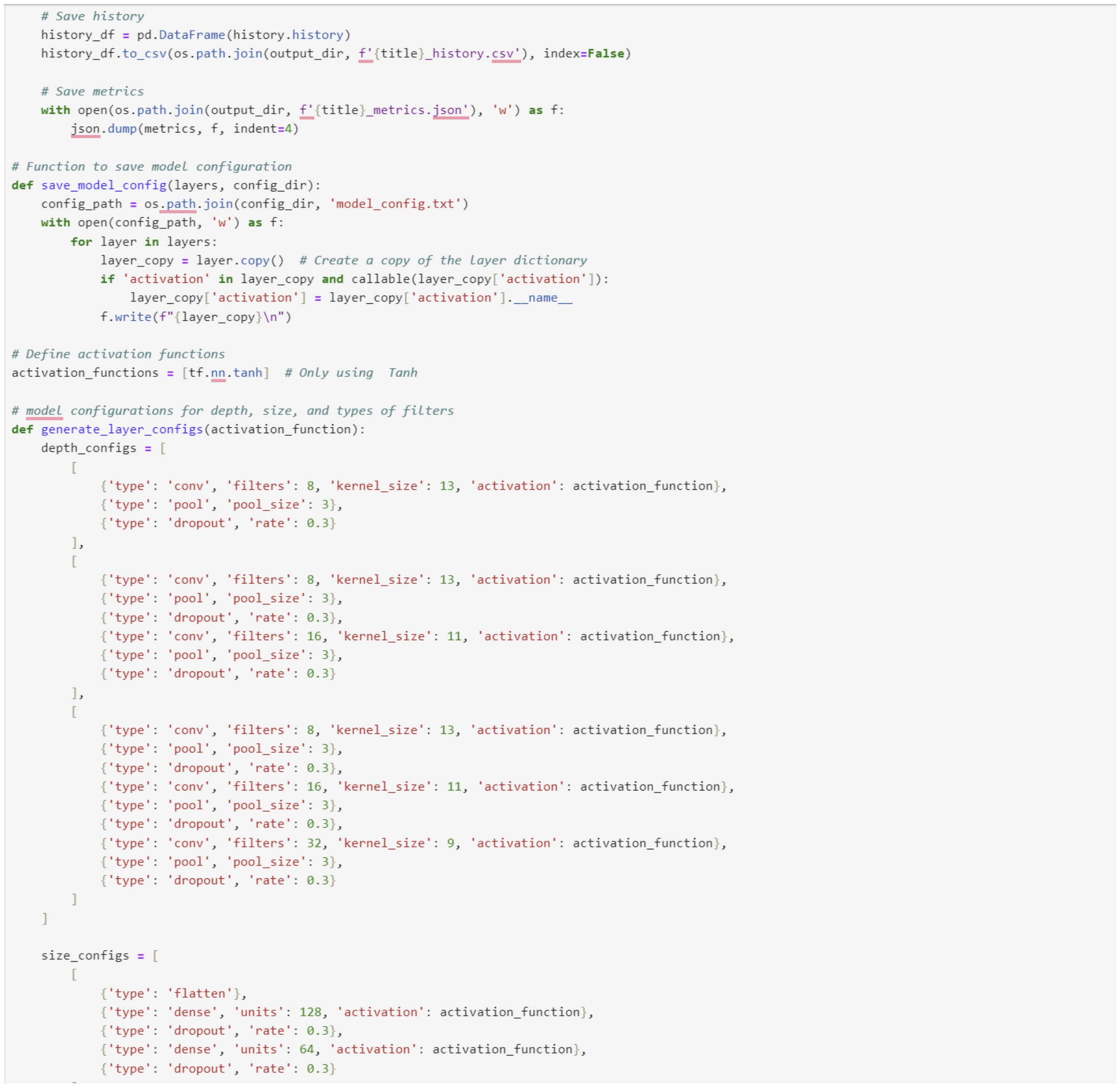
[157]:
```python
def get_Name(N):
    for x,y in code.items():
          if y==N:
                return x
for i in range(len(label)):
    label[i]=code[label[i]]
pd.DataFrame(label,columns=['Labels'])
```

[157]:
```
        Labels
    0        0
    1        0
    2        0
    3        0
    4        0
  ...      ...
