DL Model Optimization for Lightweight Visual Wake Words Task

Embedded Systems Course Project

Mruganshi Gohel (B20CS014)

Ghelani Shubham (B20EE019)

Introduction

The Lightweight Visual Wake Words (LVWW) task involves the development of an artificial intelligence (AI) model that can automatically identify whether a person is present or not in the camera input of mobile phones or microcontrollers. This is a valuable application for a range of use cases such as security systems, automated attendance systems, and monitoring applications.

The LVWW task is particularly challenging as the model needs to be optimized for deployment on resource-constrained devices with limited computing power and memory. Therefore, the model must be lightweight and efficient, while maintaining high accuracy in detecting people.

About the topic and Plan of Implementation:

In this project, we need to build a model which can automatically detect whether a person is present or not in a given image. The image is captured through the camera which can be embedded in a microcontroller, embedded system, or computer. Here, we develop a Deep Learning model which performs all the computations on the microcontroller itself and provides real-time output. This can have a wide range of practical applications in various fields, such as smart homes, security systems, and industrial automation, where real-time image classification tasks are required on low-power and low-cost embedded devices. For example, the optimized deep learning model can be deployed on a microcontroller-based security camera to detect and alert the user of any person entering a restricted area in real time. Based on the features of the image and its configuration, we need deep learning models to detect whether the person is present or not in the image. The plan of implementation was as follows:

- Stabilizing and reducing the dataset
- Preprocessing the dataset
- Building the Model, Training, and Testing the Model
- Model Robustness Check
- Analyzing the Neural Network Design by changing the size of the neural network
- Observing and concluding the final results

Dataset

Balancing and Reduction of Dataset

The dataset is derived from COCO2017 and reduced to ~150MB using the Python script shown below as the original dataset is of ~15GB train, ~10GB test, highly unbalanced, and contains duplicate data which may affect badly the result while training :

```
import pandas as pd
import json
import numpy as np
import os

an = json.load(open('')) # link to json file from COCO official website
an = an['annotations']  # annotations hold category information

# create a new data frame to hold information on images and corresponding categories
col = ['image-id', 'category-id']
data = pd.DataFrame(columns=col)

for i in range(len(an)):
    id = an[i]['image_id']
    imgname = str(id).zfill(12)   # pad with zeros to match image name in COCO folder
    data = data.append( {'image-id' : imgname,
                     'category-id' : an[i]['category-id']}, ignore_index=True)
                      #add a row to a data frame

# Multiple instances of the same object category in an image are stored as separate rows
# deleting them
data = data.drop_duplicates(data.columns)

# Cleaning the data frame content
img_names = np.unique(data['image-id']) # get set of image names
```

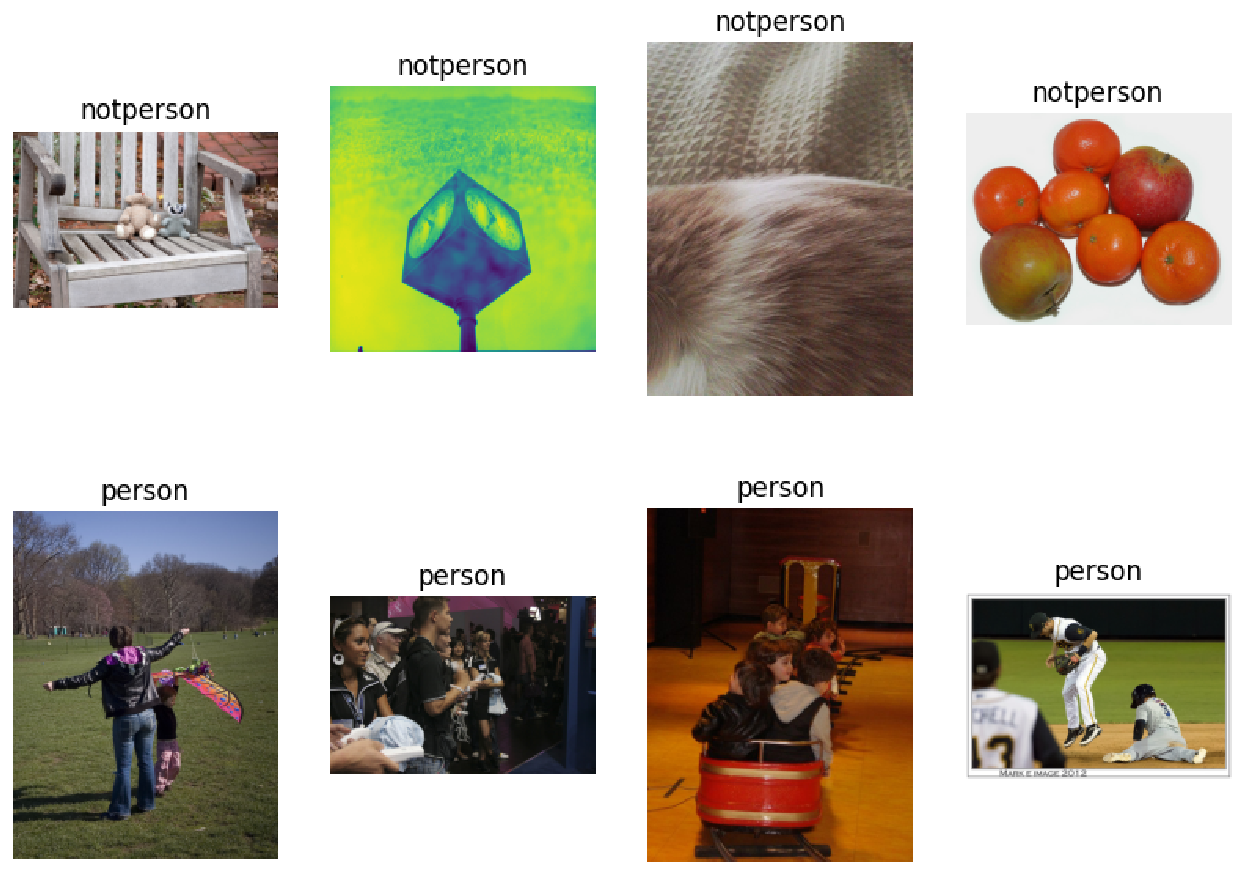
```
for i in img_names:
    # If the image contains an element under the person category (category #1)
    if len(data[(data['image-id'] == str(i)) & (data['category-id'] == 1)]) != 0:
        # Remove rows that indicate the presence of other category elements in the image
        indices = data[(data['image-id'] == str(i)) & (data['category-id'] != 1)].index
        if(len(indices) != 0):
            data = data.drop(index=indices)

# separate into 2 dataframes
data_person = data[data['category-id'] == 1]
data_not = data[data['category-id'] != 1]

# set category_id = 0 to indicate not_person
data_not['category-id'] = 0

# there may be several not_person elements leading to redundant rows, eliminating them
data_not = data_not.drop_duplicates()

path = ''   # path to store the categorized image files
curpath = ''    # path to COCO image files

# make appropriate folders
os.mkdir(path + 'vww')
os.mkdir(path + 'vww/person')
os.mkdir(path + 'vww/notperson')

pt = path + 'vww/person/'
npt = path + 'vww/notperson/'

datap = list(data_person['image-id'])
datanp = list(data_not['image-id'])

n = 300 # number of images to pick from each category

for i in range(n):
   os.rename(curpath + str(dfp[i]).zfill(12) + '.jpg',
                          pt + str(dfp[i]).zfill(12) + '.jpg')
   os.rename(curpath + str(dfnp[i]).zfill(12) + '.jpg',
                          npt + str(dfnp[i]).zfill(12) + '.jpg')
```

