Embedded System
ESC7 - DL model optimization for Lightweight Keyword Spotting
Project - Report

Name - *Karan Singh Kushwah (B20CS024)*
Khobragade Atul Yashwant (B20CS027)

***Abstract* -**
The following report outlines the use of Deep Learning (DL) model optimization for Lightweight Keyword Spotting. For many applications, including speech recognition, virtual assistants, and security systems, keyword spotting is a crucial task. However, traditional keyword spotting models require significant computation and resources, making them unsuitable for lightweight applications such as smartphones, wearable devices, and Internet of Things (IoT) devices. The results demonstrate that the improved DL model is ideal for lightweight applications since it can achieve excellent accuracy with little computational and memory cost.

Dataset Used: Google Speech Commands dataset

Introduction:
The Google Speech Commands dataset is a publicly available dataset that contains over 105,000 recordings of 35 different English words, each spoken by thousands of different people. The dataset was created for voice assistants, mobile phones, and other applications that require voice recognition to train and evaluate keyword recognition models.

The dataset comprises recordings of each word for one second that were made with a range of tools and in various acoustic settings. A training set and a validation set are created from the recordings. The training set contains 85,000 recordings, while the validation set contains 20,000 recordings.

Reasons for Choosing the Dataset:

The Google Speech Commands dataset was chosen for a specific task because it is a widely used dataset for training keyword spotting models, making it a reliable benchmark for comparing different models. The dataset is also openly accessible, which makes it simple for researchers and developers to access.

Subset of the Dataset:

For the task at hand, a subset of the Google Speech Commands dataset was used, consisting of only eight commands: "yes", "no", "up", "down", "left", "right", "stop", and "go". These commands were chosen because they are commonly used in voice-controlled devices and applications, and because they represent a diverse range of phonetic characteristics.

Pre-processing of data:

We have pre-processed the audio data in the provided dataset for keyword detection. A few parameters are first defined, including the location of the dataset, the number of samples to be considered, and the construction of a dictionary that will be used to record the mapping between each audio file's labels, MFCCs, and filenames. The audio files in each subdirectory of the dataset are then processed by the code in a loop. In order to ensure that the length of the signal remains constant, it loads and slices each audio file separately.

Any audio files with less samples than the predetermined quantity are then deleted by the programme. The code then extracts the signal's mel-frequency cepstral coefficients (MFCCs). MFCCs are a common feature of audio processing that keep track of the signal's spectrum characteristics. The MFCCs are extracted using the librosa library and then saved in the data dictionary. The label and file name of the audio file are also listed in the dictionary.

The processed data is then placed in a JSON file for subsequent use. Overall, the code is in charge of preparing the audio data for use in training and testing Deep Learning models for Keyword Spotting using MFCC feature extraction and storing in an easy-to-understand format.

Below is the main code for this -

```
data = {
    "mapping": [],
    "labels": [],
    "MFCCs": [],
    "files": []
}

for i, (dirpath, dirnames, filenames) in enumerate(os.walk(DATASET_PATH)):

    if dirpath is not DATASET_PATH:

        label = dirpath.split("/")[-1]
        data["mapping"].append(label)
        print("\nProcessing: '{}'".format(label))

        for f in filenames:
            file_path = os.path.join(dirpath, f)

            signal, sample_rate = librosa.load(file_path)

            if len(signal) >= SAMPLES_TO_CONSIDER:

                signal = signal[:SAMPLES_TO_CONSIDER]

                num_mfcc = 13
                n_fft = 2048
                hop_length = 512
                MFCCs = librosa.feature.mfcc(y=signal, n_mfcc=num_mfcc, hop_length=hop_length, n_fft=n_fft)

                data["MFCCs"].append(MFCCs.T.tolist())
                data["labels"].append(i-1)
                data["files"].append(file_path)
                print("{}: {}".format(file_path, i-1))
```

Fig1 - Pre-processing Code Snippet

Model Training and testing -

We decided to train the Deep Neural Network (DNN) and the Convolutional Neural Network (CNN), two Deep Learning models.Because they can learn to represent audio signals in a way that is appropriate for classification, DNN and CNN are popular options for audio classification tasks.

DNN models are good at learning high-level features from the input data. In the case of audio classification, a DNN can learn to extract features such as frequency content and amplitude modulation, which are relevant to the task of identifying speech commands. However, the performance of DNN models may suffer if the input data has a complex structure, such as the spectrogram representation of audio signals used in this dataset.