41844       29
41845       29
41846       29
41847       29
41848       29

41849 rows × 1 columns
```

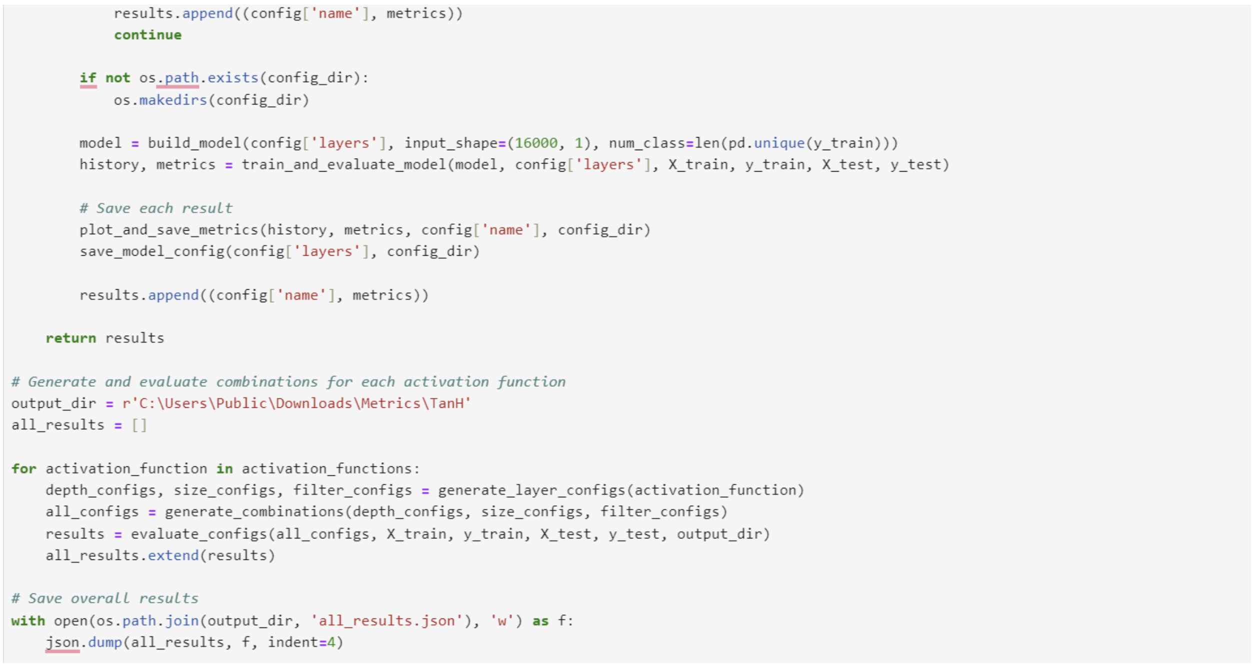


```
[174]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score, precision_score, recall_score, f1_score
import tensorflow as tf
from tensorflow import keras
import seaborn as sns
import itertools
import os
import json
import time

# Assuming data and label are already defined
# Reshape and split the data
data = np.array(data).reshape(-1, 16000, 1)
label = np.array(label)
X_train, X_test, y_train, y_test = train_test_split(data, label, test_size=0.1, random_state=44, shuffle=True)
print('X_train shape is', X_train.shape)
print('X_test shape is', X_test.shape)
print('y_train shape is', y_train.shape)
print('y_test shape is', y_test.shape)

def build_model(layers, input_shape, num_class):
    model = keras.Sequential()
    for i, layer in enumerate(layers):
        if layer['type'] == 'conv':
            if i == 0:
                model.add(keras.layers.Conv1D(filters=layer['filters'], kernel_size=layer['kernel_size'],
                                              activation=layer['activation'], input_shape=input_shape))
            else:
                model.add(keras.layers.Conv1D(filters=layer['filters'], kernel_size=layer['kernel_size'],
                                              activation=layer['activation']))
        elif layer['type'] == 'pool':
            model.add(keras.layers.MaxPooling1D(layer['pool_size']))
        elif layer['type'] == 'dropout':
            model.add(keras.layers.Dropout(layer['rate']))
        elif layer['type'] == 'dense':
            model.add(keras.layers.Dense(layer['units'], activation=layer['activation']))
        elif layer['type'] == 'flatten':
            model.add(keras.layers.Flatten())
    model.add(keras.layers.Dense(num_class, activation='softmax'))
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model

# Function to train and evaluate the model
def train_and_evaluate_model(model, layers, X_train, y_train, X_test, y_test):
    print("Model configuration:")
    for layer in layers:
        layer_copy = layer.copy()
        if 'activation' in layer_copy and callable(layer_copy['activation']):
            layer_copy['activation'] = layer_copy['activation'].__name__
        print(layer_copy)

    start_time = time.time()
    history = model.fit(X_train, y_train, epochs=10, validation_data=(X_test, y_test))
    training_time = time.time() - start_time
```

```python
    training_time = time.time() - start_time

    y_pred = np.argmax(model.predict(X_test), axis=-1)

    metrics = {
        'accuracy': accuracy_score(y_test, y_pred),
        'precision': precision_score(y_test, y_pred, average='weighted'),
        'recall': recall_score(y_test, y_pred, average='weighted'),
