DEEP LEARNING IN AUDIO PROCESSING

HOW DO CONVOLUTIONAL NEURAL NETWORKS COMPARE TO RECURRENT NEURAL NETWORKS IN TERMS OF SPEED AND ACCURACY WHEN PERFORMING SPEECH RECOGNITION?

Group 4: Experimental Sciences--Category 4

Word Count: 3,995

CS EE World
https://cseeworld.wixsite.com/home
May 2021
27/34
A

Submitter Info: Anonymous

Table of Contents

1. Introduction 4

1.1 Societal Applications and Importance of Speech Recognition 4

1.2 Computational Approach to Audio Waves and Audio Processing Tools 4

1.3 The Advantages of Deep Learning for Speech Recognition 7

2. Theoretical Background 8

2.1 Neural Networks 8

2.2 Convolutional Neural Networks (CNNs) 11

2.3 Recurrent Neural Networks (RNNs) 13

3. Experiment Methodology 16

3.1 Programming Platform, Language, and Libraries 16

3.2 The Dataset 17

3.3 CNN Preprocessing and Model Architecture 17

3.4 RNN Preprocessing and Model Architecture 19

3.5 The Independent Variable 20

3.6 The Dependent Variables 21

3.7 The Hypothesis 21

4. Experiment Result Analysis and Conclusion 22

4.1 CNN Results 22

4.2 RNN Results 23

4.3 Comparison and Analysis 24

4.4 Conclusion 25

Works Cited 27

Appendices 30

Python Code for the CNN 30

Terminal Output for the CNN 33

Python Code for the RNN 39

Terminal Output for the RNN 42

1. Introduction

1.1 Societal Applications and Importance of Speech Recognition

Speech recognition technology has been influential to society for over a decade. The most obvious application of speech recognition is digital voice assistants such as Siri (Apple) and Alexa (Amazon). Integrated into many people's lives through smartphones and home assistance devices, virtual assistants help out with digital tasks like web searches, messages, and calls, as well as smart tasks such as daily reminders and health tracking. Advanced speech recognition software is a big factor in the convenience of digital voice assistants.

Other speech recognition systems integrated into different platforms also allow for similar convenience. For example, Google and YouTube search both have a "search by voice" option for those who type slowly, or are unable to type due to mental or physical challenges. Google docs provides a "voice typing" option that speeds up the process of writing papers. Google translate, Duolingo and other foreign language learning platforms utilize speech recognition to analyze and enhance students' oral speaking skills.

1.2 Computational Approach to Audio Waves and Audio Processing Tools

Automatic speech recognition systems have been around since the beginning of the 21st century. However, it is only in recent years that speech recognition software has made significant advancements and gained popular traction, thanks to the deep learning (DL) subset of artificial intelligence (AI). The concept of speech recognition is

simple; a computer converts the spoken audio detected by the microphone into written text. However, this process is much more complicated in practice.

To begin with, sound is originally in the form of an analog wave picked up by the microphone, representing the longitudinal waves that move through the air. For computational use, the analog wave is converted into a series of digital values through a process called analog to digital conversion. The resulting array of binary numbers sampled at the correct rate give a relatively accurate representation of the original sound wave. (Anvarjon)

There are many ways to represent sound. Most commonly, amplitude-time graphs are used to display audio files in audio editing softwares (Figure 0). In these graphs, the amplitude, which corresponds to the intensity, is scaled and plotted against time (in seconds). In comparison, spectrograms plot frequency against time, and use a range of colors to display sound intensity at different frequencies (Figure 0.5). Every representation of audio provides some valuable information, but no single graph can show every property of a sound wave. Hence, each of these graphs are often used in a specific audio processing application. (“Understanding”)

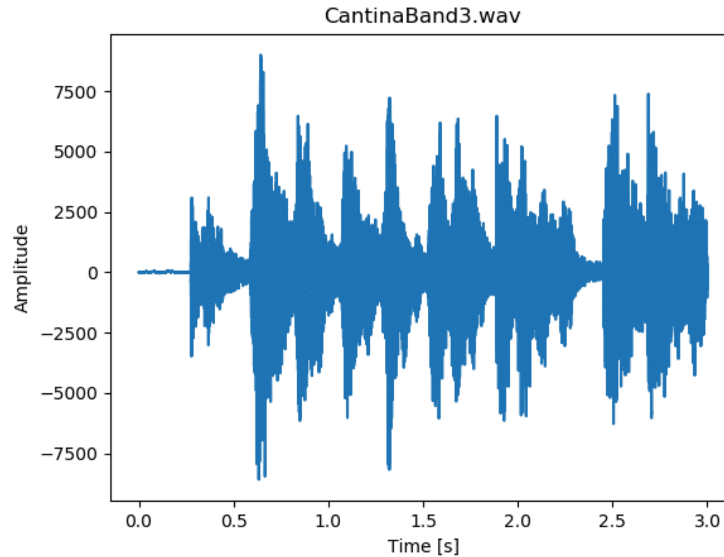


Figure 0. Amplitude-Time graph of a .wav audio file[1]

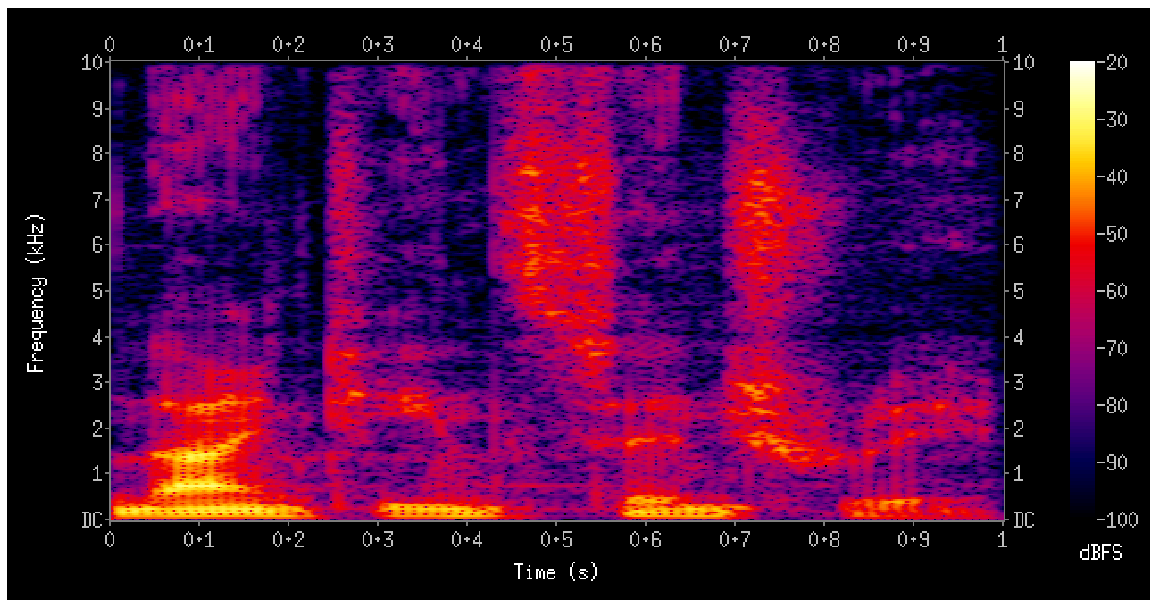


Figure 0.5. Spectrogram with a dBFS range of -100 to -20 (dark purple to light yellow), a 10 kHz maximum frequency, and 100 millisecond time intervals[2]

[1] Nathan, Mathura. "Plot Audio File as Time Series Using Scipy Python." *GaussianWaves*, 2 Aug. 2020, www.gaussianwaves.com/2020/01/how-to-plot-audio-files-as-time-series-using-scipy-python/.

[2] "Understanding Spectrograms." *IZotope*, 11 Apr. 2019, www.izotope.com/en/learn/understanding-spectrograms.html.

1.3 The Advantages of Deep Learning for Speech Recognition

The reason DL methods involving neural networks are preferred over traditional speech recognition algorithms is because of the efficiency and flexibility of DL. Due to their nature, neural networks can successfully identify spoken syllables and words with a range of variation, which may be caused by difference in voice, accent, or simply mispronunciation. Additionally, thanks to advancements in the programming language python and many of the cloud based integrated development environments (IDEs), it has become exceptionally easy to code neural networks with less powerful computers, thus making the application of DL more convenient.

One question that comes up while planning to develop a speech recognition system using deep learning is “what type of neural network is best for the task?” Some of the most commonly used neural networks for speech recognition are convolutional and recurrent neural networks. In this extended essay, I will be describing them and experimenting with both to ultimately answer the question of which is more accurate and faster for speech recognition: convolutional neural networks or recurrent neural networks?