We begin by loading the JSON file and annotations from COCO that contains the dataset's category information. We will then create a new data frame to hold the information on the images and corresponding categories, preprocess it (i.e., remove duplicate data, clean the data frame content, separate it into two categories (person and non-person), and create a folder called "vww" that will contain the dataset) and separate it into two categories (person and non-person).

Two sub-folders, person and nonperson, each holding photographs of a certain sort, are included in the dataset folder. It is a balanced dataset because there are 300 photos in each folder. Using the Keras.preprocessing module's image_dataset_from_directory() method, the photos are loaded by providing the path to the image folder, the dataset type (train/test), and the split ratio (80:20).

Visualization

Lets look into a few images in the dataset -



Preprocessing

Normalizing the pixel values

The pictures' pixel values range from 0 to 255. Speeding up model coverage is achieved by scaling them to load values in the [0,1] range. The rescaling feature of the keras.layers.experimental.preprocessing module is used for this. Here is some code:

```
normalization_layer = layers.experimental.preprocessing.Rescaling(1./255)
train_data = train_data.map(lambda x, y: (normalization_layer(x), y))
val_data = val_data.map(lambda x, y: (normalization_layer(x), y))
```

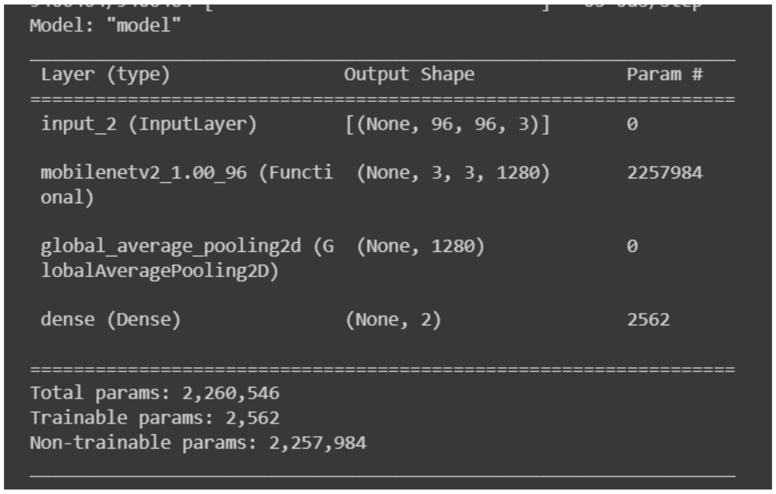
The Model

We are utilizing transfer learning to train the dataset, which is the idea of using the information discovered while resolving one problem to another. The size of the model's input is specified as (256,256,3). The classification layer, the last layer of the DenseNet121 model, is not included; instead, it is appended with a GlobalAveragePooling layer, a Dense layer with softmax activation, and an equal number of nodes as classes.

The model is set up so that the appended layers are the only portions of the model that can be trained. This indicates that the weights of the nodes in the DenseNet121 architecture stay the same as the training epochs go on.

Because the final layer of the model has the softmax activation function and the outputs of the model are not one-hot encoded here, the model is constructed using the sparse categorical cross-entropy loss function. The Adam optimizer is employed, and the model's correctness is monitored over time.

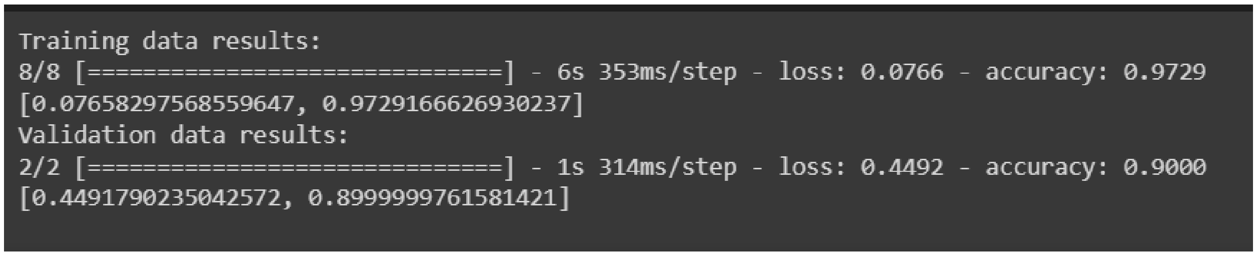
The EarlyStopping callback function of the Keras.callbacks module monitors the validation loss and stops the training if it doesn’t for 5 epochs continuously. The restore_best_weights parameter ensures that the model with the least validation loss is restored to the model variable.



```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 96, 96, 3)]       0

 mobilenetv2_1.00_96 (Functi  (None, 3, 3, 1280)       2257984
 onal)

 global_average_pooling2d (G  (None, 1280)             0
 lobalAveragePooling2D)

 dense (Dense)               (None, 2)                 2562

=================================================================
Total params: 2,260,546
Trainable params: 2,562
Non-trainable params: 2,257,984
_________________________________________________________________
```

Results

The model was able to achieve ~97% accuracy on training and ~90% accuracy on validation data with a learning rate of 0.01.