        'f1_score': f1_score(y_test, y_pred, average='weighted'),
        'classification_report': classification_report(y_test, y_pred, output_dict=True),
        'confusion_matrix': confusion_matrix(y_test, y_pred).tolist(),
        'training_time': training_time,
        'validation_accuracy': history.history['val_accuracy'],
        'validation_loss': history.history['val_loss']
    }

    return history, metrics

# Function to plot and save metrics
def plot_and_save_metrics(history, metrics, title, output_dir):
    # Create directory if it doesn't exist
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    fig, axs = plt.subplots(4, 1, figsize=(12, 24))

    # Plot training & validation accuracy values
    axs[0].plot(history.history['accuracy'])
    axs[0].plot(history.history['val_accuracy'])
    axs[0].set_title(f'{title} Accuracy')
    axs[0].set_ylabel('Accuracy')
    axs[0].set_xlabel('Epoch')
    axs[0].legend(['Train', 'Validation'], loc='upper left')

    # Plot training & validation loss values
    axs[1].plot(history.history['loss'])
    axs[1].plot(history.history['val_loss'])
    axs[1].set_title(f'{title} Loss')
    axs[1].set_ylabel('Loss')
    axs[1].set_xlabel('Epoch')
    axs[1].legend(['Train', 'Validation'], loc='upper left')

    # Plot confusion matrix
    sns.heatmap(metrics['confusion_matrix'], annot=True, fmt="d", ax=axs[2])
    axs[2].set_title(f'{title} Confusion Matrix')
    axs[2].set_xlabel('Predicted')
    axs[2].set_ylabel('True')

    # Display classification report
    report_df = pd.DataFrame(metrics['classification_report']).transpose()
    sns.heatmap(report_df.iloc[:-1, :-1], annot=True, fmt=".2f", cmap="Blues", ax=axs[3])
    axs[3].set_title(f'{title} Classification Report')

    # Save plot
    plt.savefig(os.path.join(output_dir, f'{title}_metrics.png'))
    plt.close()
```



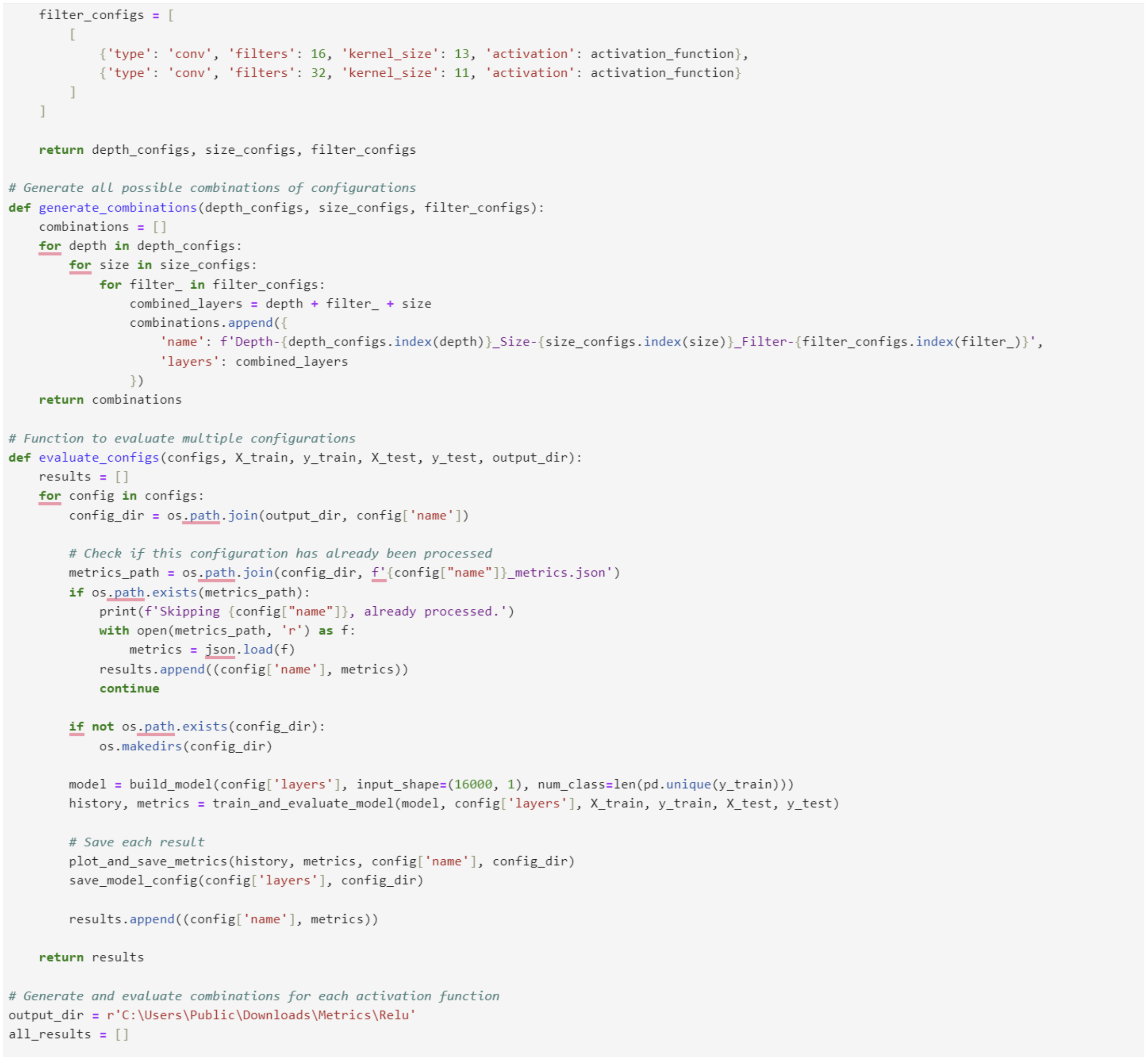
```python
    # Save history
    history_df = pd.DataFrame(history.history)
    history_df.to_csv(os.path.join(output_dir, f'{title}_history.csv'), index=False)

    # Save metrics
    with open(os.path.join(output_dir, f'{title}_metrics.json'), 'w') as f:
        json.dump(metrics, f, indent=4)

# Function to save model configuration
def save_model_config(layers, config_dir):
    config_path = os.path.join(config_dir, 'model_config.txt')
    with open(config_path, 'w') as f:
        for layer in layers:
            layer_copy = layer.copy()  # Create a copy of the layer dictionary
            if 'activation' in layer_copy and callable(layer_copy['activation']):
                layer_copy['activation'] = layer_copy['activation'].__name__
            f.write(f"{layer_copy}\n")

# Define activation functions
activation_functions = [tf.nn.tanh]  # Only using  Tanh

# model configurations for depth, size, and types of filters
def generate_layer_configs(activation_function):