2. Theoretical Background

2.1 Neural Networks

Neural networks (NNs) are a subcategory of machine learning (ML) that deal with the tasks of classifying or creating data by making a model that learns from particular sets of inputs. NNs consist of individual nodes connected in multiple layers, each of which manipulate the data as it passes from the input layer, through the hidden layer, and into the output layer (An). At a given layer, the nodes of a NN weigh the input data, sum up all of the values, add a bias, and pass the result through an activation function and into the next layer (Figure 1). From a mathematical approach, this process can be demonstrated using matrix operations. The input matrix is multiplied with the weights matrix, and added with the bias matrix (Jordan). Each result is passed through an activation function (eg. Sigmoid, ReLu, Softmax) to create the output matrix (Equation 1).

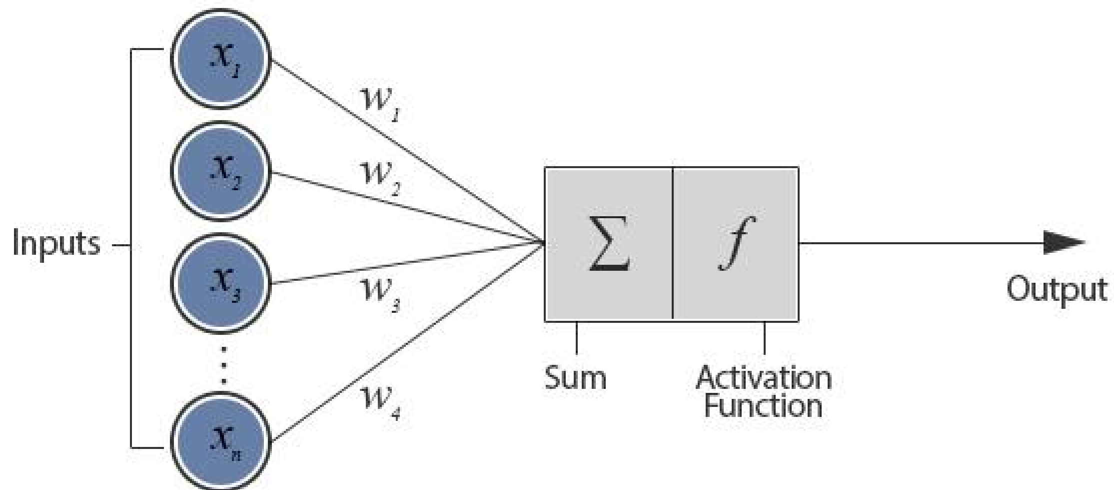
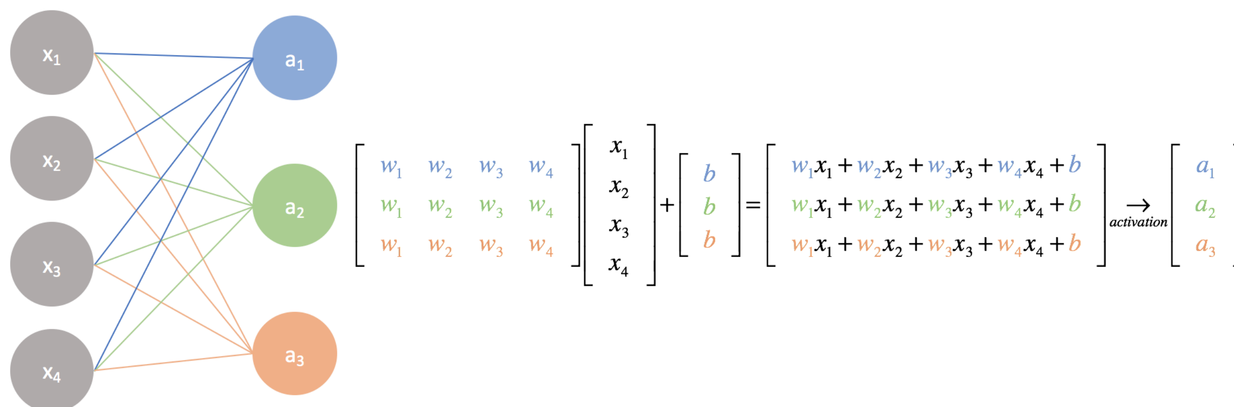


Figure 1. Individual neuron of a Feed Forward NN[3]

[3] An, Sungtae. "Feedforward Neural Networks." *Sungtae's Awesome Homepage*, Georgia Institute of Technology, 8 Oct. 2017, www.cc.gatech.edu/~san37/post/dlhc-fnn/.



Equation 1. The matrix operations that demonstrate the workings of a Feed Forward layer[4]

This is the foundational idea behind a simple “feed forward” neural network model. The NN “learns” by adjusting the weights throughout layers until the outputs are accurate with respect to what is expected as a correct output (Jordan). Each weight represents the so-called importance or influence of the data which passes through that point, and can be increased or decreased as the neural network sees fit (An). A larger weight places more significance on the information for that input, and vice versa. Each node also has a bias, which is a constant added to the weighted input. The bias at each node for every layer can be adjusted as well to give the neural network more precision (An). The activation function takes the resultant value (which is the summation of the weighted inputs and the biases), and outputs a value corresponding to the chosen function (Jordan). For example, a Sigmoid activation function will take any input and pass it through the function $1/(1 + e^{-x})$ so that the output value is between 0 and 1, whereas a ReLu activation function will convert negative inputs to zero but leave positive inputs unaltered (Figure 2).

[4] Jordan, Jeremy. “Neural Networks: Representation.” *Jeremy Jordan*, 26 Jan. 2018, www.jeremyjordan.me/intro-to-neural-networks/.

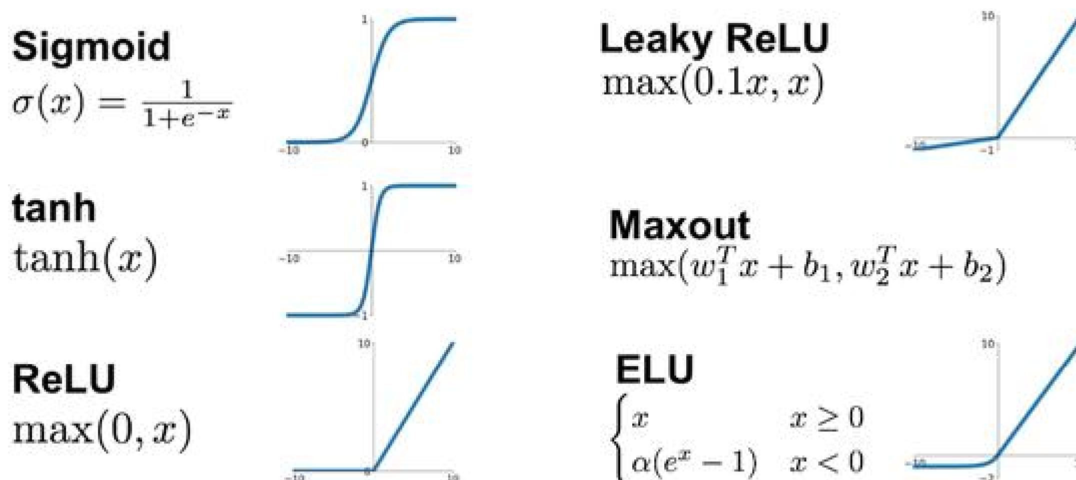


Figure 2. Some of the common activation functions[5]

The neural network passes the error for each run through a loss function, which generates a loss that is used to change the adjustable variables of the NN's layers (weights and biases) accordingly. It is important to note that the term "error" is defined as the deviation from an actual value by a prediction or expectation of that value, whereas "loss" is defined as a quantified measure of how consequential it is to get an error of a particular size or direction[6]. An example of a common loss function is Mean Squared Error (MSE), which squares the errors throughout the dataset and outputs the average, and is most commonly used in Linear Regression models ("Introduction"). The generated loss is passed through an optimizer algorithm, which adjusts the weights and biases of the neural network in order to minimize the loss (Nielsen). One of the most basic and heavily used optimizer algorithms is Gradient Descent (Figure 3)(Raschka). It

---

[5] "Introduction to Loss Functions." *Algorithmia Blog*, 28 Apr. 2021, algorithmia.com/blog/introduction-to-loss-functions#types-of-loss-functions.
[6] Errors and losses are explained in detail by Michael Nielsen in his book "Neural Networks and Deep Learning."

uses the first order derivative of the loss function, and calculates how the weights should be adjusted for the function to reach a minima (local or global).[7]