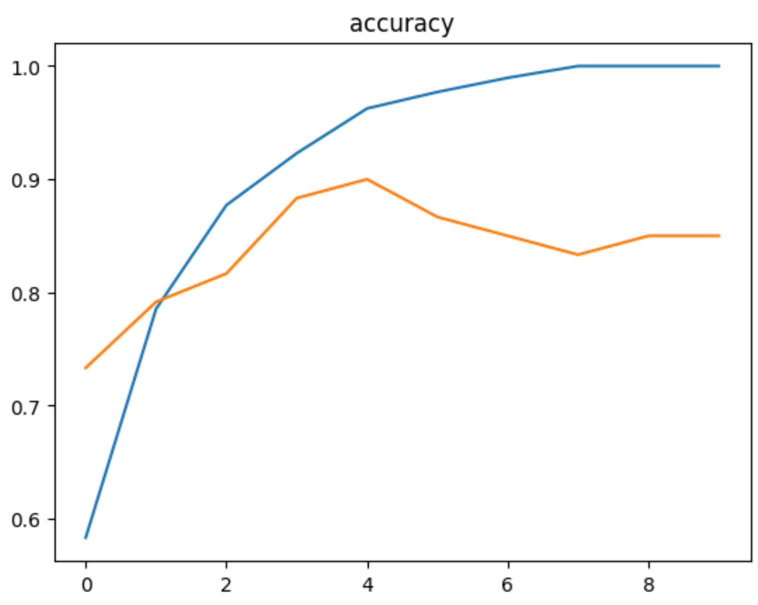


```
Training data results:
8/8 [==============================] - 6s 353ms/step - loss: 0.0766 - accuracy: 0.9729
[0.07658297568559647, 0.9729166626930237]
Validation data results:
2/2 [==============================] - 1s 314ms/step - loss: 0.4492 - accuracy: 0.9000
[0.4491790235042572, 0.8999999761581421]
```

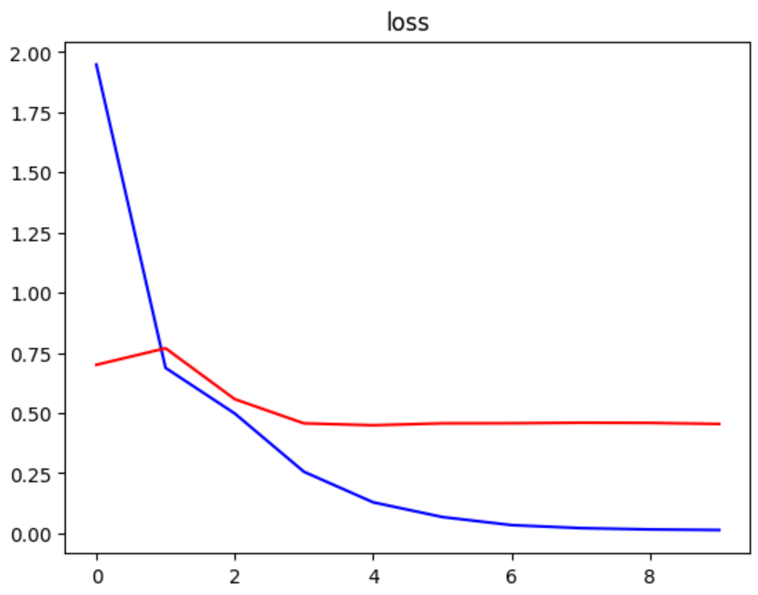
Metrics

here Blue line is showing Training accuracy/loss and the red line is showing Validation accuracy/loss.

- **Accuracy**

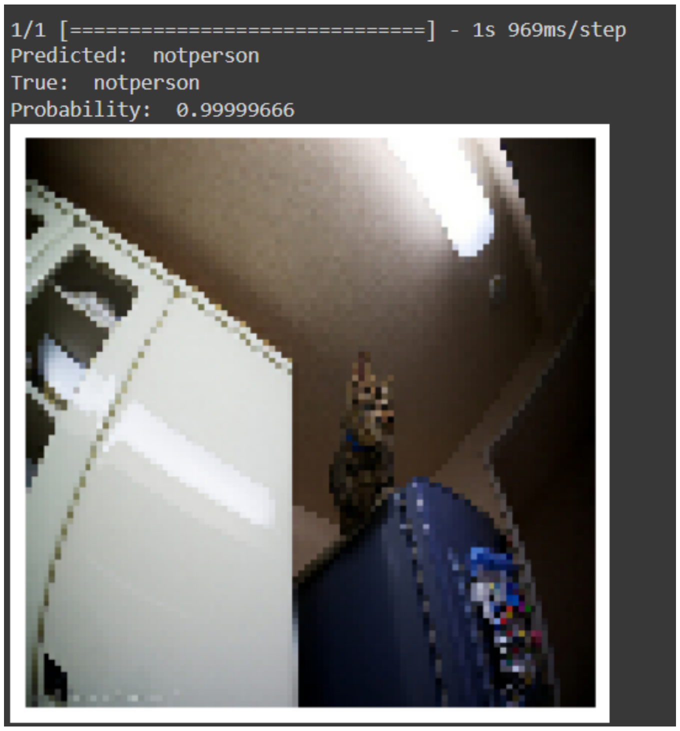


- Loss



Prediction

Now let’s take a random image from the validation set and check whether the data is properly and with the correct model trained or not.



```
1/1 [==============================] - 1s 969ms/step
Predicted:  notperson
True:  notperson
Probability:  0.99999666
```