    depth_configs = [
        [
            {'type': 'conv', 'filters': 8, 'kernel_size': 13, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3}
        ],
        [
            {'type': 'conv', 'filters': 8, 'kernel_size': 13, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'conv', 'filters': 16, 'kernel_size': 11, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3}
        ],
        [
            {'type': 'conv', 'filters': 8, 'kernel_size': 13, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'conv', 'filters': 16, 'kernel_size': 11, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'conv', 'filters': 32, 'kernel_size': 9, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3}
        ]
    ]

    size_configs = [
        [
            {'type': 'flatten'},
            {'type': 'dense', 'units': 128, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'dense', 'units': 64, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3}
        ],
        [
            {'type': 'flatten'},
            {'type': 'dense', 'units': 256, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'dense', 'units': 128, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3}
        ],
        [
            {'type': 'flatten'},
            {'type': 'dense', 'units': 512, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'dense', 'units': 256, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3}
        ]
    ]

    filter_configs = [
        [
            {'type': 'conv', 'filters': 16, 'kernel_size': 13, 'activation': activation_function},
            {'type': 'conv', 'filters': 32, 'kernel_size': 11, 'activation': activation_function}
        ]
    ]

    return depth_configs, size_configs, filter_configs

# Generate all possible combinations of configurations
def generate_combinations(depth_configs, size_configs, filter_configs):
    combinations = []
    for depth in depth_configs:
        for size in size_configs:
            for filter_ in filter_configs:
                combined_layers = depth + filter_ + size
                combinations.append({
                    'name': f'Depth-{depth_configs.index(depth)}_Size-{size_configs.index(size)}_Filter-{filter_configs.index(filter_)}',
                    'layers': combined_layers
                })
    return combinations

# Function to evaluate multiple configurations
def evaluate_configs(configs, X_train, y_train, X_test, y_test, output_dir):
    results = []
    for config in configs:
        config_dir = os.path.join(output_dir, config['name'])

        # Check if this configuration has already been processed
        metrics_path = os.path.join(config_dir, f'{config["name"]}_metrics.json')
        if os.path.exists(metrics_path):
            print(f'Skipping {config["name"]}, already processed.')