CNNs, on the other hand, are particularly well-suited for tasks involving image or sequence data, such as the spectrogram representation of audio signals. They can automatically learn features of varying complexity by convolving a set of filters over the input data. In the context of audio classification, CNNs can learn to extract spectral features that are important for identifying speech commands, such as the formants and pitch of the spoken words.

DNN model -

The Below model is a Deep Neural Network (DNN) with 4 convolutional layers and 4 dense layers. The first two convolutional layers have 32 and 64 filters respectively with a kernel size of 3x3 and ReLU activation function. The max-pooling layer with pool size 2x2 follows each convolutional layer.

```
def build_dnn(input_shape, loss="sparse_categorical_crossentropy", learning_rate=0.001):
    model = tf.keras.models.Sequential()
    model.add(tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation='relu'))
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(256, activation='relu'))
    model.add(tf.keras.layers.Dropout(0.5))
    model.add(tf.keras.layers.Dense(128, activation='relu'))
    model.add(tf.keras.layers.Dropout(0.3))
    model.add(tf.keras.layers.Dense(64, activation='relu'))
    model.add(tf.keras.layers.Dropout(0.3))
    model.add(tf.keras.layers.Dense(8, activation='softmax'))

    optimizer = tf.optimizers.Adam(learning_rate=learning_rate)

    # compile model
    model.compile(optimizer=optimizer,
                  loss=loss,
                  metrics=["accuracy"])

    # print model parameters on console
    model.summary()

    return model

input_shape = (X_train.shape[1], X_train.shape[2], 1)
model = build_dnn(input_shape, learning_rate=0.001)
```

Fig2 - Deep Neural Network Code Snippet

The output from the convolutional layers is flattened and then passed through 3 dense layers with 256, 128, and 64 units respectively, each with ReLU activation function. A dropout regularization of 0.5, 0.3, and 0.3 is applied to the 1st, 2nd, and 3rd dense layer respectively, to reduce overfitting.
Finally, the output layer with 8 units and a softmax activation function is added. The model is compiled with the Adam optimizer, sparse categorical cross-entropy loss function, and accuracy as the evaluation metric.

CNN model -

The below code creates a CNN model for audio classification. It consists of three convolutional layers followed by max-pooling layers, a dense layer, and a softmax output layer. The model has 64 filters in the first convolutional layer, 32 filters in the second convolutional layer, and 32 filters in the third convolutional layer.

The output of the third convolutional layer is flattened and fed into the dense layer. The model uses batch normalization and dropout regularization to prevent overfitting. The optimizer used for training the model is Adam, and the loss function used is sparse categorical cross-entropy.

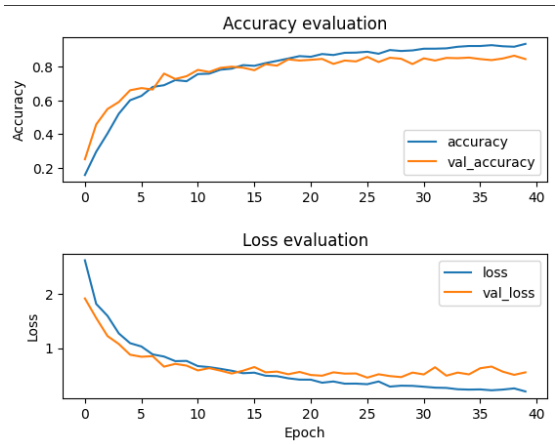
```python
def build_model(input_shape, loss="sparse_categorical_crossentropy", learning_rate=0.0001):

    # build network architecture using convolutional layers
    model = tf.keras.models.Sequential()

    # 1st conv layer
    model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu', input_shape=input_shape, kernel_regularizer=tf.keras.regularizers.l2(0.001)))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D((3, 3), strides=(2,2), padding='same'))

    # 2nd conv layer
    model.add(tf.keras.layers.Conv2D(32, (3, 3), activation='relu',kernel_regularizer=tf.keras.regularizers.l2(0.001)))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D((3, 3), strides=(2,2), padding='same'))

    # 3rd conv layer
    model.add(tf.keras.layers.Conv2D(32, (2, 2), activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.001)))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D((2, 2), strides=(2,2), padding='same'))

    # flatten output and feed into dense layer
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(64, activation='relu'))
    tf.keras.layers.Dropout(0.3)

    # softmax output layer
    model.add(tf.keras.layers.Dense(8, activation='softmax'))

    optimiser = tf.optimizers.Adam(learning_rate=learning_rate)

    # compile model
    model.compile(optimizer=optimiser,
                  loss=loss,
                  metrics=["accuracy"])

    # print model parameters on console
    model.summary()

    return model
```

