Capstone Project

Adolfo Moreno

Machine Learning Engineer Nanodegree

January 31, 2020

Simple words speech recognition system

Definition

Project Overview

In 2017 Google's teams TensorFlow and AIY released to the public the dataset "Speech Commands Dataset", containing 65,000 one-second long utterances of 30 short words [1]. This dataset was aimed to provide a simple but robust base for the development and testing of speech recognition systems. Alongside this, they also released a set o example models [2] and sponsor the Kaggle competition "TensorFlow Speech Recognition Challenge" [3] so developers or enthusiast with low knowledge could start right away working and learning.

Following the bases of the Kaggle competition, the aim of the proposed project is to build a functional speech recognition system able to spot the following set of words: *yes, no, up, down, left, right, on, off, stop,* and *go,* plus two labels for *unknown* words and *silence*, so there are 12 possible labels.

Problem Statement

The approach for the solution for the given problem statement is to build, train and test a TensorFlow model for a classification task, having as input a one-second clip and as output one of the 12 possible labels. The task involved will be:

1. Download the dataset and explore the data provided in it. Notebook *"1_Data_Exploration.ipynb".*
2. Build an input-pipeline to feed efficiently the data into the model and define some data augmentations techniques. Notebook *"2_Preprocessing.ipynb".*
3. Evaluate different audio features that could be used as input for the models. Notebook *"3_Feature_Engineering.ipynb".*
4. Build and test different custom models, evaluating its performance on the proposed features. Refinement is applied, improving model architecture and tuning training hyperparameters. Notebook *"4_Model_Training.ipynb".*

Metrics

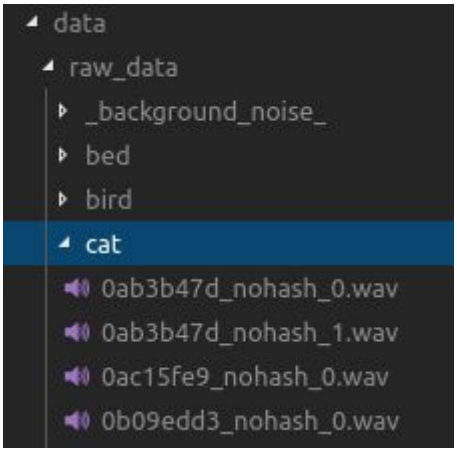
Having a balanced dataset, as we will see in the “Exploratory Visualization” chapter, and following the same metric used to evaluate submissions in the Kaggle’s competition, this project will use Multiclass Accuracy as performance metric for the classification task, multiclass accuracy is simply the average number of observations with the correct label. The goal is that the custom model implemented could reach at least 0.85.

$$accuracy = \frac{correctly\ classified\ samples}{total\ number\ of\ samples}$$

Analysis

Data Exploration

The data to be used in the project include the “Speech Commands Dataset, released by google in 2017. As described before, this dataset contains 65,000 one-second long utterances of 30 short words, by thousands of different people, contributed by members of the public through the AIY website.



The dataset is organized into 31 folders containing the audio files for each word. There are 20 core words ("Yes", "No", "Up", "Down", "Left", "Right", "On", "Off", "Stop", "Go", "Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", and "Nine"), and 10 auxiliary words ("Bed", "Bird", "Cat", "Dog", "Happy", "House", "Marvin", "Sheila", "Tree", and "Wow") to help to distinguish between unrecognized words. There is also included a set of files with background noise.

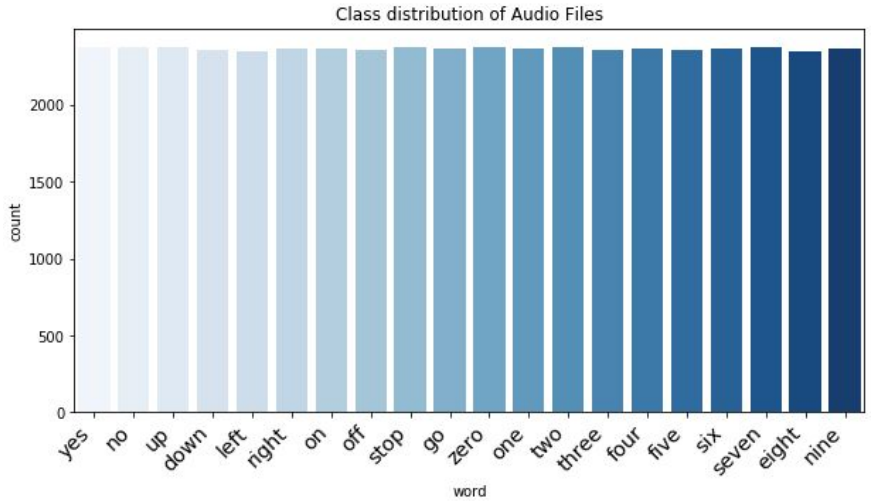
The filename of each file contains information about the user that recorded the audio (first 8 characters), as well as an index that indicate if the word is repeated by the same speaker. This structure is important when performing the train / test / validation split, as we have to ensure homogeneity in the data, for example, keeping the audios from the same speaker in the same set, hopefully Google’s team provided a function to perform this slice that we will discus later.