            with open(metrics_path, 'r') as f:
                metrics = json.load(f)
            results.append((config['name'], metrics))
            continue

        if not os.path.exists(config_dir):
            os.makedirs(config_dir)
```



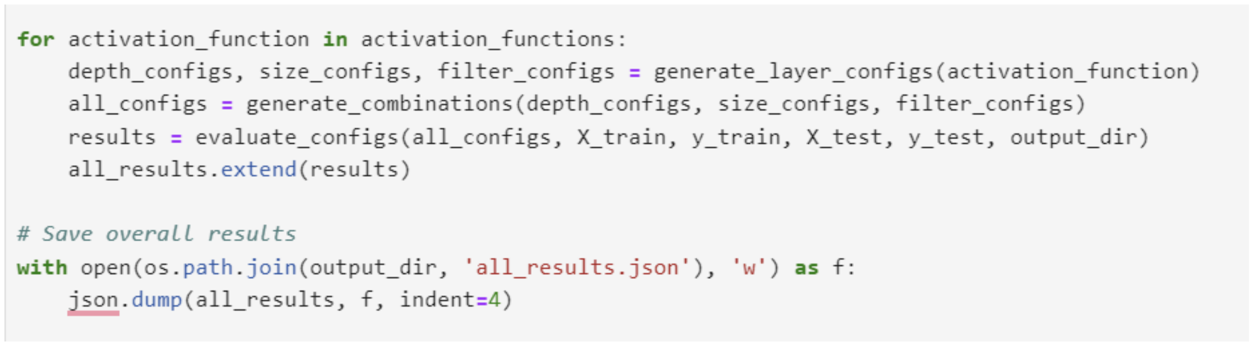
```
            results.append((config['name'], metrics))
            continue

        if not os.path.exists(config_dir):
            os.makedirs(config_dir)

        model = build_model(config['layers'], input_shape=(16000, 1), num_class=len(pd.unique(y_train)))
        history, metrics = train_and_evaluate_model(model, config['layers'], X_train, y_train, X_test, y_test)

        # Save each result
        plot_and_save_metrics(history, metrics, config['name'], config_dir)
        save_model_config(config['layers'], config_dir)

        results.append((config['name'], metrics))

    return results

# Generate and evaluate combinations for each activation function
output_dir = r'C:\Users\Public\Downloads\Metrics\TanH'
all_results = []

for activation_function in activation_functions:
    depth_configs, size_configs, filter_configs = generate_layer_configs(activation_function)
    all_configs = generate_combinations(depth_configs, size_configs, filter_configs)
    results = evaluate_configs(all_configs, X_train, y_train, X_test, y_test, output_dir)
    all_results.extend(results)

# Save overall results
with open(os.path.join(output_dir, 'all_results.json'), 'w') as f:
    json.dump(all_results, f, indent=4)
```

The above code was used for TanH. This was adjusted for ReLU and Leaky ReLU like the following (only the code from the define activation function was adjusted):

```python
# Define activation functions
activation_functions = [tf.nn.relu]  # Only using  relu

# model configurations for depth, size, and types of filters
def generate_layer_configs(activation_function):
    depth_configs = [
        [
            {'type': 'conv', 'filters': 8, 'kernel_size': 13, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3}
        ],
        [
            {'type': 'conv', 'filters': 8, 'kernel_size': 13, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'conv', 'filters': 16, 'kernel_size': 11, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3}
        ],
        [
            {'type': 'conv', 'filters': 8, 'kernel_size': 13, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'conv', 'filters': 16, 'kernel_size': 11, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'conv', 'filters': 32, 'kernel_size': 9, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3}
        ]
    ]

    size_configs = [
        [
            {'type': 'flatten'},