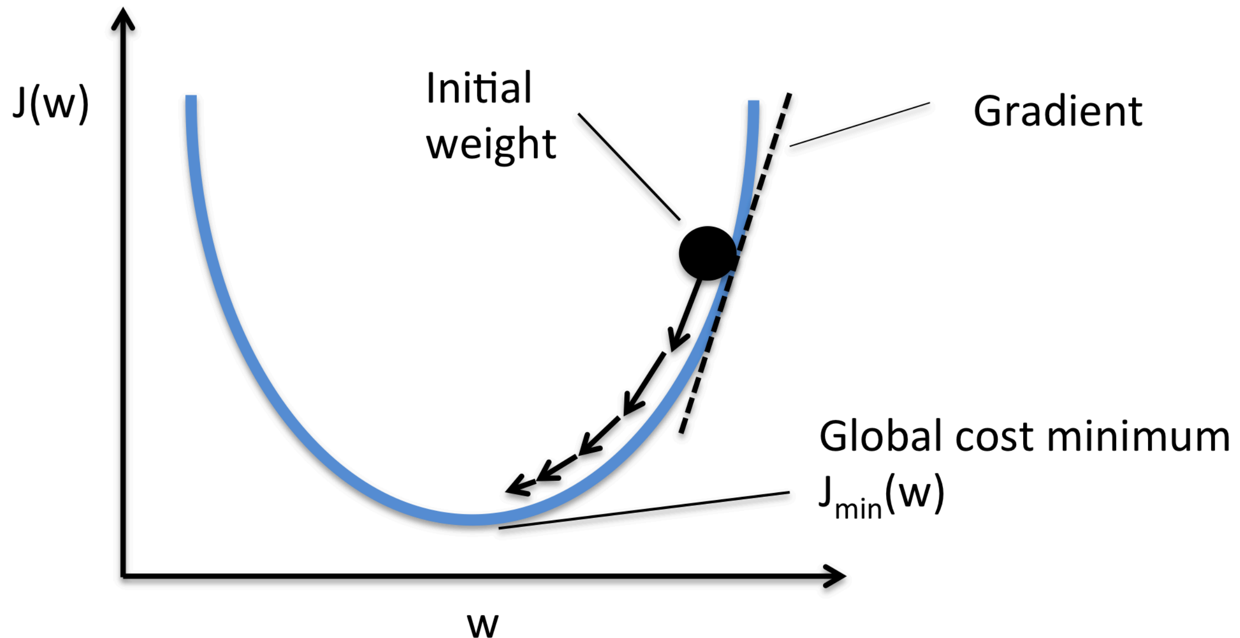


Figure 3. A graphical representation of the Gradient Descent optimizer algorithm's conceptual workings[8]

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have specialization for picking out patterns and deriving meaning from them, and thus they are used almost always for image analysis (Saha). A CNN differs from a traditional Multilayer Perceptron NN because of the convolutional layers. To begin with, most images come in either a 3D or 5D format: the height times the breadth (dimensions), and the number of color channels

---

[7] General information about NNs and DL can be found at https://static.latexstudio.net/article/2018/0912/neuralnetworksanddeeplearning.pdf

[8] Raschka, Sebastian. "Gradient Descent and Stochastic Gradient Descent." *Gradient Descent and Stochastic Gradient Descent - Mlxtend*, Mlxtend, 2014, rasbt.github.io/mlxtend/user_guide/general_concepts/gradient-optimization/.

(3 for RGB, 1 for grayscale). In grayscale, each pixel has a value between 0 and 255. Feeding this information into a standard Feed Forward NN is inconvenient because corresponding a node to each pixel overwhelms the computer and significantly slows down the training process. Furthermore, hardwiring each pixel to a node only works for identification of objects of the exact same scale and at the exact same position in the image. If the image was scaled by a small factor, or rotated by a small angle, the simple Feed Forward neural network would fail to classify it successfully.

In comparison, a convolutional layer uses simple filters (aka kernels), which are a grid of pixels with specific values adjusted throughout training, to identify special features in the image (like lines and curves). By sliding each filter through the image, from the top left to the bottom right corner, a CNN is able to capture the spatial and temporal properties of an image, thus classifying it more successfully, even with small changes such as scaling and rotating (Saha). The convolution operation works by lining up the filter with a same size patch in the image, multiplying the pixel values of the filter with the corresponding pixel values in the chosen patch of the image, summing the resultant values for each multiplication, and dividing the sum by the number of pixels (averaging). The result is stored in the location of the center pixel for a newly created image of corresponding size to the original, called the feature map, which is passed on to the next layer (Sewak). The pixel values of the individual filters are essentially the “weights” in a convolutional layer, and are adjusted through training.

Nevertheless, multiple convolution operations take a relatively long time to complete, so to scale down the data, CNNs also use pooling layers, which reduce the pixel count between convolutional layers (depending on the pooling algorithm). Some

common pooling options are Max Pooling (which picks out the maximum value in a small grid of pixels), Min Pooling (the opposite of the former), and Average Pooling (Saha). Finally, following the last convolutional layer, the CNN "Flattens" the data by changing it from a grid (which is the shape of the feature map) to an array format. The array is fed into a traditional Dense layer, consisting of an output layer that classifies the data (Figure 4). [9]

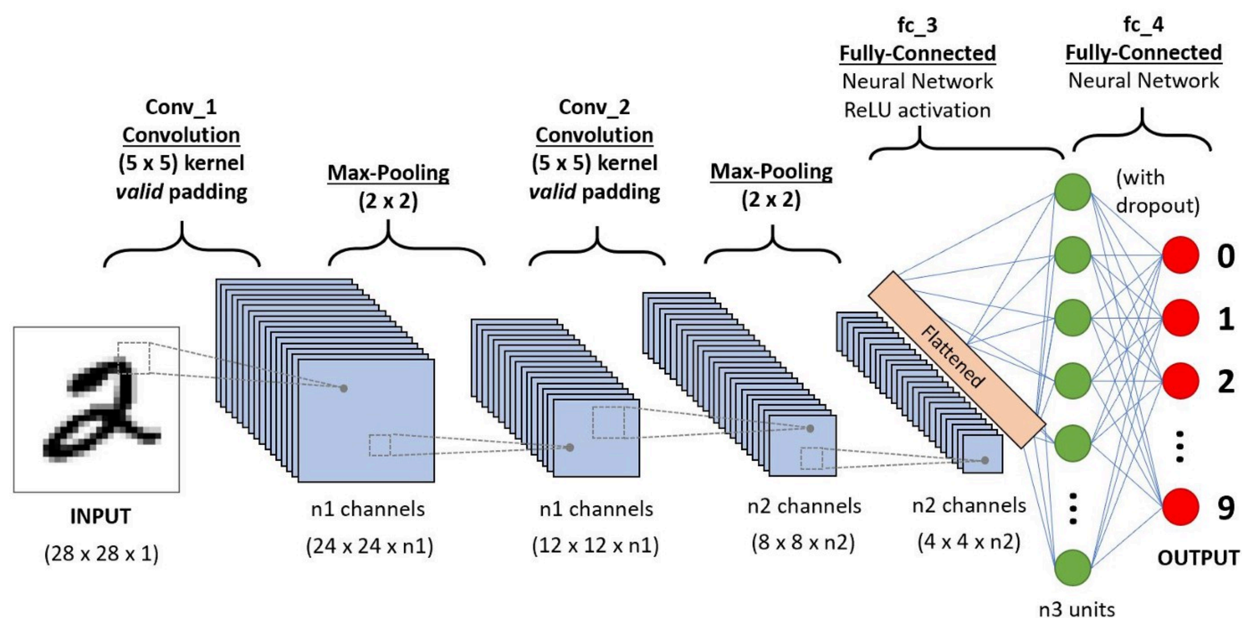


Figure 4. Visualization of a traditional CNN[10]

[9] The following book explains the types, workings, and applications of CNNs in depth: https://www.google.com/books/edition/Practical_Convolutional_Neural_Networks/bOlODwAAQBAJ?hl=en&gbpv=0

[10] Saha, Sumit. "A Comprehensive Guide to Convolutional Neural Networks - the eli5 Way." *Towards Data Science*, 17 Dec. 2018, towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

2.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) specialize in modeling sequential data through the use of memory. Thus, they're often used for audio and natural language processing (Venkatachalam). An RNN works by looping previous information forward to the next layers. In a Multilayer Perceptron, each neuron in a layer uses only the input data to produce an output. However, RNN layers also consider previous data that has passed through the network before producing an output. In addition to weighting the input data, the neurons weigh the previous data (Figure 5). This small difference allows RNNs to make sense of sequences much more efficiently.