Fig3 - Convolutional Neural Network Code Snippet

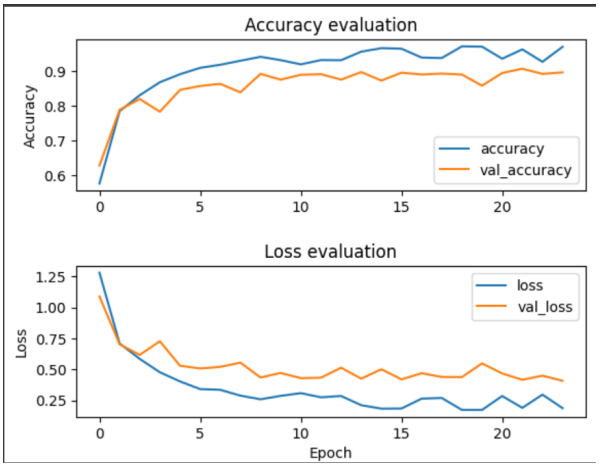
Comparison between DNN and CNN model -

Training score -

Fig4 - Comparison between DNN and CNN Model



DNN model



CNN model

Testing Score -

For DNN model -

```
45/45 [==============================] - 0s 8ms/step - loss: 0.5282 - accuracy: 0.8614

Test loss: 0.5282009840011597, test accuracy: 86.14206314086914
```

For CNN model -

```
45/45 [==============================] - 0s 9ms/step - loss: 0.4552 - accuracy: 0.8955

Test loss: 0.4551562964916229, test accuracy: 89.55431580543518
```

From the above comparison , it is clear that CNN model performs better than DNN model . This is likely due to the fact that CNNs are better suited for processing 2D data, such as images and spectrograms. The provided pre-processing code converts the audio samples into spectrograms, which are 2D representations of the audio signals.

Model Optimization and Justification -

1. Perform model robustness check i.e., if the weights/layers etc. in your neural network design change, what is the implication on the accuracy? You can carry out multiple experiments on your design and then tabulate your obtained results.

I varied the layers from 2 to 4 layers and got the best accuracy with the 3 layer CNN model.

Model 1:
This model has three convolutional layers with 64, 32, and 32 filters, respectively, followed by a single dense layer with 64 neurons. The kernel sizes for the convolutional layers are 3x3, 3x3, and 2x2, respectively, and the pooling sizes are 3x3, 3x3, and 2x2, respectively.Other Parameters are further explained in the report.

Training Score -

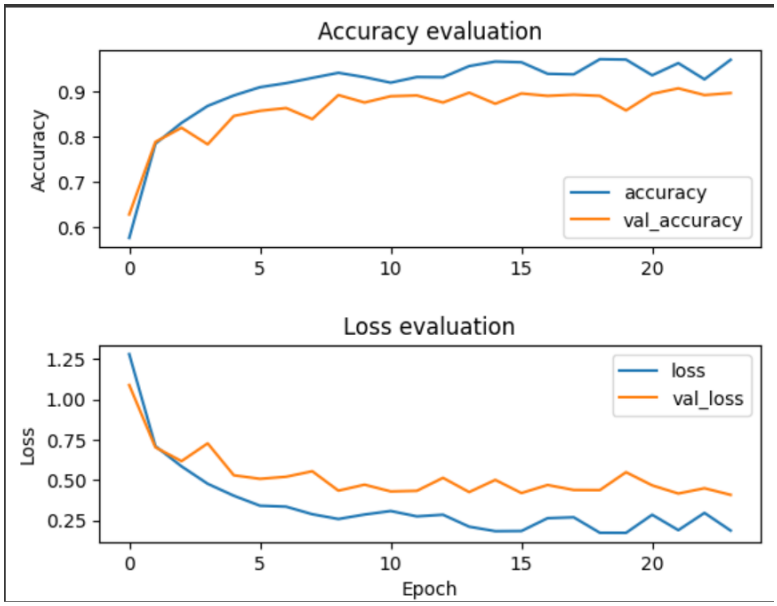


Fig5 - Accuracy and Loss variation for 3 Layer CNN model

Testing Score -

```
45/45 [==============================] - 0s 9ms/step - loss: 0.4552 - accuracy: 0.8955

Test loss: 0.4551562964916229, test accuracy: 89.55431580543518
```