Be aware that the same filename could be found in different folders as the same speaker could have recorded more than one word.

A detailed description of this dataset can be found at
https://www.kaggle.com/c/tensorflow-speech-recognition-challenge/data

Exploratory Visualization

As show in the graph below, the distribution of the target classes in the given dataset is balanced, since all core words, included those we want to classify, have a similar number of samples, around 2370.



The next table shows the head of a dataset built with the filenames and the number of samples of every file in the dataset. As we can see in the file with index 2, the number of samples is lower than 16000 (sample rate used to record all files), this means that the length of the file is lower than 1 second. Indeed, searching the number of files that do not follow this rule we found that more than 6400 files lay below the 1s. We have to treat specially this files, as we need all the input to have the same shape, one option could be padding the array with zeros.

	label	file	core_word	number_of_samples
0	bed	00176480_nohash_0.wav	False	16000
1	bed	004ae714_nohash_0.wav	False	16000
2	bed	004ae714_nohash_1.wav	False	14861
3	bed	00f0204f_nohash_0.wav	False	16000
4	bed	00f0204f_nohash_1.wav	False	16000

The next chart shows the “describe() “function applied over the files lower than 1 second

```
count     6469.000000
mean     13531.994744
std       1892.299935
min       5945.000000
25%      12288.000000
50%      14118.000000
75%      15019.000000
max      15976.000000
Name: number_of_samples, dtype: float64
```

Algorithms and Techniques

The main approach for this project's solution follows the framework proposed by Google's TensorFlow tutorial [2], which implements several neural network architectures including a low latency CNN based on the paper Convolutional Neural Networks for Small-footprint Keyword Spotting [5]. In this tutorial, audio clips are transform into spectrograms, which are a visual way to represent the spectrum o frequencies of the audio over time, from this point on, the input is treated as a single-channel image and the problem is handled as an image classification task.

Most of the algorithms and techniques applied to the data are described in the notebooks *"2_Preprocessing.ipynb"* and *"3_Feature_Engineering.ipynb"*, in the next sections we will describe the most relevant points.

Reading and decoding the audio

The first step in the input pipeline is to read and load the audio files. Mostly, we will use TensorFlow functions to manipulate and apply any transformation to the data, as most of them are implemented with TPU/GPU-compatible ops and supports gradients. We use "`tf.io.read_file`" to read the audio file and "`tf.audio.decode_wav`" to decode it.

```
data_path = 'data/raw_data'
# Choose one audio clip to work with
bed_audio = os.path.join(data_path, 'bed', '00176480_nohash_0.wav')
```