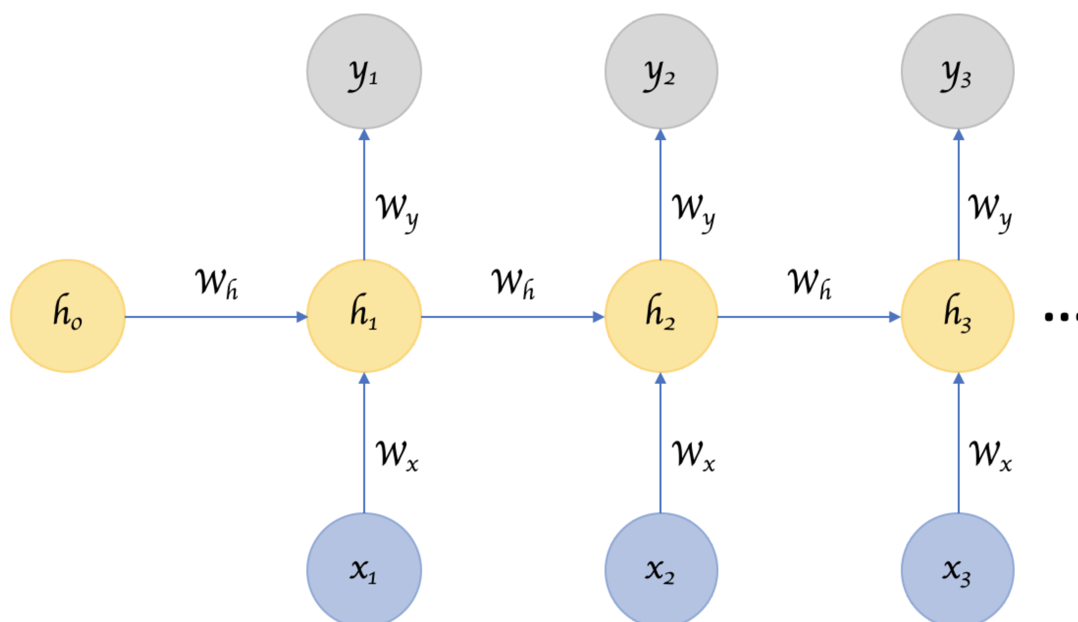


Figure 5. The structure of an RNN[11]

There are also multiple types of Recurrent Neural Networks. However, most of them suffer from short term memory, due to the vanishing gradient problem (which causes the earlier weights in the network to barely adjust through training due to the

[11] Venkatachalam, Mahendran. "Recurrent Neural Networks." *Towards Data Science*, 22 June 2019, towardsdatascience.com/recurrent-neural-networks-d4642c9bc7ce.

nature of backpropagation). Because of this, RNNs often fail to effectively use information earlier in the sequence to influence the final classification in the output layer. To combat this, Long-Short Term Memory (LSTM) neural networks, which are an evolved version of RNNs, are used (Kostadinov). LSTMs expand the scope of the memory and don't show a positive bias toward later information in a sequence.[12]

[12] Coding RNNs with Python: https://www.google.com/books/edition/Recurrent_Neural_Networks_with_Python_Qu/cC59DwAAQBAJ?hl=en&gbpv=0

3. Experiment Methodology

3.1 Programming Platform, Language, and Libraries

For the purposes of coding convenience and better visual presentations of processes, I used the Jupyter Notebook IDE in this experiment. Jupyter Notebook is free, open source, highly interactive, and advantageously structured for programming NNs. Each line of code can be programmed and run separately, but are consecutive in the execution of the written program. For example, one line can be used to import libraries, and another line to define a function. This way, the program is much easier to troubleshoot and modify. In the case of NNs, training a model usually takes a relatively long period of time. With the structure of Jupyter Notebook, changes in individual sections of the code can be made without re-training the NN.

I used the python programming language for this experiment. Python (v3) is the most common language in nearly all fields of machine learning, including deep learning. This is because of its high level and simple syntax, and the number of libraries that have been created to aid in the programming of NNs.

The most important library, however, is Tensorflow. It is developed and owned by Google, and used for both research and production. Tensorflow provides easy ways to define and create multilayer models with various types of layers, including deep, convolutional, and recurrent layers. Creating, training, and testing NNs with tensorflow is much easier than coding a NN from scratch. Going into how exactly tensorflow works is not necessary, as it is quite complicated and overwhelming, but essentially a layer in a model in tensorflow is a number of nodes representing mathematical operations in a graph, connected by tensors (multidimensional data arrays).

3.2 The Dataset

The dataset for this experiment consists of 3,000 audio samples of the spoken digits zero through nine. There are 6 different speakers, and each digit is repeated 50 times per speaker. The dataset is free for use online on GitHub[13], and is of course in English. The training and testing data are split 90% to 10%, meaning there are 2,700 samples for training and 300 for testing. Each sample is an approximately 1.5 second WAV audio file that is labeled as follows: {the digit spoken}_{name of the speaker}_{iteration of the digit}. There are also additional python programs included along with the dataset that trim the silence out of each sample, convert the WAV audio file into a grayscale JPEG spectrogram image of size 64 pixels by 64 pixels, and cross validate the data (split between testing and training). On a separate file, the metadata of the speakers are provided (including the name, gender, and accent).

3.3 CNN Preprocessing and Model Architecture

I used the 64x64, grayscale spectrogram pictures of the audio files as input to the CNN. A python function ‘create_train_data()’ defines the two arrays ‘training_data’ and ‘training_labels.’ Using a simple ‘for’ loop, for every image in the training directory, the function opens the image utilizing the PIL library (python interpreter with image editing capabilities) and saves each pixel value in the grayscale image to the ‘training_data’ array. Then, another function ‘label_img(img)’ is called to get the label (spoken digit) of the image, which is then added to the ‘training_labels’ array. The function returns the two arrays. The training data array is reshaped into the shape (2700, 64, 64, 1), with the

[13] https://github.com/Jakobovski/free-spoken-digit-dataset

2700 representing the number of images, the first 64 representing the number of rows of pixels, the second 64 representing the number of pixels per row, and the 1 representing the number of color dimensions (1 for a grayscale value between 0 and 255). This procedure is repeated for the testing portion of the dataset. Finally, each value in the training and testing data arrays are divided by 255 for the purposes of normalization. It is much easier for NNs to operate when all weights, variables, and inputs are within the same range. In order to achieve this, a process called normalization is used, where every value in a dataset is divided by the same constant so that each value is in a 0 to 1 range. In this case, 255 is the maximum value any of the pixels can have, so that is the constant.

The convolutional neural network is first defined as a sequential model (as most other NNs), which requires the 'keras' import from the tensorflow library. We begin by defining the number of filters in the layer (64), and the input shape of the image (64x64), along with the filter size and activation function. The filter size in this case is 3 by 3, which may sound small, but this is perfectly normal given that the input image is also very small and blurry. A smaller filter may lead to a much longer training period, and a larger filter may fail to train successfully, even through many iterations. The activation function used most often in convolutional layers is the rectified linear (ReLu) function. This is because ReLu converts negative values to zero, which makes sense because pixels shouldn't have negative values, but the function also preserves the positive values without converting them to either 1 or 0, as a sigmoid function would.

The convolutional layer is followed by a pooling layer, which has a grid size of 2 by 2 pixels. A pooling grid too large will result in the loss of many pixel values that may

come in handy for successful classification, but no pooling layer will make the training significantly slower.

There are three more convolutional layers of size 64 (neurons) in the NN, each with a 3 by 3 filter, a relu activation function, and a pooling layer afterwards (except for the last one). The last convolutional layer is followed by a flattening layer, which allows the data to be then passed on to a deep layer (of same size and activation function), and finally into an output layer of size 10; one neuron for each spoken digit.

The “Adam” optimizer is used, which is a stochastic gradient descent method that takes into account the first and second order moments before an estimation. The algorithm itself is complicated, but this optimizer performs much better than some others that I tested. The “Sparse Categorical Cross Entropy” loss function was used, which computes the crossentopy loss between the labels and predictions.[14]

3.4 RNN Preprocessing and Model Architecture

I chose to use the spectrograms of the audio samples as input to the RNN for a more fair comparison. Because of this, the preprocessing for the RNN is the same as that for the CNN. The only difference is that instead of a grid, the image is in an array format. Creating the model for the recurrent neural network is also a similar process. The model is first defined as sequential, then the individual recurrent layers are added. The arguments passed into the first recurrent layer are as follows: the number of neurons (128), the input data shape (64 by 64), the activation function (rectified linear), and a boolean called ‘return_sequences.’ As mentioned in section 2.3, RNNs are

[14] The code for my CNN is a modification of the code from this tutorial: https://colab.research.google.com/drive/1ZZXnCjFEOkp_KdNcNabd14yok0BAIuwS

special because they loop data. The ‘return_sequences’ variable lets the network know whether the current sequence of data should be kept for looping. If there is another recurrent layer after the current one, this boolean variable will be set true. This RNN consists of two recurrent layers, so this variable will be set true for the first layer. The second recurrent layer also has 128 neurons and a relu activation function. Next, there is a dense layer of 32 neurons with a ReLu activation function, and finally an output layer with a softmax activation function. The softmax function is used to ensure that all of the probabilities in the output layer combined add up to one, so that the output with the highest probability is chosen.

After every hidden layer, there is a “dropout” layer. Dropout is a technique used to limit overfitting. Though all types of neural networks are naturally prone to overfitting, RNNs and Deep Neural Networks (DNNs) are highly vulnerable, especially over many epochs. So, during every run, weights at a given layer are chosen at random and multiplied with the dropout constant. This puts less significance on those weights, and thus prohibits the NN from over-relying on them.

For the sake of a fair comparison, the optimizer and loss function for the RNN are the same as that of the CNN (the Adam optimizer, and the SCGD loss function).[15]

3.5 The Independent Variable

The only independent variable in this experiment that is shared by both neural networks is the number of epochs. The independent variables specific to the CNN are the filter

[15] The code for my RNN is a modification of the code from this tutorial: https://pythonprogramming.net/recurrent-neural-network-deep-learning-python-tensorflow-keras/

size and the pooling layer grid size. The independent variable specific to the RNN is the dropout constant.