Model Robustness Checking

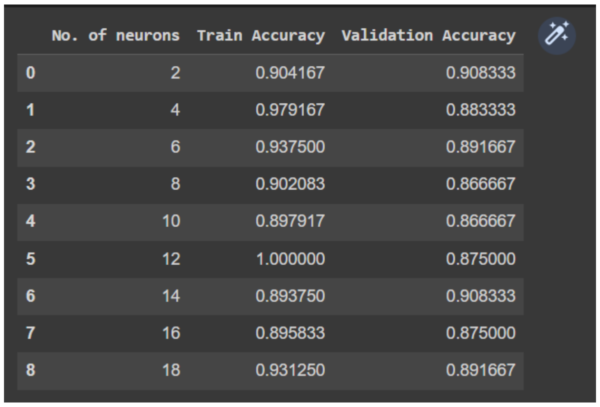
To check the robustness of the model, we performed different experiments on our original neural network design by changing different parameters. We choose different parameters in such a way that changing those parameters will impact neural network design in different ways and ultimately that will change in accuracy.

The parameters that will affect the neural network's design, visualization, training/validation accuracy, and training/validation loss are provided below in tabular form.

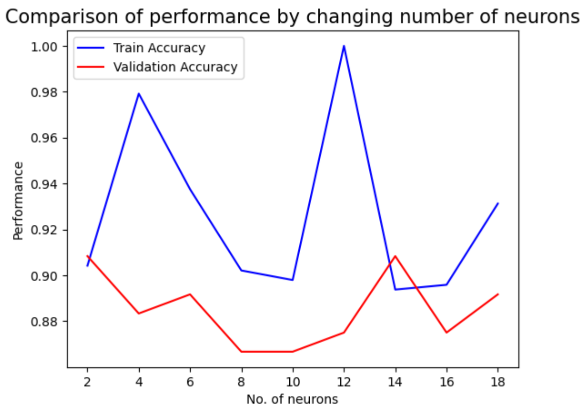
1. Number of neurons

Increasing the number of neurons can increase the capacity of the network, allowing it to learn more complex patterns. However, if the number of neurons is too high, it can lead to overfitting and poor generalization of new data. The number of neurons in the output layer of a neural network depends on the specific task and the number of classes to be predicted.

We changed a number of neurons in a layer from 2 to 18 and check its impact on accuracy, which is shown in tabular form and visualized using a line graph.



	No. of neurons	Train Accuracy	Validation Accuracy
0	2	0.904167	0.908333
1	4	0.979167	0.883333
2	6	0.937500	0.891667
3	8	0.902083	0.866667
4	10	0.897917	0.866667
5	12	1.000000	0.875000
6	14	0.893750	0.908333
7	16	0.895833	0.875000
8	18	0.931250	0.891667

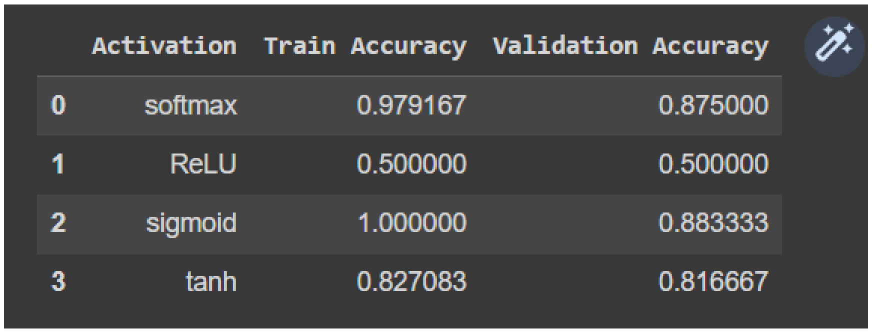


We can see from the graph that there is a huge drop in the accuracy for some values of neurons this is due to overfitting. and hence we will not take those values.

2. Activation Function

The choice of activation function can impact the network's ability to model complex non-linear relationships. For example, using a ReLU activation function can speed up training and improve performance, while using a sigmoid activation function can help prevent vanishing gradients. However, the choice of activation function can also depend on the specific task and the dataset.

We experiment using 4 activation functions in a layer which are **softmax, ReLU, sigmoid, and tanh,** and checked their impact on training and validation accuracy which is shown in tabular form below.

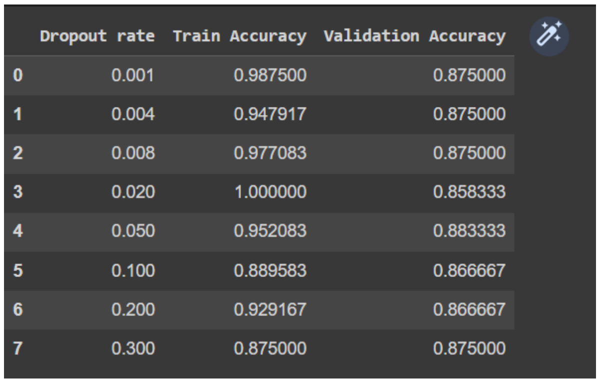


	Activation	Train Accuracy	Validation Accuracy
0	softmax	0.979167	0.875000
1	ReLU	0.500000	0.500000
2	sigmoid	1.000000	0.883333
3	tanh	0.827083	0.816667

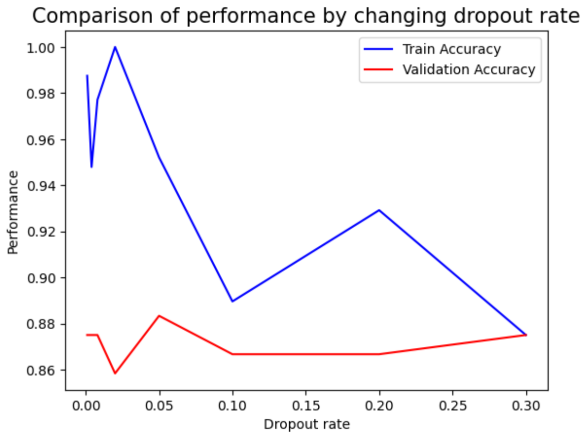
3. Dropout Rate

Dropout is a regularization technique that randomly drops out some neurons during training, which can prevent overfitting. Increasing the dropout rate can lead to better generalization, but too many dropouts can also reduce the network's capacity and accuracy. By randomly dropping out nodes during training, the network becomes less sensitive to the specific weights of any individual node. We can adjust the dropout rate to increase or decrease the amount of dropout regularization applied to the network.