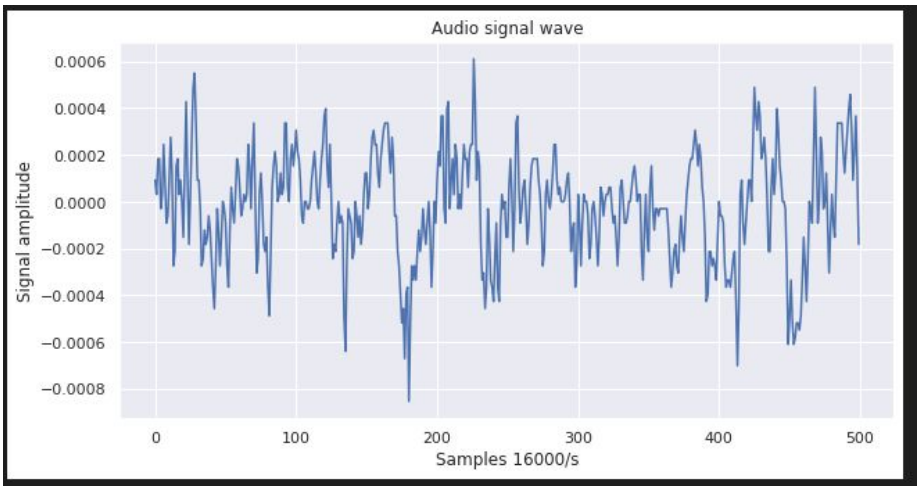
```
#readind the audio file...
raw_audio = tf.io.read_file(bed_audio)
# Let's check what we've got
print(f'raw_audio {type(raw_audio)}')
print(f'raw_audio {raw_audio.numpy()[:20]}') # Print the head(20) of the raw audio
```

```
raw_audio <class 'tensorflow.python.framework.ops.EagerTensor'>
raw_audio b'RIFF$}\x00\x00WAVEfmt \x10\x00\x00\x00'
```

```
# Decoding...
# We will use another handy tf function: audio.decode_wav...
# https://www.tensorflow.org/api_docs/python/tf/audio/decode_wav

# All audio files were recorded at 16000 samples / seconds
sample_rate = 16000
audio, sample_rate = tf.audio.decode_wav(raw_audio,
                                         desired_channels=1,  # mono
                                         desired_samples=sample_rate )

print(f'audio {type(audio)}')
print(f'audio {audio.shape}')
print(f'audio {audio.numpy()[:10]}') # Print the head(10) of the decoded audio
```

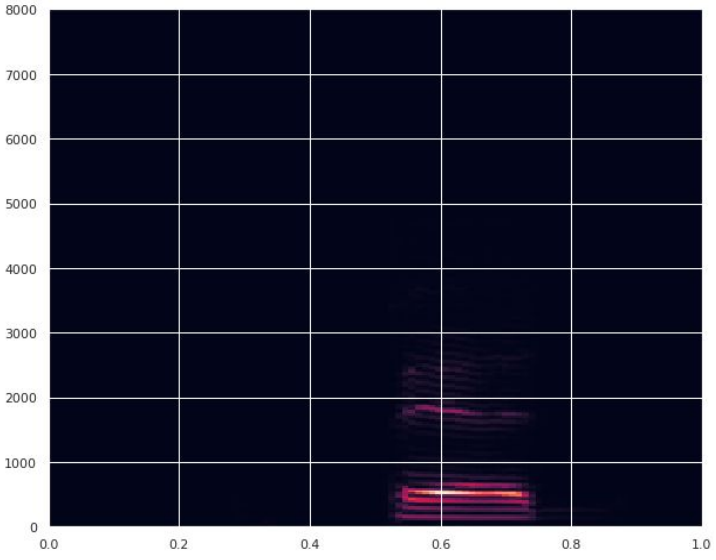
```
audio <class 'tensorflow.python.framework.ops.EagerTensor'>
audio (16000, 1)
audio [[ 9.1552734e-05]
 [ 3.0517578e-05]
 [ 1.8310547e-04]
 [ 1.8310547e-04]
 [-3.0517578e-05]
 [ 3.0517578e-05]
 [ 2.4414062e-04]
 [ 9.1552734e-05]
 [-9.1552734e-05]
 [-6.1035156e-05]]
```

Most of us are familiar with the representation of an audio signal, nevertheless here is how it looks like:



Spectograms

As stated earlier, a spectogram is a visual representation of the energy contained in a range of frequencies of audio signals over time. The following image shows an example of one spectogram generated from an audio of the word “bed”, the “y” axis represents the frequencies, which range from 0 to 8000 Hz, and the “x” axis is time (1 second), the brighter the point, the higher is the energy on that frequency at that specific moment.