3.6 The Dependent Variables

There are two dependent variables in this experiment: training duration and test dataset accuracy. Training duration is the amount of time the NN takes to train (calculated using the duration per epoch), and test dataset accuracy is the percentage accuracy of the model performing on the test dataset.

3.7 The Hypothesis

I have never before coded recurrent or convolutional neural networks, and therefore am not sure what to expect for the speed of each, in terms of the magnitude of time they will take to train (whether they will train in seconds, minutes, or hours). I am confident, however, that the RNN will train faster, because the CNN has more weights to adjust, due to the nature of its filters. In terms of accuracy, however, I believe that the two will be close, with the RNN slightly beating the CNN. I believe this because memory is important for audio classification, since audio is sequential data. Therefore, the absence of memory is a disadvantage to the CNN in my eyes.

4. Experiment Result Analysis and Conclusion

4.1 CNN Results

Following 10 epochs, the CNN yielded an accuracy of 94.67% over a training period of around 125 seconds. Following 16 epochs, the CNN yielded an accuracy of 96.67% over a training period of around 201 seconds. Following 24 epochs, the CNN yielded an accuracy of 96% over a training period of around 298 seconds.

```
Epoch 1/24
85/85 [==============================] - 13s 146ms/step - loss: 2.0011 - accuracy: 0.2567
Epoch 2/24
85/85 [==============================] - 13s 155ms/step - loss: 0.9295 - accuracy: 0.6730
Epoch 3/24
85/85 [==============================] - 13s 151ms/step - loss: 0.5796 - accuracy: 0.8111
Epoch 4/24
85/85 [==============================] - 12s 143ms/step - loss: 0.3424 - accuracy: 0.8893
Epoch 5/24
85/85 [==============================] - 12s 145ms/step - loss: 0.2502 - accuracy: 0.9152
Epoch 6/24
85/85 [==============================] - 13s 148ms/step - loss: 0.1847 - accuracy: 0.9437
Epoch 7/24
85/85 [==============================] - 13s 147ms/step - loss: 0.1495 - accuracy: 0.9489
Epoch 8/24
85/85 [==============================] - 12s 144ms/step - loss: 0.1385 - accuracy: 0.9515
Epoch 9/24
85/85 [==============================] - 12s 144ms/step - loss: 0.0985 - accuracy: 0.9674
Epoch 10/24
85/85 [==============================] - 12s 144ms/step - loss: 0.0689 - accuracy: 0.9778
Epoch 11/24
85/85 [==============================] - 12s 144ms/step - loss: 0.0712 - accuracy: 0.9737
Epoch 12/24
85/85 [==============================] - 12s 143ms/step - loss: 0.0721 - accuracy: 0.9748
Epoch 13/24
85/85 [==============================] - 12s 143ms/step - loss: 0.0592 - accuracy: 0.9815
Epoch 14/24
85/85 [==============================] - 12s 143ms/step - loss: 0.0463 - accuracy: 0.9841
Epoch 15/24
85/85 [==============================] - 12s 144ms/step - loss: 0.0423 - accuracy: 0.9867
Epoch 16/24
85/85 [==============================] - 12s 143ms/step - loss: 0.0536 - accuracy: 0.9815
Epoch 17/24
85/85 [==============================] - 12s 146ms/step - loss: 0.0365 - accuracy: 0.9870
Epoch 18/24
85/85 [==============================] - 12s 145ms/step - loss: 0.0245 - accuracy: 0.9911
Epoch 19/24
85/85 [==============================] - 12s 144ms/step - loss: 0.0239 - accuracy: 0.9926
Epoch 20/24
85/85 [==============================] - 12s 146ms/step - loss: 0.0165 - accuracy: 0.9948
Epoch 21/24
85/85 [==============================] - 13s 150ms/step - loss: 0.0096 - accuracy: 0.9967
Epoch 22/24
85/85 [==============================] - 12s 146ms/step - loss: 0.0314 - accuracy: 0.9881
Epoch 23/24
85/85 [==============================] - 12s 144ms/step - loss: 0.0576 - accuracy: 0.9804
Epoch 24/24
85/85 [==============================] - 12s 145ms/step - loss: 0.0175 - accuracy: 0.9948
```

```
10/10 - 1s - loss: 0.1588 - accuracy: 0.9600
0.9599999785423279
```

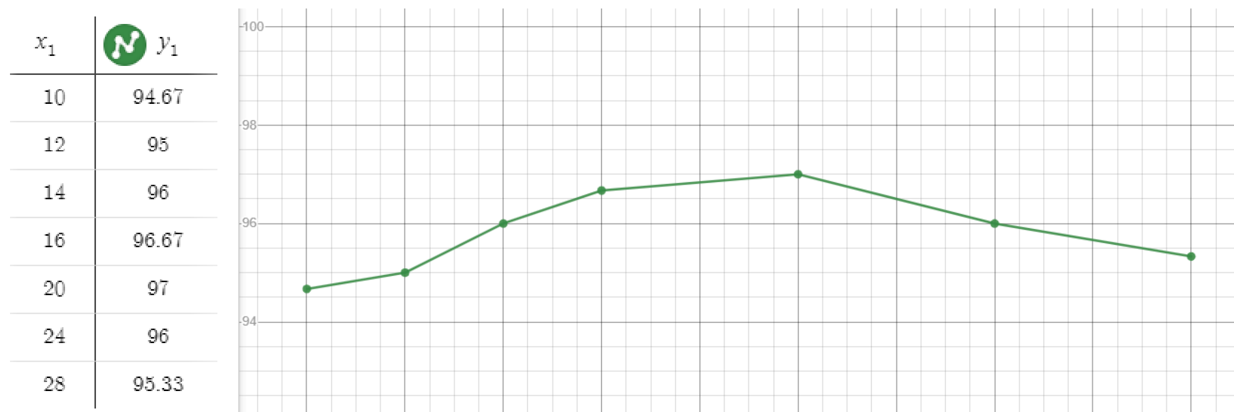
4.2 RNN Results

Following 10 epochs, the RNN yielded an accuracy of 29% over a training period of around 71 seconds. Following 16 epochs, the RNN yielded an accuracy of 87% over a training period of around 110 seconds. Following 24 epochs, the RNN yielded an accuracy of 90.67% over a training period of around 165 seconds.

```
Epoch 1/24
85/85 [==============================] - 9s 80ms/step - loss: 2.3265 - accuracy: 0.1300
Epoch 2/24
85/85 [==============================] - 7s 79ms/step - loss: 2.2486 - accuracy: 0.1748
Epoch 3/24
85/85 [==============================] - 7s 79ms/step - loss: 2.0432 - accuracy: 0.2537
Epoch 4/24
85/85 [==============================] - 7s 79ms/step - loss: 1.5871 - accuracy: 0.4181
Epoch 5/24
85/85 [==============================] - 7s 80ms/step - loss: 1.3630 - accuracy: 0.5085
Epoch 6/24
85/85 [==============================] - 7s 82ms/step - loss: 1.1781 - accuracy: 0.5807
Epoch 7/24
85/85 [==============================] - 7s 82ms/step - loss: 1.1111 - accuracy: 0.6019
Epoch 8/24
85/85 [==============================] - 7s 87ms/step - loss: 0.8484 - accuracy: 0.7041
Epoch 9/24
85/85 [==============================] - 7s 84ms/step - loss: 0.7543 - accuracy: 0.7415
Epoch 10/24
85/85 [==============================] - 7s 81ms/step - loss: 0.6766 - accuracy: 0.7633
Epoch 11/24
85/85 [==============================] - 7s 80ms/step - loss: 0.6452 - accuracy: 0.7811
Epoch 12/24
85/85 [==============================] - 7s 80ms/step - loss: 0.5533 - accuracy: 0.8200
Epoch 13/24
85/85 [==============================] - 7s 79ms/step - loss: 0.4820 - accuracy: 0.8348
Epoch 14/24
85/85 [==============================] - 7s 79ms/step - loss: 0.7497 - accuracy: 0.7489
Epoch 15/24
85/85 [==============================] - 7s 79ms/step - loss: 0.5311 - accuracy: 0.8274
Epoch 16/24
85/85 [==============================] - 7s 79ms/step - loss: 0.4321 - accuracy: 0.8530
Epoch 17/24
85/85 [==============================] - 7s 80ms/step - loss: 0.4428 - accuracy: 0.8493
Epoch 18/24
85/85 [==============================] - 7s 80ms/step - loss: 0.4451 - accuracy: 0.8600
Epoch 19/24
85/85 [==============================] - 7s 81ms/step - loss: 0.3537 - accuracy: 0.8830
Epoch 20/24
85/85 [==============================] - 7s 79ms/step - loss: 0.3475 - accuracy: 0.8841
Epoch 21/24
85/85 [==============================] - 7s 80ms/step - loss: 0.3149 - accuracy: 0.8944
Epoch 22/24
85/85 [==============================] - 7s 79ms/step - loss: 0.3262 - accuracy: 0.8930
Epoch 23/24
85/85 [==============================] - 7s 82ms/step - loss: 0.3207 - accuracy: 0.8944
Epoch 24/24
85/85 [==============================] - 7s 80ms/step - loss: 0.2828 - accuracy: 0.9033
```