Model2:
This model is similar to Model 1, but has only two convolutional layers with 64 and 32 filters, respectively, followed by a single dense layer with 64 neurons. The kernel sizes for the convolutional layers are 3x3, and 3x3, respectively, and the pooling sizes are 3x3 and 2x2, respectively.

Training Score -

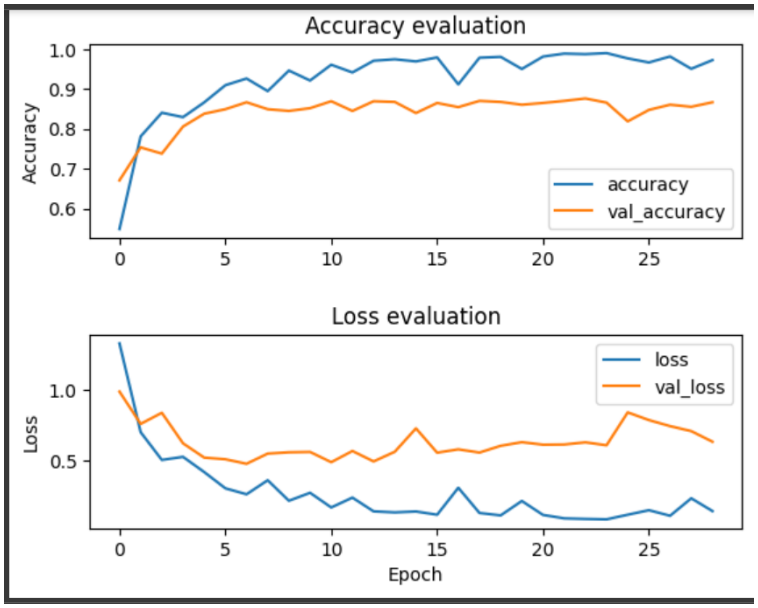


Fig6 - Accuracy and Loss variation for 2 Layer CNN model

Testing Score -

```
45/45 [==============================] - 0s 11ms/step - loss: 0.6226 - accuracy: 0.8774

Test loss: 0.6226077079772949, test accuracy: 87.74373531341553
```

Model3:
This model is also similar to Model 1, but has an additional convolutional layer with 64 filters and a kernel size of 4x4, followed by a single dense layer with 64 neurons. The pooling sizes for the first and third convolutional layers are 4x4 and 2x2, respectively, while the second convolutional layer has a pooling size of 3x3.

Training Score -

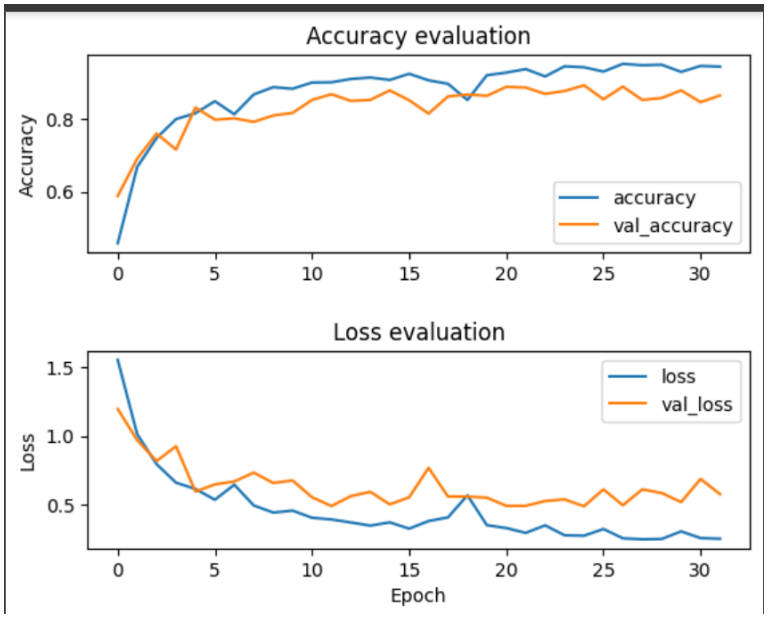


Fig7 - Accuracy and Loss variation for 4 Layer CNN model

Testing Score -

```
45/45 [==============================] - 0s 10ms/step - loss: 0.5308 - accuracy: 0.8705

Test loss: 0.5307533740997314, test accuracy: 87.0473563671112
```