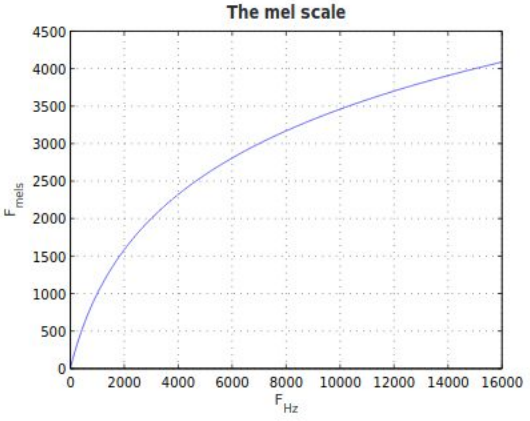


To obtain the spectogram we used the function “`tf.signal.stft`” from the signal processing methods included in TensorFlow. More information can be found at https://www.tensorflow.org/api_docs/python/tf/signal/stft

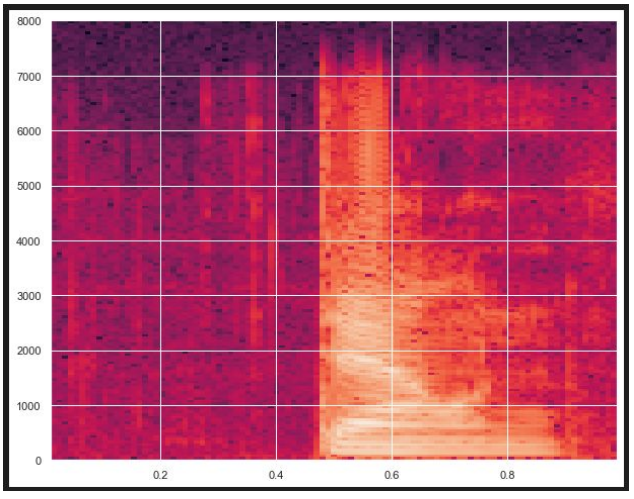
```python
# tf.signal.stft()
# https://www.tensorflow.org/api_docs/python/tf/signal/stft

def audio_to_spectogram(audio):
    """Returns the spectrogram of an audio signal"""
    stfts = tf.signal.stft(audio,
                           frame_length=480,
                           frame_step=160,
                           fft_length=None,
                           window_fn=tf.signal.hann_window,
                           pad_end=False,
                           name=None)
    spectrogram = tf.abs(stfts)
    return spectrogram
```

Mel-Spectrogram

Mel Spectogram is an “upgrade” of the standard spectogram, in which a log transformation is applied to the frequency scale. This new scale, called the Mel Scale, is build in such a way that frequencies are ranged similar to the way we perceive sounds. We humans do not interpret pitch linearly, as frequency increase our ability to differentiate them drop.



The next image shows the previous spectogram, but applying the Mel scale, clearly we obtain a better representation of the audio signal.



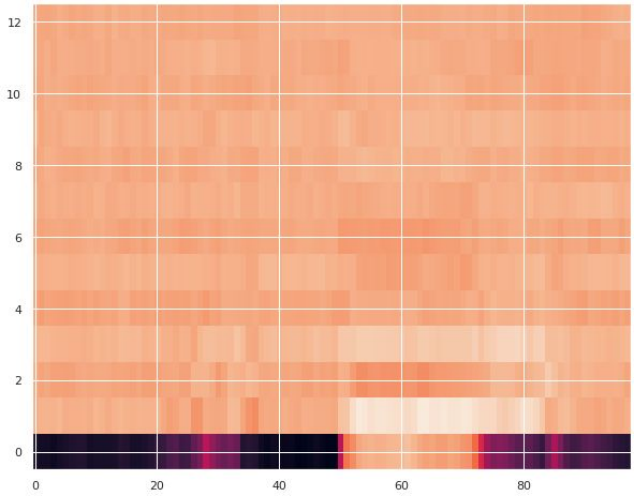
Here we use the same function as before “stft”, but applying the Mel transformation “`tf.signal.linear_to_mel_weight_matrix`”. More information can be found at: https://www.tensorflow.org/api_docs/python/tf/signal/linear_to_mel_weight_matrix

```
def audio_to_melspectogram(audio):
    """Returns the Mel-spectrogram of an audio signal"""
    stfts = tf.signal.stft(audio,
                           frame_length=480,
                           frame_step=160,
                           fft_length=None,
                           window_fn=tf.signal.hann_window,
                           pad_end=False,
                           name=None)
    spectrograms = tf.abs(stfts)
    # Warp the linear scale spectrograms into the mel-scale.
    num_spectrogram_bins = stfts.shape[-1]
    #print(num_spectrogram_bins)
    lower_edge_hertz, upper_edge_hertz, num_mel_bins = 80.0, 7600.0, 80
    linear_to_mel_weight_matrix = tf.signal.linear_to_mel_weight_matrix(num_mel_bins,
        num_spectrogram_bins, sample_rate, lower_edge_hertz,upper_edge_hertz)
    mel_spectrograms = tf.tensordot(spectrograms, linear_to_mel_weight_matrix, 1)
    mel_spectrograms.set_shape(spectrograms.shape[:-1].concatenate(linear_to_mel_weight_matrix.shape[-1:]))

    # Compute a stabilized log to get log-magnitude mel-scale spectrograms.
    log_mel_spectrograms = tf.math.log(mel_spectrograms + 1e-6)
    return log_mel_spectrograms
```