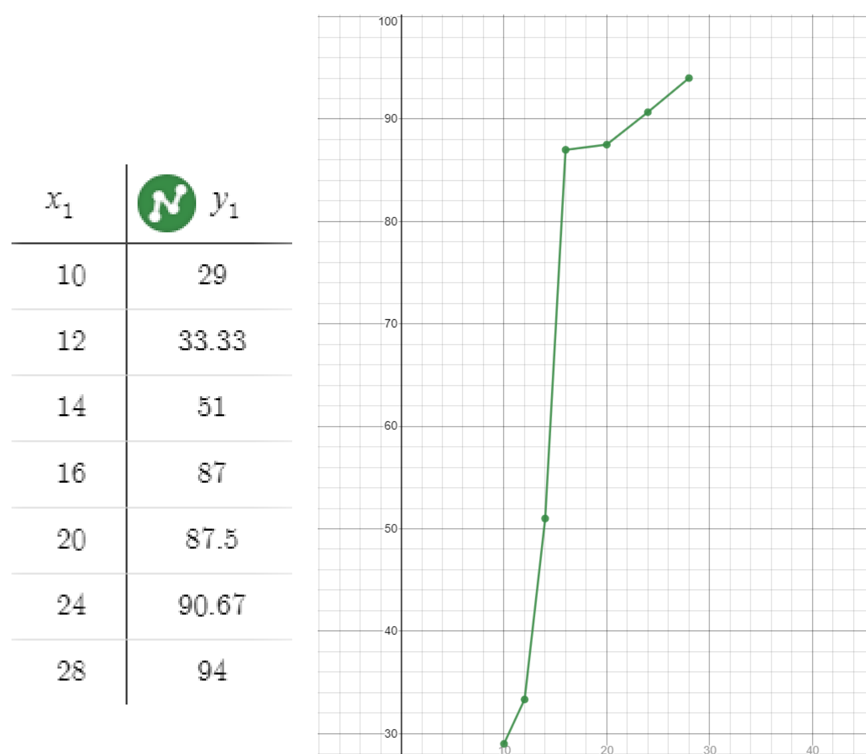
```
10/10 - 1s - loss: 0.2440 - accuracy: 0.9067
0.9066666960716248
```

4.3 Comparison and Analysis



x_1	y_1
10	94.67
12	95
14	96
16	96.67
20	97
24	96
28	95.33

My CNN's Performance Table and Graph (epoch vs accuracy) made using the Desmos Graphing Calculator, with x_1 representing epochs and y_1 the test dataset accuracy



x_1	y_1
10	29
12	33.33
14	51
16	87
20	87.5
24	90.67
28	94

My RNN's Performance Table and Graph (epoch vs accuracy) made using the Desmos Graphing Calculator, with x_1 representing epochs and y_1 the test dataset accuracy

The accuracy of the CNN was significantly greater than that of the RNN for all three trials. The accuracy for the CNN increased on trial two (16 epochs), but decreased on trial three (24 epochs). This was likely because of slight overfitting to the training dataset, since 24 epochs is relatively large for a dataset of 2700 samples. The accuracy of the RNN was surprisingly low for trial one (10 epochs), but increased greatly from trial one to trial two. This suggests that the RNN has a “warm up” period during which the loss function and the optimizer slowly adjust the weights in different directions, followed by a period of much more progressive set of epochs. The accuracy of the RNN continued to increase for trial 3. This shows that the dropout method worked successfully to prevent overfitting the small training dataset. I would hypothesize that more epochs would continue to increase the RNN’s accuracy.

The training duration of the RNN was significantly shorter than that of the CNN for all three trials. This was likely because the weights of the RNN are simple, classical NN weights, whereas the weights of the CNN are pixel values between 0 and 255 for each filter of size 3x3.

Finally, an important factor was the amount of troubleshooting each NN took to work successfully. I was able to code CNN and achieve the above accuracy in under 8 hours. In comparison, the RNN took over 20 man hours to code.

4.4 Conclusion

The final drop in accuracy of the CNN due to overfitting leads me to conclude that, keeping the size of the dataset constant, there is a limit to the CNN’s accuracy (in

this case, roughly 97%), achieved at the right number of epochs. Less epochs will not be enough to reach this maximum, and more will lead to overfitting. This is not the case for the RNN. Because the accuracy of the RNN continued to rise with an increase in epochs, I believe that the RNN will be able to come closer to 100% accuracy, but over a very large number of epochs. However, because the CNN was more accurate for each trial, I will conclude that it is more accurate in speech recognition applications. Nevertheless, the difference in training duration proves that the CNN was slower than the RNN when training on the same dataset. Thus, the RNN is a faster approach to speech recognition, because it is easier to adjust variables such as the layer size, the optimizer algorithm, and the loss function, and see the corresponding effects on training and testing accuracy for the RNN.

Works Cited

An, Sungtae. “Feedforward Neural Networks.” *Sungtae's Awesome Homepage*, Georgia Institute of Technology, 8 Oct. 2017, www.cc.gatech.edu/~san37/post/dlhc-fnn/.

Anvarjon, Tursunov, et al. “Deep-Net: A LIGHTWEIGHT CNN-Based Speech Emotion Recognition System Using Deep Frequency Features.” *MDPI*, Multidisciplinary Digital Publishing Institute, 12 Sept. 2020, www.mdpi.com/1424-8220/20/18/5212

“Introduction to Loss Functions.” *Algorithmia Blog*, 28 Apr. 2021, algorithmia.com/blog/introduction-to-loss-functions#types-of-loss-functions.

Jordan, Jeremy. “Neural Networks: Representation.” *Jeremy Jordan*, 26 Jan. 2018, www.jeremyjordan.me/intro-to-neural-networks/.

Kinsley, Harrison. “Recurrent Neural Networks - Deep Learning Basics with Python, TensorFlow and Keras P.7.” *Python Programming Tutorials*, 7 Sept. 2018, pythonprogramming.net/recurrent-neural-network-deep-learning-python-tensorflow-keras/.

Kostadinov, Simeon. “Recurrent Neural Networks with Python Quick Start Guide.” *Google Books*, Packt>, 30 Oct. 2018, www.google.com/books/edition/Recurrent_Neural_Networks_with_Python_Qu/cC59DwAAQBAJ.

Nathan, Mathura. “Plot Audio File as Time Series Using Scipy Python.” *GaussianWaves*, 2 Aug. 2020, www.gaussianwaves.com/2020/01/how-to-plot-audio-files-as-time-series-using-scipy-python/.

Nielsen, Michael. “Neural Networks and Deep Learning.” *Static Latex Studio*, Dec. 2019, static.latexstudio.net/article/2018/0912/neuralnetworksanddeeplearning.pdf.

Raschka, Sebastian. “Gradient Descent and Stochastic Gradient Descent.” *Gradient Descent and Stochastic Gradient Descent - Mlxtend*, Mlxtend, 2014, rasbt.github.io/mlxtend/user_guide/general_concepts/gradient-optimization/.

Ruscica, Tim. “Deep Computer Vision.” *Google Colaboratory*, 3 Mar. 202AD, colab.research.google.com/drive/1ZZXnCjFEOkp_KdNcNabd14yok0BAIuwS.

Saha, Sumit. “A Comprehensive Guide to Convolutional Neural Networks - the eli5 Way.” *Towards Data Science*, 17 Dec. 2018, towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

Sewak, Mohit, et al. “Practical Convolutional Neural Networks.” *Google Books*, Packt>, 27 Feb. 2018, www.google.com/books/edition/Practical_Convolutional_Neural_Networks/bOlODwAAQBAJ.

“Understanding Spectrograms.” *IZotope*, 11 Apr. 2019, www.izotope.com/en/learn/understanding-spectrograms.html.

Venkatachalam, Mahendran. “Recurrent Neural Networks.” *Towards Data Science*, 22 June 2019, towardsdatascience.com/recurrent-neural-networks-d4642c9bc7ce.