Fig8 - Tabulate the results for the 3 models

Model	**No. Of Convolutional Layer**	**Filters**	**Kernel Sizes**	**Test Loss**	**Test Accuracy**
Model 1	3	64, 32, and 32 filters	3x3, 3x3, and 2x2	0.455	**89.554**
Model 2	2	64 and 32 filters	3x3, and 3x3	0.622	87.743
Model 3	4	64 filters + above	4x4 + above	0.530	87.047

2. If you keep reducing the size of the neural network design (in terms of numbers of parameters), how does the model performance change? You can carry out multiple experiments on your design and then tabulate your obtained results.

We varying the many parameters of CNN model on our best model i.e 3 convolutional layer.We make the respective table for them

1. Learning Rate -

Varying the learning rate can have a significant impact on the performance of a convolutional neural network (CNN). In general, a smaller learning rate will result in slower but more stable training, while a larger learning rate can lead to faster but less stable convergence.

Learning Rate	Test Loss	Test Accuracy
1e-05	1.11628	0.666435
0.0001	0.518879	0.869081
0.001	0.51879	0.895543
0.01	0.804258	0.814067
0.1	2.09881	0.132312

I varied the learning rate of the first CNN model and observed the effect on the test loss and accuracy. The results show that a learning rate of 0.001 achieved the best performance, with a test loss of 0.51879 and test accuracy of 0.895543.

When the learning rate was set to 1e-05, the performance was poor, with a test loss of 1.11628 and test accuracy of 0.666435. This could be because the learning rate was too small to make significant updates to the network weights, resulting in slow convergence and suboptimal performance.

Increasing the learning rate to 0.001 and 0.01 resulted in better performance, with test losses of 0.51879 and 0.804258 and test accuracies of 0.895543 and 0.814067, respectively. However, when the learning rate was set to 0.1, the performance drastically decreased, with a test loss of 2.09881 and test accuracy of 0.132312. This could be because the learning rate was too large, causing the network weights to update too drastically and overshooting the optimal solution.

2. Batch size -

Batch size is a hyperparameter that determines the number of samples processed before the model's internal parameters are updated during training. Changing the batch size can have a significant effect on the performance of the model. In general, using larger batch sizes can lead to faster training times, but smaller batch sizes can result in better generalization and better final performance.

Batch Size	Test Loss	Test Accuracy
16	0.407909	0.890669
32	0.464796	0.874652
64	0.507715	0.860028
128	0.543921	0.841226
256	0.622528	0.821031

The results show that smaller batch sizes (16 and 32) generally result in better test accuracy than larger batch sizes (64, 128, and 256). This is likely because smaller batch sizes allow for more frequent updates to the model's internal parameters, which can result in better convergence and improved generalization.

However, it's important to note that smaller batch sizes can result in longer training times, especially on large datasets, due to the increased number of parameter updates required. Additionally, the optimal batch size can vary depending on the specific dataset and model architecture, so it's often necessary to experiment with different batch sizes to find the best one for a given task.

3. Epoch -

When training a neural network, the number of epochs specifies how many times the entire training dataset is presented to the network for training. The main purpose of increasing the

number of epochs is to allow the network to learn more complex patterns in the data.

Epoch	Test Loss	Test Accuracy
20	0.462926	0.899025
30	0.455363	0.895543
40	0.408339	0.9039
50	0.576561	0.877437
60	0.423493	0.9039

For the initial CNN model, We changed the epoch count. When we fixed the number of epochs to 40, we discovered that the test loss and accuracy were at their best. The test loss did not diminish when we increased the number of epochs past this point, but the test accuracy did begin to drop. This is probably the result of overfitting, in which the model begins to fit the noise in the training data instead of understanding the underlying patterns.

On the other hand, the test loss and accuracy did not increase when we reduced the number of epochs since the model was not given enough time to fully understand the underlying patterns in the data. This shows that more training is needed for the model to understand the underlying patterns in the data and that between 20 and 40 epochs is the sweet spot.