We changed the Dropout rate from very small to big and visualization and accuracies are shown below in tabular form.



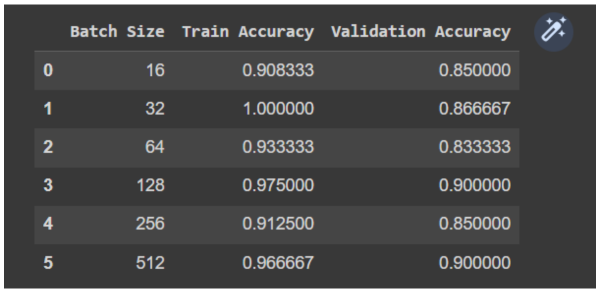
	Dropout rate	Train Accuracy	Validation Accuracy
0	0.001	0.987500	0.875000
1	0.004	0.947917	0.875000
2	0.008	0.977083	0.875000
3	0.020	1.000000	0.858333
4	0.050	0.952083	0.883333
5	0.100	0.889583	0.866667
6	0.200	0.929167	0.866667
7	0.300	0.875000	0.875000



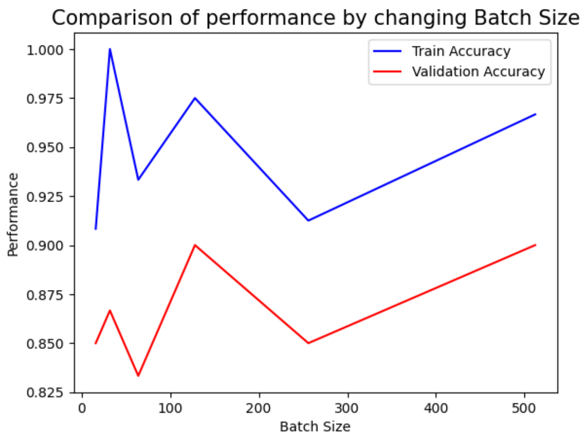
4. Batch Size

The batch size determines the number of samples processed in each training iteration. A larger batch size can reduce the variance of the gradient estimates and speed up training, but it can also require more memory and reduce the network's ability to generalize. Changing the batch size can indirectly affect the robustness of the model by influencing the training process and potentially affecting the model's generalization ability.

If the model's accuracy remains stable or improves with different batch sizes, it suggests that the model is robust to changes in the batch size.



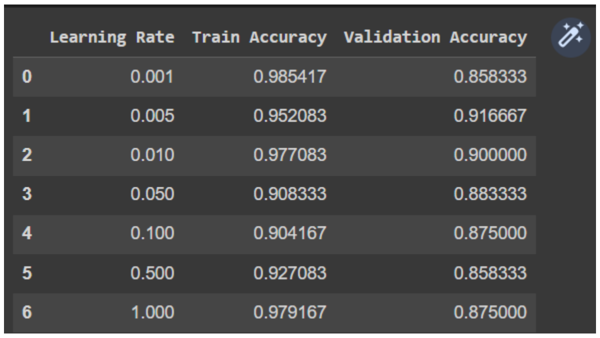
	Batch Size	Train Accuracy	Validation Accuracy
0	16	0.908333	0.850000
1	32	1.000000	0.866667
2	64	0.933333	0.833333
3	128	0.975000	0.900000
4	256	0.912500	0.850000
5	512	0.966667	0.900000



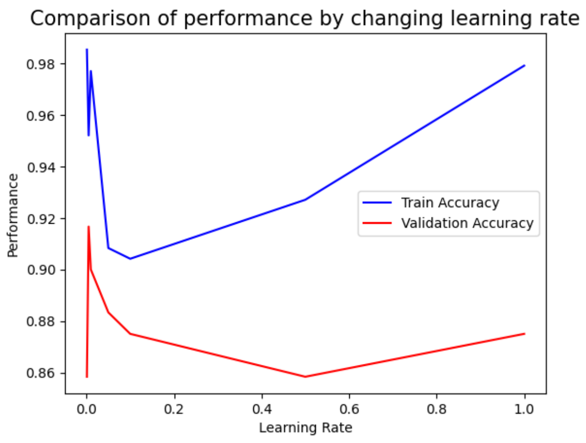
5. Learning Rate

The Learning Rate determines the step size at which the optimizer updates the weights during training. A high learning rate may result in overshooting the optimal weights, while a low learning rate may result in slow convergence and getting stuck in local minima.

To check the robustness of a model with respect to the learning rate, we tried training the same model with different learning rates and compare the training and validation accuracies. If the model is robust to the learning rate, then we would be able to see similar performance across different learning rates.