            {'type': 'dense', 'units': 128, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'dense', 'units': 64, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3}
        ],
        [
            {'type': 'flatten'},
            {'type': 'dense', 'units': 256, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'dense', 'units': 128, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3}
        ],
        [
            {'type': 'flatten'},
            {'type': 'dense', 'units': 512, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'dense', 'units': 256, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3}
        ]
    ]

    filter_configs = [
```



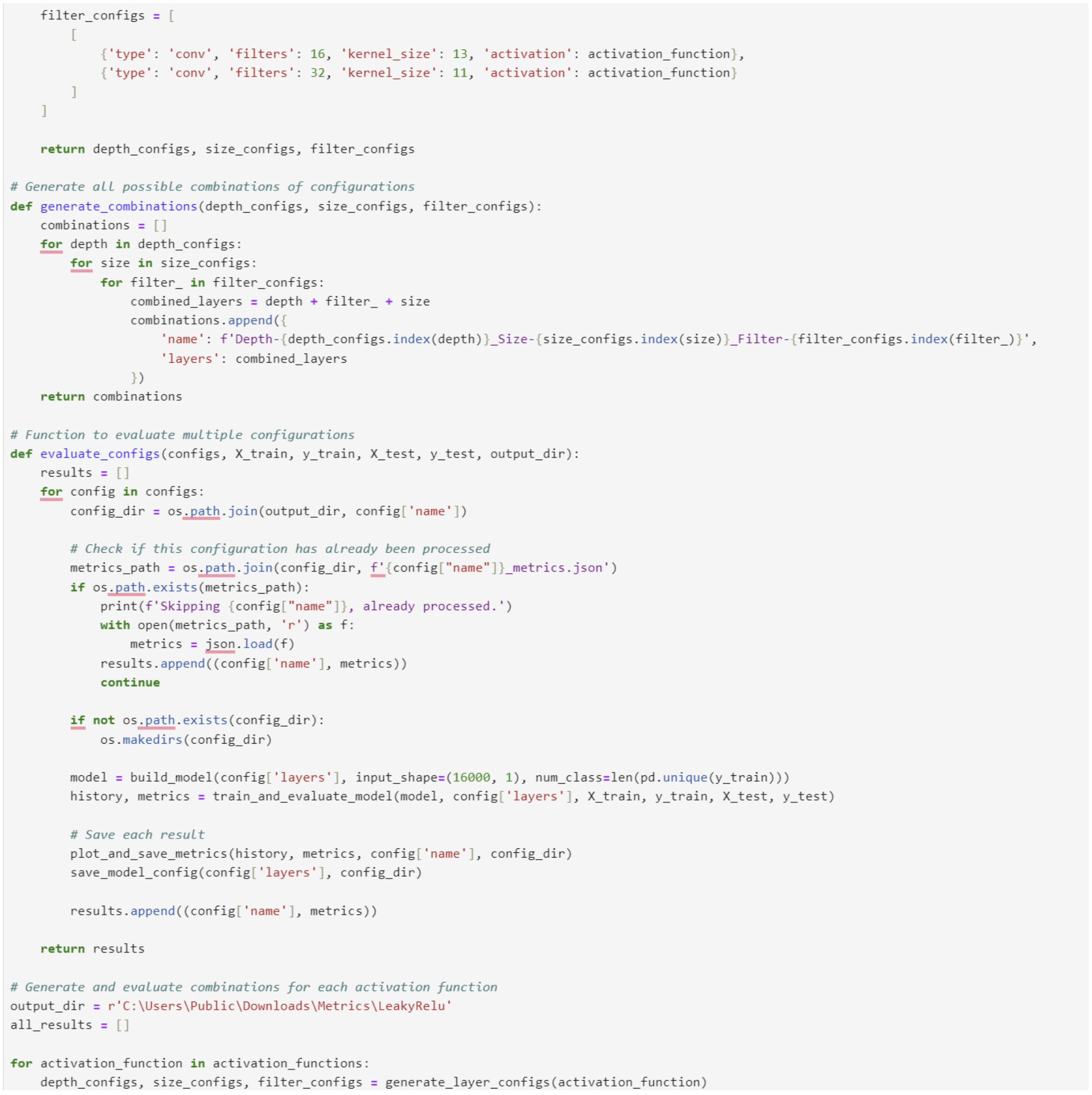
```python
    filter_configs = [
        [
            {'type': 'conv', 'filters': 16, 'kernel_size': 13, 'activation': activation_function},
            {'type': 'conv', 'filters': 32, 'kernel_size': 11, 'activation': activation_function}
        ]
    ]

    return depth_configs, size_configs, filter_configs

# Generate all possible combinations of configurations
def generate_combinations(depth_configs, size_configs, filter_configs):
    combinations = []
    for depth in depth_configs:
        for size in size_configs:
            for filter_ in filter_configs:
                combined_layers = depth + filter_ + size
                combinations.append({
                    'name': f'Depth-{depth_configs.index(depth)}_Size-{size_configs.index(size)}_Filter-{filter_configs.index(filter_)}',
                    'layers': combined_layers
                })
    return combinations

# Function to evaluate multiple configurations
def evaluate_configs(configs, X_train, y_train, X_test, y_test, output_dir):
    results = []
    for config in configs:
        config_dir = os.path.join(output_dir, config['name'])

        # Check if this configuration has already been processed
        metrics_path = os.path.join(config_dir, f'{config["name"]}_metrics.json')
        if os.path.exists(metrics_path):
            print(f'Skipping {config["name"]}, already processed.')
            with open(metrics_path, 'r') as f:
                metrics = json.load(f)
            results.append((config['name'], metrics))
            continue

        if not os.path.exists(config_dir):
            os.makedirs(config_dir)

        model = build_model(config['layers'], input_shape=(16000, 1), num_class=len(pd.unique(y_train)))
        history, metrics = train_and_evaluate_model(model, config['layers'], X_train, y_train, X_test, y_test)

        # Save each result
        plot_and_save_metrics(history, metrics, config['name'], config_dir)
        save_model_config(config['layers'], config_dir)