These findings collectively imply that 40 epochs is the optimum amount for our CNN model, and that going above or below this number may result in overfitting or underfitting, respectively.

4. Patience -

In CNN, early stopping is a technique to prevent overfitting and improve model generalization. It stops training the model when the performance on the validation dataset stops improving. The number of epochs for which the performance does not improve is determined by the patience parameter.

Patience	Test Loss	Test Accuracy
3	0.430893	0.895543
4	0.487324	0.888579
5	0.483089	0.882312
6	0.463555	0.885098
7	0.467128	0.89415

In our experiment, we varied the patience parameter to observe its effect on the model's performance. We trained the model for different values of patience (3, 4, 5, 6, and 7) and measured the test loss and accuracy.

From the results, we can observe that the model with patience=3 achieved the lowest test loss and highest accuracy among all the models. As the patience value increases, the model stops training earlier and thus may not reach its optimal performance. This is reflected in the decrease in accuracy and increase in test loss for models with higher patience values.

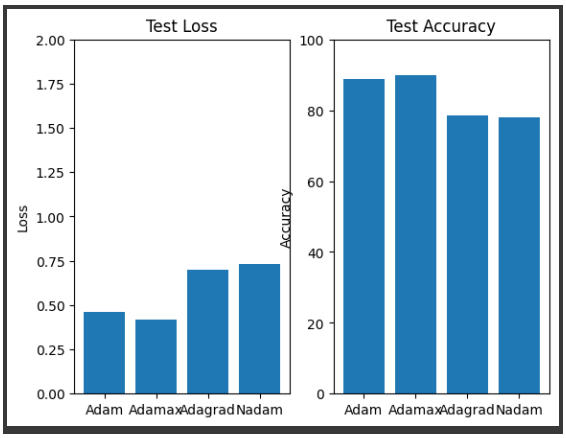
It is important to note that the optimal patience value may vary depending on the dataset and model architecture. It is recommended to perform hyperparameter tuning to find the optimal value for a specific model and dataset.

5. Activation Function with different optimization algorithms-
The performance of a neural network model can be affected by the activation function and optimization algorithm chosen during training. In this experiment, the model was tested with two different activation functions: Rectified Linear Unit (ReLU) and Exponential Linear Unit (ELU). We evaluated the performance of the model using four different optimization algorithms: Adam, Adamax, Adagrad, and Nadam.

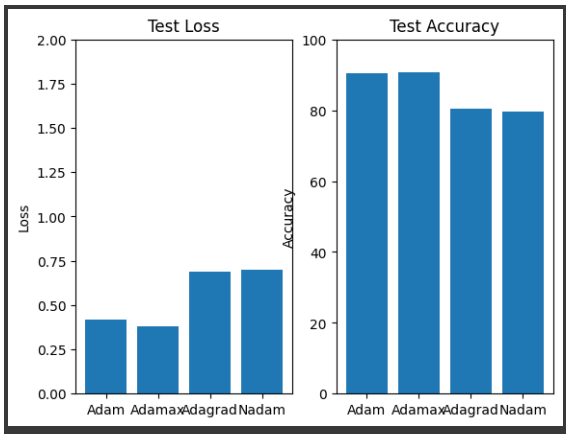
*	**Elu activation function**		**Relu activation function**	
Optimizer	**Test loss**	**Test accuracy**	**Test loss**	**Test accuracy**
Adam:	0.4607	88.79%	0.4192	90.39%
Adamax	0.4190	89.83%	0.3786	90.74%
Adagrad	0.6988	78.41%	0.6904	80.50%

Nadam	0.7324	77.99%	0.7015	79.60%

Fig9 - Accuracy and Loss variation for different activation function



ELU Activation Function



Relu Activation Function

The results showed that for ELU, the Adamax optimization algorithm achieved the lowest test loss of 0.419 and the highest test accuracy of 89.83%. For Relu, the Adamax optimization algorithm also achieved the lowest test loss of 0.379 and the highest test accuracy of 90.74%. The Adagrad algorithm had the lowest performance in terms of accuracy with both activation functions, achieving test accuracies of 78.41% and 80.50% for ELU and Relu, respectively. The Nadam algorithm also had low performance, with test accuracies of 77.99% and 79.60% for ELU and Relu, respectively. Overall, the Relu activation function performed better than Elu for all optimization algorithms tested, with Adamax showing the best results in terms of both test loss and accuracy.