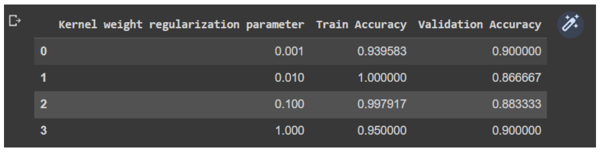


	Learning Rate	Train Accuracy	Validation Accuracy
0	0.001	0.985417	0.858333
1	0.005	0.952083	0.916667
2	0.010	0.977083	0.900000
3	0.050	0.908333	0.883333
4	0.100	0.904167	0.875000
5	0.500	0.927083	0.858333
6	1.000	0.979167	0.875000

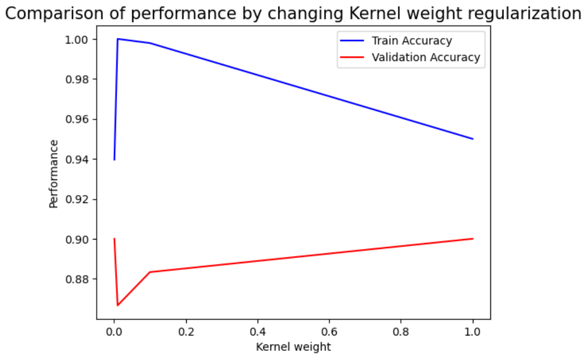


6. Weight Regularization

Weight Regularization is another technique used to prevent overfitting by adding a penalty term to the loss function that encourages the weights to have smaller values. Changing the weight regularization kernel can affect the magnitude. We defined a list of regularization parameters to test, and then loop through each parameter to create a new model with that regularization parameter. Each model is then trained and evaluated, and the training and validation accuracies are recorded for each model.



	Kernel weight regularization parameter	Train Accuracy	Validation Accuracy
0	0.001	0.939583	0.900000
1	0.010	1.000000	0.866667
2	0.100	0.997917	0.883333
3	1.000	0.950000	0.900000



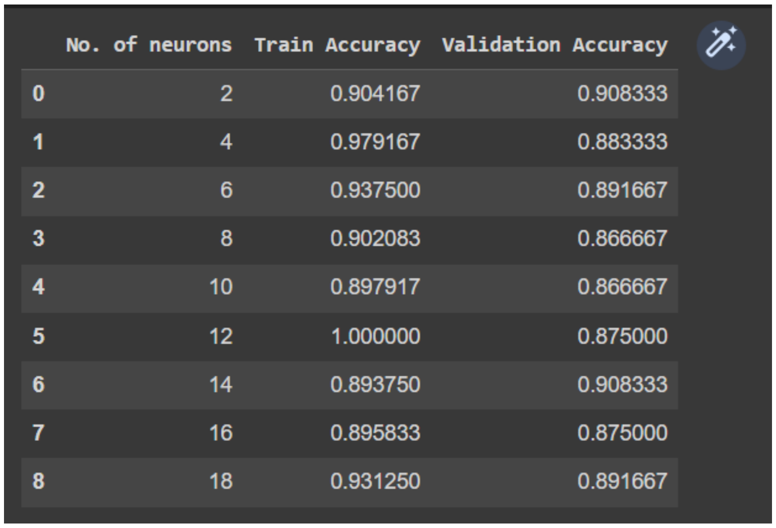
Reducing the size of Neural Network and observing performance

Reducing the size of the neural network design generally results in a decrease in model performance, as the model has fewer parameters to learn from and may not be able to capture the complexity of the data as well. However, the impact of reducing the size of the neural network can vary depending on the specific dataset and task.

To carry out experiments and observe the impact of reducing the size of the neural network design, we have considered changing the following parameters:

1. Number of Neurons per layer

By reducing the number of neurons per layer, the model will have fewer parameters to learn from and may not be able to capture the complexity of the data as well.

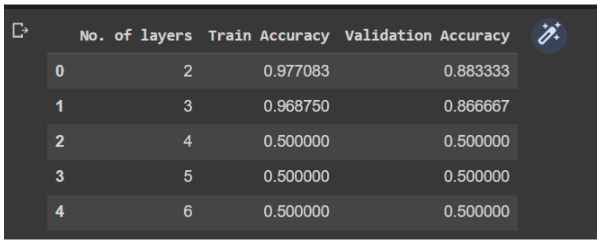


	No. of neurons	Train Accuracy	Validation Accuracy
0	2	0.904167	0.908333
1	4	0.979167	0.883333
2	6	0.937500	0.891667
3	8	0.902083	0.866667
4	10	0.897917	0.866667
5	12	1.000000	0.875000
6	14	0.893750	0.908333
7	16	0.895833	0.875000
8	18	0.931250	0.891667

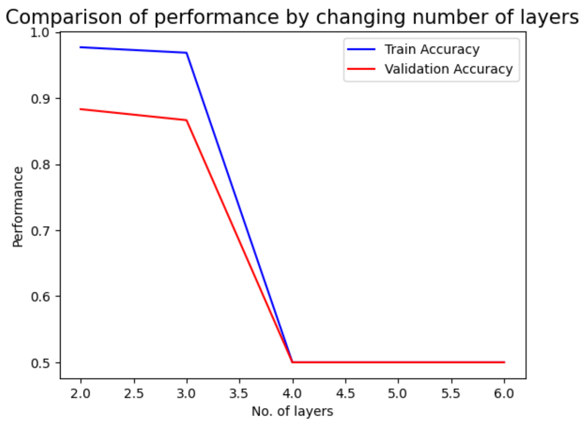
It seems from the table that reducing the number of neurons can have a negative impact on validation accuracy. This could be because reducing the number of neurons can limit the model's capacity to learn complex patterns in the data, which can result in lower accuracy. But the optimal number of neurons can vary depending on the complexity of the task and the amount of data available for training.

2. Number of Layers in Neural Network

The depth multiplier controls the number of layers in the network by reducing the number of filters in each layer. It effectively scales the depth of the network, with a smaller depth multiplier resulting in a shallower network and a larger depth multiplier resulting in a deeper network. However, changing the depth multiplier also affects the performance of the network. A smaller depth multiplier reduces the number of parameters and computational cost, but may also reduce the network's accuracy. By reducing the number of layers, the model will have fewer parameters to learn from and may not be able to capture the complexity of the data as well.