        results.append((config['name'], metrics))

    return results

# Generate and evaluate combinations for each activation function
output_dir = r'C:\Users\Public\Downloads\Metrics\Relu'
all_results = []
```



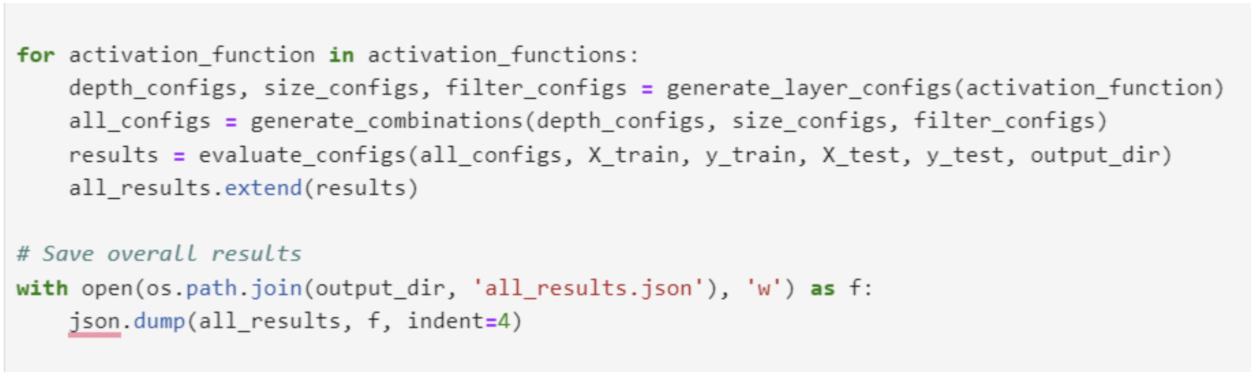
```python
for activation_function in activation_functions:
    depth_configs, size_configs, filter_configs = generate_layer_configs(activation_function)
    all_configs = generate_combinations(depth_configs, size_configs, filter_configs)
    results = evaluate_configs(all_configs, X_train, y_train, X_test, y_test, output_dir)
    all_results.extend(results)

# Save overall results
with open(os.path.join(output_dir, 'all_results.json'), 'w') as f:
    json.dump(all_results, f, indent=4)
```

For Leaky Relu:

```
# Define activation functions
activation_functions = [tf.nn.leaky_relu]  # Only using  relu

# model configurations for depth, size, and types of filters
def generate_layer_configs(activation_function):
    depth_configs = [
        [
            {'type': 'conv', 'filters': 8, 'kernel_size': 13, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3}
        ],
        [
            {'type': 'conv', 'filters': 8, 'kernel_size': 13, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'conv', 'filters': 16, 'kernel_size': 11, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3}
        ],
        [
            {'type': 'conv', 'filters': 8, 'kernel_size': 13, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'conv', 'filters': 16, 'kernel_size': 11, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'conv', 'filters': 32, 'kernel_size': 9, 'activation': activation_function},
            {'type': 'pool', 'pool_size': 3},
            {'type': 'dropout', 'rate': 0.3}
        ]
    ]

    size_configs = [
        [
            {'type': 'flatten'},
            {'type': 'dense', 'units': 128, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'dense', 'units': 64, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3}
        ],
        [
            {'type': 'flatten'},
            {'type': 'dense', 'units': 256, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'dense', 'units': 128, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3}
        ],
        [
            {'type': 'flatten'},
            {'type': 'dense', 'units': 512, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3},
            {'type': 'dense', 'units': 256, 'activation': activation_function},
            {'type': 'dropout', 'rate': 0.3}
        ]
    ]

    filter_configs = [
```



```
    filter_configs = [
        [
            {'type': 'conv', 'filters': 16, 'kernel_size': 13, 'activation': activation_function},