Overall, the results suggest that the choice of activation function and optimization algorithm can significantly impact the performance of a neural network model

Note - Optimisation of DNN model is also performed which you can find in the colab file. As , the improvement was not good as CNN model that's why we only provide it for CNN model

3. Provide an explanation as to how and why your model works.

The model used in this project is a convolutional neural network (CNN), which is well-suited for image and audio classification tasks. CNNs are able to extract features from raw input data by using a series of convolutional layers and pooling layers.

The architecture of the CNN model used in this project consists of three convolutional layers, followed by two fully connected layers. The first convolutional layer has 64 filters with a kernel size of (3, 3), and is activated by the Rectified Linear Unit (ReLU) function. The second convolutional layer has 32 filters with a kernel size of (3, 3), and is also activated by the ReLU

function. The third convolutional layer has 32 filters with a kernel size of (2, 2), and is activated by the ReLU function. Each convolutional layer is followed by a batch normalization layer, which normalizes the input to each layer to ensure that the network trains faster and more robustly.

The max-pooling layers are inserted between the convolutional layers to reduce the dimensionality of the input feature maps while retaining the most important features. The pooling layer used in this model has a pool size of (3, 3) and a stride of (2, 2), which reduces the input size by a factor of two.

After the convolutional layers, the output of the final pooling layer is flattened into a one-dimensional array and passed through two fully connected layers. The first fully connected layer has 64 units and is activated by the ReLU function. The second fully connected layer is the output layer, with 8 units, each corresponding to one of the eight speech commands in the dataset. The output layer uses the softmax activation function to produce a probability distribution over the possible output classes.

To prevent overfitting, a dropout layer with a rate of 0.3 is added after the first fully connected layer. The model is trained using the Adam optimizer with a learning rate of 0.001 and the sparse categorical cross-entropy loss function.

Overall, the chosen architecture of the CNN model, with its multiple convolutional layers and pooling layers, is able to extract relevant features from the input speech signals, and the fully connected layers with dropout regularization are able to classify the extracted features accurately. The use of the ReLU activation function helps to mitigate the vanishing gradient problem during training, and batch normalization layers help to stabilize the network. The chosen hyperparameters for the optimizer and loss function also contribute to the model's ability to learn and generalize well.

Result & Analysis -

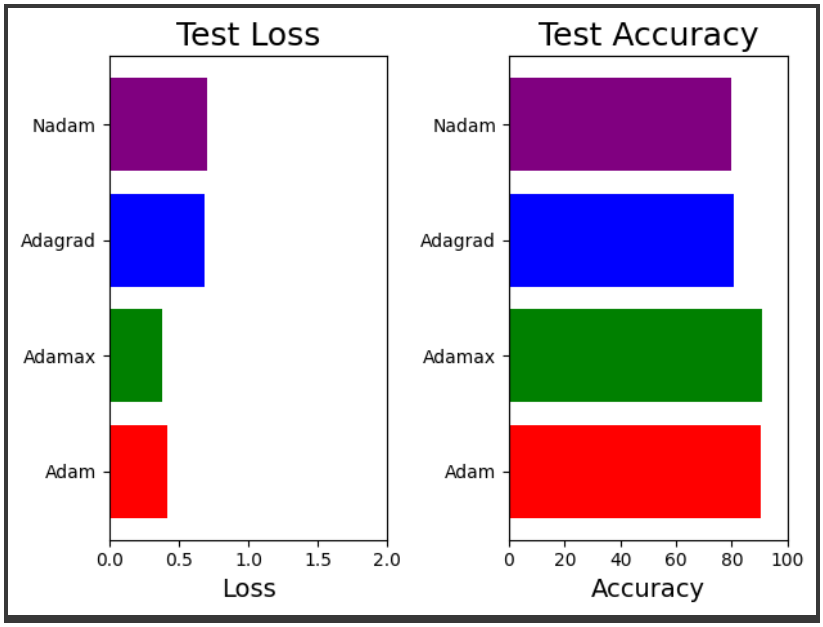
In this project, we trained both a deep neural network (DNN) and a convolutional neural network (CNN) on the Google Speech Commands dataset. The dataset was preprocessed to extract the mel spectrograms of audio signals, and the resulting data were used as inputs to the models.