Appendices

Python Code for the CNN

```
1 #!/usr/bin/env python
2 # coding: utf-8
3
4 # In[1]:
5
6
7 import numpy as np
8 import os
9 import sys
10 from random import shuffle
11 from tqdm import tqdm
12 import cv2
13 import matplotlib.pyplot as plt
14 import PIL
15
16 TRAIN_DIR = 'C:/Users/yigit/Desktop/free-spoken-digit-dataset-master/training-
spectrograms'
17 TEST_DIR = 'C:/Users/yigit/Desktop/free-spoken-digit-dataset-master/testing-
spectrograms'
18
19 img_count = 2700
20 trainsize = 64 * 64 * img_count
21 img_test = 300
22 testsize = 64 * 64 * img_test
23
24 x_train = np.arange(trainsize)
25 x_labels = np.arange(img_count)
26 y_test = np.arange(testsize)
27 y_labels = np.arange(img_test)
28
29
30 # In[2]:
31
32
33 def label_img(img):
34     word_label = img.split('_')[-0]
35     label = int(word_label)
36     return label
37
38
39 # In[3]:
40
41
42 def create_train_data():
43     training_data = []
44     training_labels = []
45     count = 0
46     for img in tqdm(os.listdir(TRAIN_DIR)):
47         count = count + 1
48         label = label_img(img)
49         path = os.path.join(TRAIN_DIR, img)
50         image = PIL.Image.open('C:/Users/yigit/Desktop/free-spoken-digit-dataset-
master/training-spectrograms/' + img)
51         sequence = image.getdata()
52         image_array = np.array(sequence)
53         training_data.append(image_array)
54         training_labels.append(label_img(img))
55
56
57     return training_data, training_labels
```

```
58
59
60 # In[4]:
61
62
63 training, labels = create_train_data()
64
65
66 # In[5]:
67
68
69 def create_test_data():
70     testing_data = []
71     testing_labels = []
72     count = 0
73     for img in tqdm(os.listdir(TEST_DIR)):
74         count = count + 1
75         label = label_img(img)
76         path = os.path.join(TRAIN_DIR, img)
77         image = PIL.Image.open('C:/Users/yigit/Desktop/free-spoken-digit-dataset-
master/testing-spectrograms/' + img)
78         sequence = image.getdata()
79         image_array = np.array(sequence)
80         testing_data.append(image_array)
81         testing_labels.append(label_img(img))
82
83
84     return testing_data, testing_labels
85
86
87 # In[6]:
88
89
90 testing, tags = create_test_data()
91
92
93 # In[7]:
94
95
96 traincount = 0
97 for i in range(len(training)):
98     for j in range(len(training[i])):
99         value = training[i][j][0]
100        x_train[traincount] = value
101        traincount = traincount + 1
102
103
104 # In[8]:
105
106
107 x_train = x_train.reshape(img_count, 64, 64, 1)
108
109
110 # In[9]:
111
112
113 for i in range(img_count):
114     x_labels[i] = labels[i]
115
116
```

```
117 # In[10]:
118
119
120 x_labels = x_labels.reshape(img_count, 1)
121
122
123 # In[11]:
124
125
126 testcount = 0
127 for i in range(len(testing)):
128     for j in range(len(testing[i])):
129         value = testing[i][j][0]
130         y_test[testcount] = value
131         testcount = testcount + 1
132
133
134 # In[12]:
135
136
137 y_test = y_test.reshape(img_test, 64, 64, 1)
138
139
140 # In[13]:
141
142
143 for i in range(img_test):
144     y_labels[i] = tags[i]
145
146
147 # In[14]:
148
149
150 y_labels = y_labels.reshape(img_test, 1)
151
152
153 # In[15]:
154
155
156 x_train, y_test = x_train / 255.0, y_test / 255.0
157
158
159 # In[16]:
160
161
162 class_names = ['0', '1', '2', '3', '4',
163                '5', '6', '7', '8', '9']
164
165
166 # In[17]:
167
168
169 import tensorflow as tf
170
171 from tensorflow.keras import datasets, layers, models
172
173
174 # In[18]:
175
176
```

```
177 model = models.Sequential()
178 model.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(64, 64, 1)))
179 model.add(layers.MaxPooling2D((2, 2)))
180 model.add(layers.Conv2D(64, (3, 3), activation='relu'))
181 model.add(layers.MaxPooling2D((2, 2)))
182 model.add(layers.Conv2D(64, (3, 3), activation='relu'))
183 model.add(layers.MaxPooling2D((2, 2)))
184 model.add(layers.Conv2D(64, (3, 3), activation='relu'))
185
186
187 # In[20]:
188
189
190 model.add(layers.Flatten())
191 model.add(layers.Dense(64, activation='relu'))
192 model.add(layers.Dense(10))
193
194
195 # In[22]:
196
197
198 model.compile(optimizer='adam',
199               loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
200               metrics=['accuracy'])
201
202
203 # In[24]:
204
205
206 model.fit(x_train, x_labels, epochs = 24)
207
208
209 # In[25]:
210
211
212 test_loss, test_acc = model.evaluate(y_test, y_labels, verbose=2)
213 print(test_acc)
```

Terminal Output for the CNN

```
100%|████████████████████████████████████████████████████████████████████████████| 2700/2700 [00:19<00:00, 141.09it/s]
100%|████████████████████████████████████████████████████████████████████████████| 300/300 [00:02<00:00, 129.04it/s]
```

2021-08-10 11:13:01.049707: W
tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cudart64_110.dll'; dlerror: cudart64_110.dll not found
2021-08-10 11:13:01.049882: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.
2021-08-10 11:13:05.075755: W
tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'nvcuda.dll'; dlerror: nvcuda.dll not found
2021-08-10 11:13:05.075906: W tensorflow/stream_executor/cuda/cuda_driver.cc:326] failed call to cuInit: UNKNOWN ERROR (303)
2021-08-10 11:13:05.081306: I
tensorflow/stream_executor/cuda/cuda_diagnostics.cc:169] retrieving CUDA diagnostic information for host: DESKTOP-GJ6KHSP
2021-08-10 11:13:05.081560: I
tensorflow/stream_executor/cuda/cuda_diagnostics.cc:176] hostname: DESKTOP-GJ6 KHSP
2021-08-10 11:13:05.081974: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations: AVX
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.

2021-08-10 11:13:07.956747: I

tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:176] None of the MLIR

Optimization Passes are enabled (registered 2)