            {'type': 'conv', 'filters': 32, 'kernel_size': 11, 'activation': activation_function}
        ]
    ]

    return depth_configs, size_configs, filter_configs

# Generate all possible combinations of configurations
def generate_combinations(depth_configs, size_configs, filter_configs):
    combinations = []
    for depth in depth_configs:
        for size in size_configs:
            for filter_ in filter_configs:
                combined_layers = depth + filter_ + size
                combinations.append({
                    'name': f'Depth-{depth_configs.index(depth)}_Size-{size_configs.index(size)}_Filter-{filter_configs.index(filter_)}',
                    'layers': combined_layers
                })
    return combinations

# Function to evaluate multiple configurations
def evaluate_configs(configs, X_train, y_train, X_test, y_test, output_dir):
    results = []
    for config in configs:
        config_dir = os.path.join(output_dir, config['name'])

        # Check if this configuration has already been processed
        metrics_path = os.path.join(config_dir, f'{config["name"]}_metrics.json')
        if os.path.exists(metrics_path):
            print(f'Skipping {config["name"]}, already processed.')
            with open(metrics_path, 'r') as f:
                metrics = json.load(f)
            results.append((config['name'], metrics))
            continue

        if not os.path.exists(config_dir):
            os.makedirs(config_dir)

        model = build_model(config['layers'], input_shape=(16000, 1), num_class=len(pd.unique(y_train)))
        history, metrics = train_and_evaluate_model(model, config['layers'], X_train, y_train, X_test, y_test)

        # Save each result
        plot_and_save_metrics(history, metrics, config['name'], config_dir)
        save_model_config(config['layers'], config_dir)

        results.append((config['name'], metrics))

    return results

# Generate and evaluate combinations for each activation function
output_dir = r'C:\Users\Public\Downloads\Metrics\LeakyRelu'
all_results = []

for activation_function in activation_functions:
    depth_configs, size_configs, filter_configs = generate_layer_configs(activation_function)
```



```
for activation_function in activation_functions:
    depth_configs, size_configs, filter_configs = generate_layer_configs(activation_function)
    all_configs = generate_combinations(depth_configs, size_configs, filter_configs)
    results = evaluate_configs(all_configs, X_train, y_train, X_test, y_test, output_dir)
    all_results.extend(results)

# Save overall results
with open(os.path.join(output_dir, 'all_results.json'), 'w') as f:
    json.dump(all_results, f, indent=4)
```