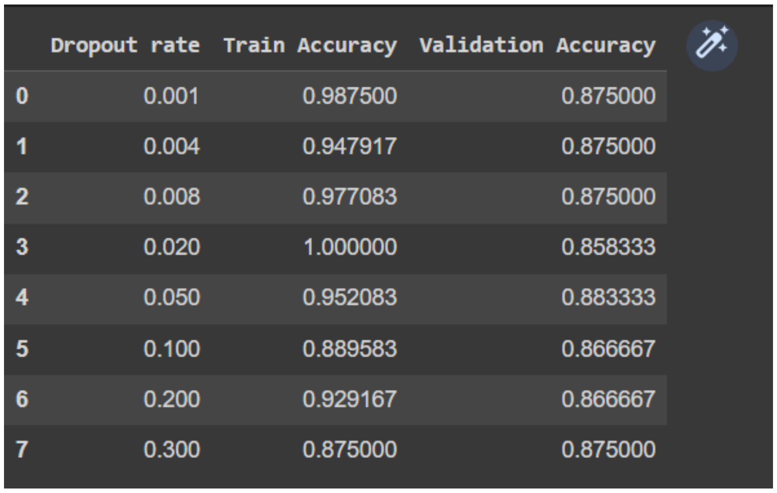


	No. of layers	Train Accuracy	Validation Accuracy
0	2	0.977083	0.883333
1	3	0.968750	0.866667
2	4	0.500000	0.500000
3	5	0.500000	0.500000
4	6	0.500000	0.500000



3. Dropout Rate

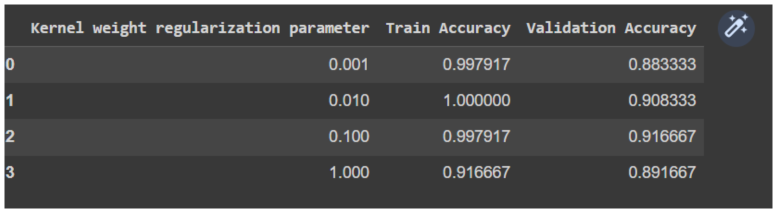
In general, increasing the dropout rate can lead to smaller effective network size and prevent overfitting. However, too high of a dropout rate can lead to underfitting and poor performance on both the training and validation sets. The optimal dropout rate depends on the specific network architecture and dataset being used and should be determined through experimentation.



	Dropout rate	Train Accuracy	Validation Accuracy
0	0.001	0.987500	0.875000
1	0.004	0.947917	0.875000
2	0.008	0.977083	0.875000
3	0.020	1.000000	0.858333
4	0.050	0.952083	0.883333
5	0.100	0.889583	0.866667
6	0.200	0.929167	0.866667
7	0.300	0.875000	0.875000

4. Kernel Weight Regularization

The kernel regularization parameter affects the size of the neural network by controlling the magnitude of the weights in the network. By decreasing the regularization parameter, the magnitude of the weights in the network will increase, which may cause overfitting and increase the size of the network. Conversely, increasing the regularization parameter will shrink the weights and make the network smaller, reducing the risk of overfitting. However, increasing the regularization parameter too much can lead to underfitting and reduced performance.

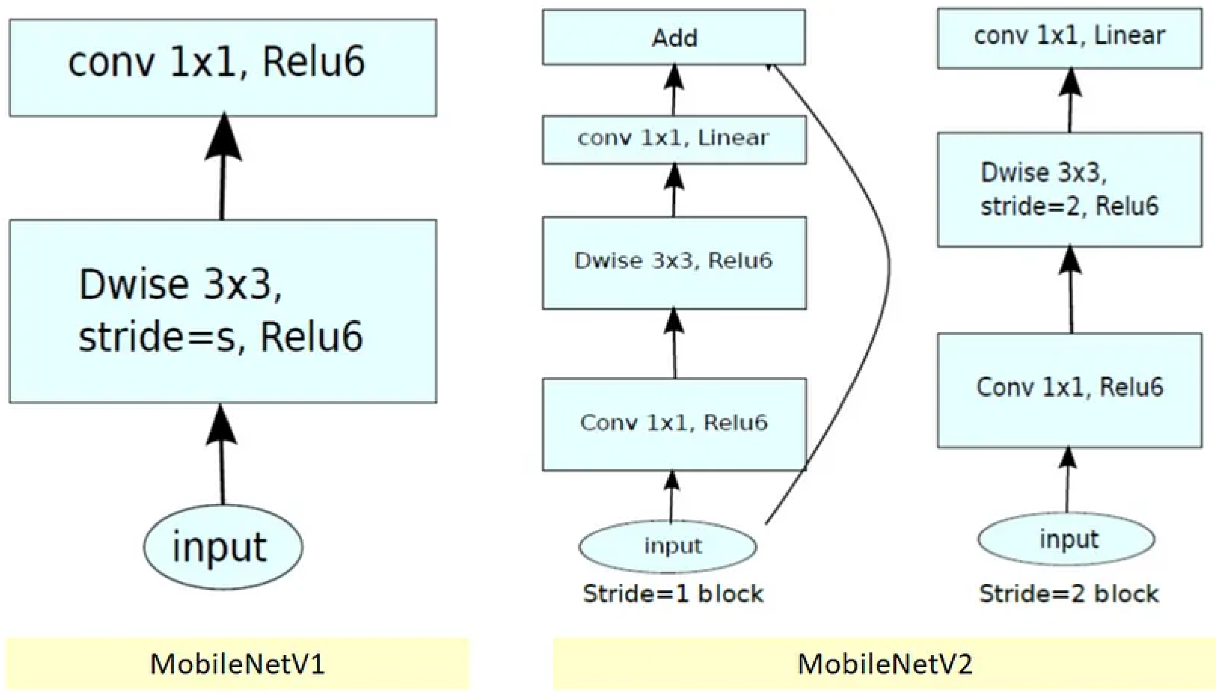


	Kernel weight regularization parameter	Train Accuracy	Validation Accuracy
0	0.001	0.997917	0.883333
1	0.010	1.000000	0.908333
2	0.100	0.997917	0.916667
3	1.000	0.916667	0.891667

How and Why Does the model work?