Epoch 1/24

85/85 [==============================] - 14s 155ms/step - loss: 1.9459 -

accuracy: 0.2815

Epoch 2/24

85/85 [==============================] - 13s 148ms/step - loss: 0.9275 -

accuracy: 0.6937

Epoch 3/24

85/85 [==============================] - 12s 144ms/step - loss: 0.5315 -

accuracy: 0.8248

Epoch 4/24

85/85 [==============================] - 13s 154ms/step - loss: 0.3486 -

accuracy: 0.8822

Epoch 5/24

85/85 [==============================] - 14s 159ms/step - loss: 0.2183 -

accuracy: 0.9289

Epoch 6/24

85/85 [==============================] - 14s 159ms/step - loss: 0.1653 -

accuracy: 0.9452

Epoch 7/24

85/85 [==============================] - 13s 155ms/step - loss: 0.1254 -

accuracy: 0.9578

Epoch 8/24

85/85 [==============================] - 13s 156ms/step - loss: 0.1079 -

accuracy: 0.9648

Epoch 9/24

85/85 [==============================] - 12s 146ms/step - loss: 0.0919 -

accuracy: 0.9670

Epoch 10/24

85/85 [==============================] - 15s 174ms/step - loss: 0.0848 -

accuracy: 0.9704

Epoch 11/24

85/85 [==============================] - 16s 185ms/step - loss: 0.0658 -

accuracy: 0.9789

Epoch 12/24

85/85 [==============================] - 15s 172ms/step - loss: 0.0560 -

accuracy: 0.9819

Epoch 13/24

85/85 [==============================] - 15s 175ms/step - loss: 0.0595 -

accuracy: 0.9822

Epoch 14/24

85/85 [==============================] - 15s 181ms/step - loss: 0.0404 -

accuracy: 0.9874

Epoch 15/24

85/85 [==============================] - 15s 176ms/step - loss: 0.0297 -

accuracy: 0.9904

Epoch 16/24

85/85 [==============================] - 14s 164ms/step - loss: 0.0410 -

accuracy: 0.9844

Epoch 17/24

85/85 [==============================] - 13s 158ms/step - loss: 0.0642 -

accuracy: 0.9807

Epoch 18/24

85/85 [==============================] - 14s 164ms/step - loss: 0.0344 -

accuracy: 0.9881

Epoch 19/24

85/85 [==============================] - 15s 174ms/step - loss: 0.0326 -

accuracy: 0.9889

Epoch 20/24

85/85 [==============================] - 15s 179ms/step - loss: 0.0344 -

accuracy: 0.9881

Epoch 21/24

85/85 [==============================] - 15s 171ms/step - loss: 0.0370 -

accuracy: 0.9889

Epoch 22/24

85/85 [==============================] - 14s 164ms/step - loss: 0.0104 -

accuracy: 0.9956

Epoch 23/24

85/85 [==============================] - 13s 148ms/step - loss: 0.0076 -

accuracy: 0.9981

Epoch 24/24

85/85 [==============================] - 14s 161ms/step - loss: 0.0267 -

accuracy: 0.9922

10/10 - 1s - loss: 0.1988 - accuracy: 0.9667

0.9666666388511658

Python Code for the RNN

```
1 #!/usr/bin/env python
2 # coding: utf-8
3
4 # In[1]:
5
6
7 import numpy as np
8 import os
9 import sys
10 from random import shuffle
11 from tqdm import tqdm
12 import cv2
13 import matplotlib.pyplot as plt
14 import PIL
15
16 TRAIN_DIR = 'C:/Users/yigit/Desktop/free-spoken-digit-dataset-master/training-
spectrograms'
17 TEST_DIR = 'C:/Users/yigit/Desktop/free-spoken-digit-dataset-master/testing-
spectrograms'
18
19 img_count = 2700
20 trainsize = 64 * 64 * img_count
21 img_test = 300
22 testsize = 64 * 64 * img_test
23
24 x_train = np.arange(trainsize)
25 x_labels = np.arange(img_count)
26 y_test = np.arange(testsize)
27 y_labels = np.arange(img_test)
28
29
30 # In[2]:
31
32
33 def label_img(img):
34     word_label = img.split('_')[-0]
35     label = int(word_label)
36     return label
37
38
39 # In[3]:
40
41
42 def create_train_data():
43     training_data = []
44     training_labels = []
45     count = 0
46     for img in tqdm(os.listdir(TRAIN_DIR)):
47         count = count + 1
48         label = label_img(img)
49         path = os.path.join(TRAIN_DIR, img)
50         image = PIL.Image.open('C:/Users/yigit/Desktop/free-spoken-digit-dataset-
master/training-spectrograms/' + img)
51         sequence = image.getdata()
52         image_array = np.array(sequence)
53         training_data.append(image_array)
54         training_labels.append(label_img(img))
55
56
57     return training_data, training_labels
```

```
58
59
60 # In[4]:
61
62
63 training, labels = create_train_data()
64
65
66 # In[5]:
67
68
69 def create_test_data():
70     testing_data = []
71     testing_labels = []
72     count = 0
73     for img in tqdm(os.listdir(TEST_DIR)):
74         count = count + 1
75         label = label_img(img)
76         path = os.path.join(TRAIN_DIR, img)
77         image = PIL.Image.open('C:/Users/yigit/Desktop/free-spoken-digit-dataset-
master/testing-spectrograms/' + img)
78         sequence = image.getdata()
79         image_array = np.array(sequence)
80         testing_data.append(image_array)
81         testing_labels.append(label_img(img))
82
83
84     return testing_data, testing_labels
85
86
87 # In[6]:
88
89
90 testing, tags = create_test_data()
91
92
93 # In[7]:
94
95
96 traincount = 0
97 for i in range(len(training)):
98     for j in range(len(training[i])):
99         value = training[i][j][0]
100        x_train[traincount] = value
101        traincount = traincount + 1
102
103
104 # In[9]:
105
106
107 x_train = x_train.reshape(img_count, 64, 64)
108
109
110 # In[10]:
111
112
113 for i in range(img_count):
114     x_labels[i] = labels[i]
115
116
```

```
117 # In[11]:
118
119
120 x_labels = x_labels.reshape(img_count, 1)
121
122
123 # In[12]:
124
125
126 print(x_train.shape)
127 print(x_labels.shape)
128
129
130 # In[13]:
131
132
133 testcount = 0
134 for i in range(len(testing)):
135     for j in range(len(testing[i])):
136         value = testing[i][j][0]
137         y_test[testcount] = value
138         testcount = testcount + 1
139
140
141 # In[15]:
142
143
144 y_test = y_test.reshape(img_test, 64, 64)
145
146
147 # In[16]:
148
149
150 for i in range(img_test):
151     y_labels[i] = tags[i]
152
153
154 # In[17]:
155
156
157 y_labels = y_labels.reshape(img_test, 1)
158
159
160 # In[18]:
161
162
163 x_train, y_test = x_train / 255.0, y_test / 255.0
164
165
166 # In[19]:
167
168
169 class_names = ['0', '1', '2', '3', '4',
170                '5', '6', '7', '8', '9']
171
172
173 # In[20]:
174
175
176 import tensorflow as tf
```

```
177 from tensorflow import keras
178 from tensorflow.keras import models, layers
179 from tensorflow.keras.models import Sequential
180 from tensorflow.keras.layers import Dense, Dropout, LSTM
181
182
183 # In[25]:
184
185
186 model = keras.Sequential()
187
188 model.add(layers.LSTM(128, input_shape=(x_train.shape[1:]), activation='relu',
    return_sequences=True))
189 model.add(Dropout(0.2))
190
191 model.add(layers.LSTM(128, activation='relu'))
192 model.add(Dropout(0.2))
193
194 model.add(layers.Dense(32, activation='relu'))
195 model.add(Dropout(0.2))
196 model.add(layers.Dense(10, activation='softmax'))
197
198
199 # In[26]:
200
201
202 opt = tf.keras.optimizers.Adam(learning_rate=0.001, decay=1e-5)
203
204 model.compile(
205     loss='sparse_categorical_crossentropy',
206     optimizer=opt,
207     metrics=['accuracy'],
208 )
209
210
211 # In[28]:
212
213
214 model.fit(x_train, x_labels, epochs=24)
215
216
217 # In[29]:
218
219
220 test_loss, test_acc = model.evaluate(y_test, y_labels, verbose=2)
221 print(test_acc)
```

Terminal Output for the RNN

```
100%|██████████| 2700/2700 [00:18<00:00, 146.43it/s]

100%|██████████| 300/300 [00:01<00:00, 153.55it/s]

(2700, 64, 64)
```

(2700, 1)

2021-08-10 11:28:55.410371: W

tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cudart64_110.dll'; dlerror: cudart64_110.dll not found

2021-08-10 11:28:55.410536: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.

2021-08-10 11:28:58.510744: W

tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'nvcuda.dll'; dlerror: nvcuda.dll not found

2021-08-10 11:28:58.511072: W tensorflow/stream_executor/cuda/cuda_driver.cc:326] failed call to cuInit: UNKNOWN ERROR (303)

2021-08-10 11:28:58.518412: I

tensorflow/stream_executor/cuda/cuda_diagnostics.cc:169] retrieving CUDA diagnostic information for host: DESKTOP-GJ6KHSP

2021-08-10 11:28:58.518682: I

tensorflow/stream_executor/cuda/cuda_diagnostics.cc:176] hostname:

DESKTOP-GJ6KHSP

2021-08-10 11:28:58.519533: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations: AVX

To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.

2021-08-10 11:28:59.272410: I

tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:176] None of the MLIR

Optimization Passes are enabled (registered 2)

Epoch 1/24

85/85 [==============================] - 10s 92ms/step - loss: 2.2899 -

accuracy: 0.1315

Epoch 2/24

85/85 [==============================] - 8s 89ms/step - loss: 2.7080 -

accuracy: 0.1807

Epoch 3/24

85/85 [==============================] - 7s 81ms/step - loss: 2.4055 -

accuracy: 0.1837

Epoch 4/24

85/85 [==============================] - 8s 91ms/step - loss: 2.2728 -

accuracy: 0.1415

Epoch 5/24

85/85 [==============================] - 7s 85ms/step - loss: 2.0897 -

accuracy: 0.2237

Epoch 6/24

85/85 [==============================] - 7s 83ms/step - loss: 1.8103 -

accuracy: 0.3148

Epoch 7/24

85/85 [==============================] - 8s 94ms/step - loss: 3969.2903 -

accuracy: 0.3733

Epoch 8/24

85/85 [==============================] - 8s 89ms/step - loss: 2.5946 -

accuracy: 0.1311

Epoch 9/24

85/85 [==============================] - 8s 91ms/step - loss: 2.2648 -

accuracy: 0.1663

Epoch 10/24

85/85 [==============================] - 8s 99ms/step - loss: 2.0859 -

accuracy: 0.2093

Epoch 11/24

85/85 [==============================] - 8s 92ms/step - loss: 1.9029 -

accuracy: 0.2493

Epoch 12/24

85/85 [==============================] - 7s 85ms/step - loss: 1.7761 -

accuracy: 0.3026

Epoch 13/24

85/85 [==============================] - 7s 78ms/step - loss: 1.6150 -

accuracy: 0.3804

Epoch 14/24

85/85 [==============================] - 10s 115ms/step - loss: 1.7884 -

accuracy: 0.3015

Epoch 15/24

85/85 [==============================] - 9s 102ms/step - loss: 1.5950 -

accuracy: 0.4004

Epoch 16/24

85/85 [==============================] - 7s 83ms/step - loss: 1.3970 -

accuracy: 0.4630

Epoch 17/24

85/85 [==============================] - 8s 91ms/step - loss: 1.3381 -

accuracy: 0.4993

Epoch 18/24

85/85 [==============================] - 7s 84ms/step - loss: 1.2130 -

accuracy: 0.5585

Epoch 19/24

85/85 [==============================] - 7s 86ms/step - loss: 1.0974 -

accuracy: 0.5948

Epoch 20/24

85/85 [==============================] - 7s 84ms/step - loss: 1.0235 -

accuracy: 0.6244

Epoch 21/24

85/85 [==============================] - 7s 85ms/step - loss: 0.9381 -

accuracy: 0.6659

Epoch 22/24

85/85 [==============================] - 7s 85ms/step - loss: 0.8594 - accuracy: 0.6941

Epoch 23/24

85/85 [==============================] - 7s 82ms/step - loss: 0.8534 - accuracy: 0.6900

Epoch 24/24

85/85 [==============================] - 7s 86ms/step - loss: 0.8193 - accuracy: 0.7074

10/10 - 1s - loss: 0.5634 - accuracy: 0.8400

0.8399999737739563