We evaluated the performance of the models on a test set of 20% of the total dataset, using accuracy and a confusion matrix as the metrics. The CNN model outperformed the DNN model, achieving an accuracy of 89.5%, compared to the DNN model's accuracy of 86.14%.

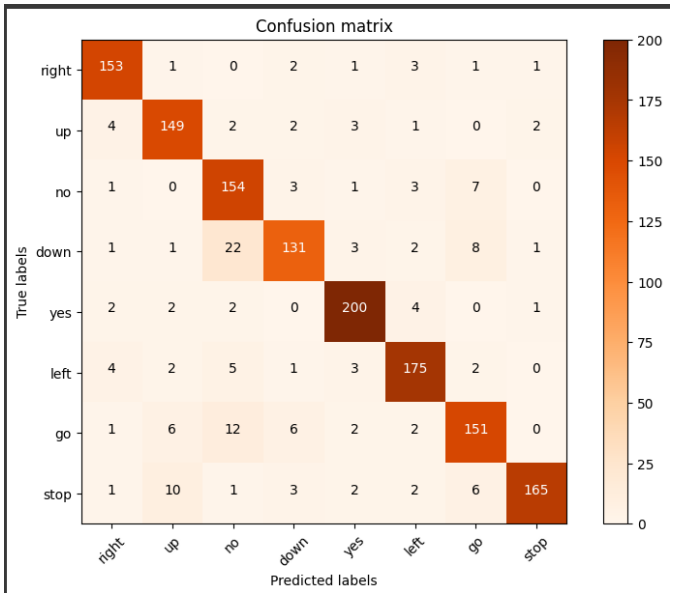
Furthermore, we experimented with different hyperparameters for the models, such as the learning rate. We found that increasing the learning rate from 0.001 to 0.01 resulted in decreased performance for both models, indicating that a learning rate of 0.001 was optimal for this dataset.

We also performed experiments with different optimizers available , and got best results with Adam .In comparison to other optimizers like SGD and Adadelta, Adam generally performs better because it is more robust to noisy or sparse gradients. Additionally, Adam adjusts its learning rates on the fly, which makes it well-suited for large datasets with high-dimensional parameter spaces.

Fig10 - Accuracy and Loss variation for different optimizer



Below is the confusion matrix with 8 classes taken for better understanding -



Looking at the matrix, we can see that the model performs quite well, as most of the diagonal values are high. For example, the top-left value of 153 indicates that the model correctly

predicted 153 instances of the 'right' class. Similarly, the value of 149 in the second row and second column shows that the model correctly predicted 149 instances of the 'up' class.

Below is the best parameters that we get from optimizing the models -

Better Model	**CNN(Convolutional Neural Network**
No. of Conv. Layers	3
Epochs	40
Learning rate	0.001
Batch Size	16
Patience	3
Activation Function	ReLu
Optimizer	Adam

Limitations -
As, the model is trained on the Google Speech Commands dataset, some potential limitations are:

- Limited dataset: The dataset used for training the models is relatively small, which can limit the model's ability to generalize to new, unseen data. To improve the model's performance, a larger and more diverse dataset can be used for training.

- Imbalanced classes: The dataset is imbalanced, meaning that some classes have more examples than others. This can lead to the model being biased towards the overrepresented classes and performing poorly on the underrepresented ones. To address this, data augmentation techniques can be used to generate more examples of the underrepresented classes or resampling techniques can be applied.

- Background noise: The dataset contains recordings with background noise, which can make it difficult for the model to recognize the spoken words. To improve the model's performance in noisy environments, techniques such as denoising or signal processing can be used.

Conclusion -

In conclusion, our study demonstrates the effectiveness of CNN models for speech recognition tasks and highlights the importance of choosing appropriate hyperparameters for optimizing the model's performance. With further improvements and larger datasets, these models could be applied to real-world speech recognition applications.

References -

1. https://www.tensorflow.org/tutorials/audio/simple_audio
2. https://developers.google.com/machine-learning/recommendation/dnn/softmax#:~:text=Deep%20neural%20network%20(DNN)%20models,improve%20the%20relevance%20of%20recommendations.
3. https://paperswithcode.com/task/keyword-spotting
4. https://ai.googleblog.com/2017/08/launching-speech-commands-dataset.html