The model is based on transfer learning using the MobileNetV2 architecture that was pre-trained on the ImageNet dataset. The pre-trained weights of MobileNetV2 serve as an effective initialization for the model, which is then fine-tuned on the target dataset. The use of transfer learning allows for quicker and more efficient training by leveraging the learned features from a large, general dataset.

In this model, the pre-trained MobileNetV2 model is used as a feature extractor. The model is set to non-trainable so that the pre-trained weights are not updated during the fine-tuning process, which ensures that the learned features are preserved. The last layer of MobileNetV2 is removed, and a global average pooling layer is added to produce a 1D tensor output. The output is then passed through a dense layer with sigmoid activation to produce the final classification probabilities for the input image.



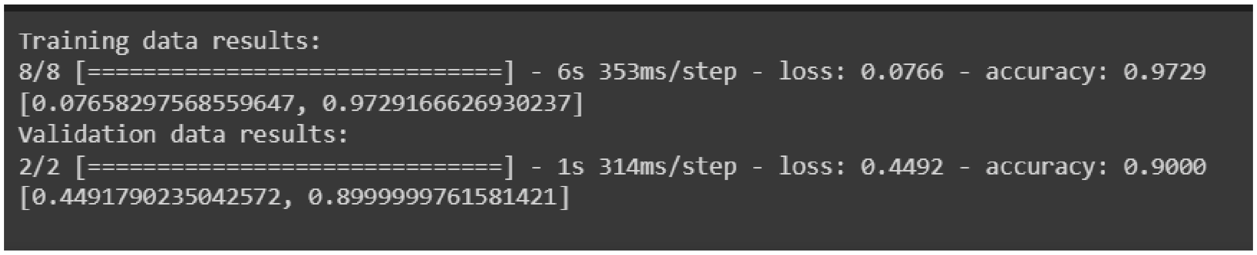
The choice of sigmoid activation function is appropriate for binary classification tasks, as it outputs probabilities between 0 and 1. The use of sparse categorical cross-entropy loss is appropriate for multi-class classification tasks with integer labels, as in this case. The Adam optimizer with a learning rate of 0.01 is chosen for optimization, which is a widely used optimizer in deep learning and has shown good results in practice.

The training is performed for 32 epochs with early stopping and validation-based model checkpointing to prevent overfitting. The model is evaluated on the training and validation datasets to check the performance.

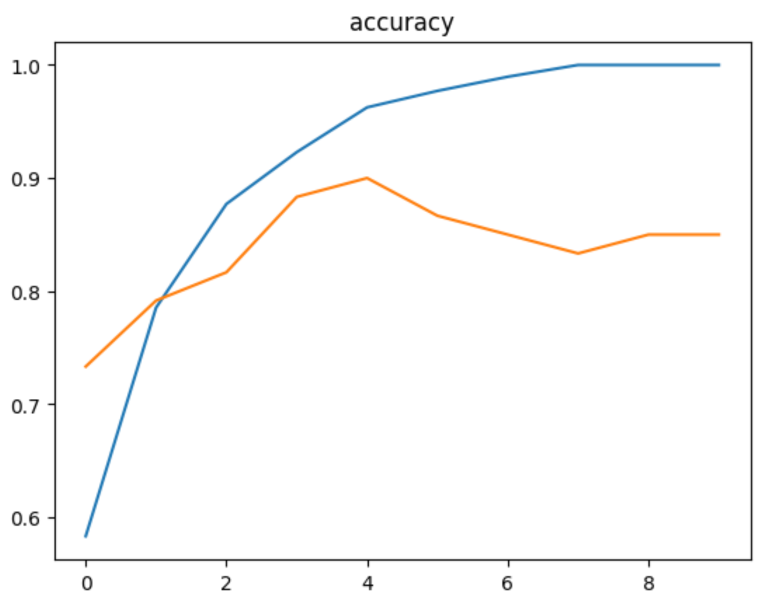
The choice of MobileNetV2 architecture, sigmoid activation, and Adam optimizer with a learning rate of 0.01 has shown good results in practice, which is reflected in the reported accuracy results. The use of transfer learning allows for quicker training and better performance by leveraging the learned features from the pre-trained model. Overall, the model provides a good balance between model complexity and performance, making it a suitable choice for this classification task.

Result

The model was able to achieve ~97% accuracy on training and ~90% accuracy on validation data with a learning rate of 0.01.

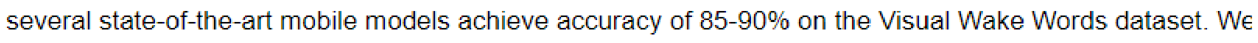


```
Training data results:
8/8 [==============================] - 6s 353ms/step - loss: 0.0766 - accuracy: 0.9729
[0.07658297568559647, 0.9729166626930237]
Validation data results:
2/2 [==============================] - 1s 314ms/step - loss: 0.4492 - accuracy: 0.9000
[0.4491790235042572, 0.8999999761581421]
```



What's innovation?

The original paper which we had to follow says that



several state-of-the-art mobile models achieve accuracy of 85-90% on the Visual Wake Words dataset. We

but we experimented using different models, tried different parameters and see their visualizations, accuracies and after all experiments we chose the best parameters and model that fit the dataset, Not only this but after using this model and parameter every time we picked up a random image from input to predict the result and compared it with original one and every time we got correct output. as a result we get more accuracy than even several state-of-art mobile models which is 97% of training and ~90% of validation.

References

https://github.com/Mxbonn/visualwakewords

https://towardsdatascience.com/review-mobilenetv2-light-weight-model-image-classification-8febb490e61c

https://towardsdatascience.com/visualizing-and-preprocessing-image-dataset-e3ad574f7be6

https://analyticsindiamag.com/a-deep-dive-into-image-data-preprocessing-by-tensorflow/

https://www.quora.com/How-are-smaller-sized-deep-learning-models-i-e-MobileNetv2-EfficientNet-B1-NasNet-Mobile-advantageous-over-larger-sized-models-i-e-VGG-16

https://gist.github.com/AyishaR/ccad108c6c5c4838833766ab63ad8bf7