MFCC

Mel Frequency Cepstral Coefficients go further from the Mel spectogram and as described by Nair [6] are “*features that represent phonemes (distinct units of sound) as the shape of the vocal tract (which is responsible for sound generation) is manifest in them*”.

We use the function `tf.signal.mfccs_from_log_mel_spectrograms` to calculate them, usually, the first 13 coefficients are taken, here is how they look like:



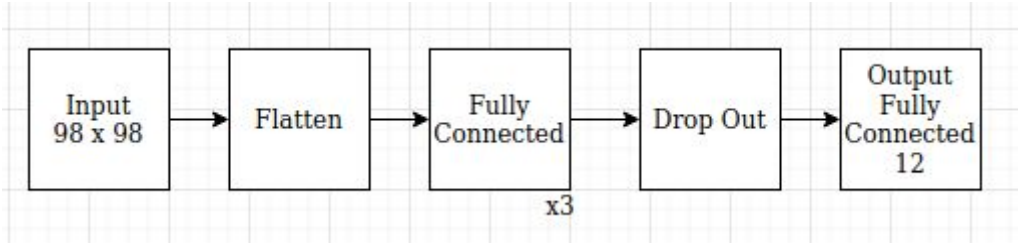
CNN

Convolutional Neural Networks are a special neural network architecture, used mainly to extract and analyze features on images, where filters are applied to the

input as a mathematical operation called convolution. This concept will be the base we will use to build our custom models.
We built three different neural networks with distinct structures, so we could analyze and test the performance of each architecture. These models can be found in the module “models.py”

A simple fully connected NN

This is a simple fully connected 3 layers neural network that we use just to check all the input pipeline and test our functions. As shown in the following diagram, this network is a simple fully connected network were the input with shape *m* x *n* is flattened, after we add multiple layers of fully connected nodes and at the end we add a drop out layer followed by the final dense layer with 12 nodes using *softmax* as activation function.



To implement our neural networks we use the abstraction layers found in the keras API:

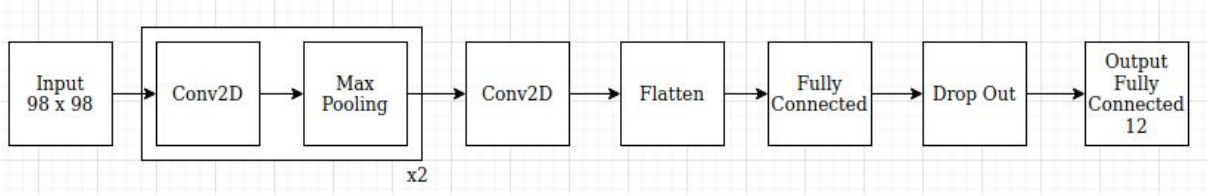
```
# import the necessary packages
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
```

Here is the implementation of the described network, wrapped on the function “`simple_stacked_fc_nn`”

```
def simple_stacked_fc_nn(width, heigth, depth, units_x_layer, classes, dropout_rate=0.2, activation_fc='relu'):

    model = Sequential()
    input_shape = (width, heigth, 1)

    #flat input
    model.add(Flatten(input_shape=input_shape))

    #core layers
    for _ in range(depth):
        model.add(Dense(units_x_layer, activation_fc))

    #Output
    model.add(Dropout(dropout_rate))
    model.add(Dense(classes, 'softmax')) # the shape of the output is the desired # of classes
    return model
```

A simple CNN

This is a simple convolutional neural network and our first approach to the solution. As shown in the following diagram, this network starts with two pairs of layers applying a 2D convolution followed by a Max Pooling layer, after this other 2D convolution is added, the output is flattened, and we add a layer of fully connected nodes, at the end we add a drop out layer followed by the final dense layer with 12 nodes using *softmax* as activation function.

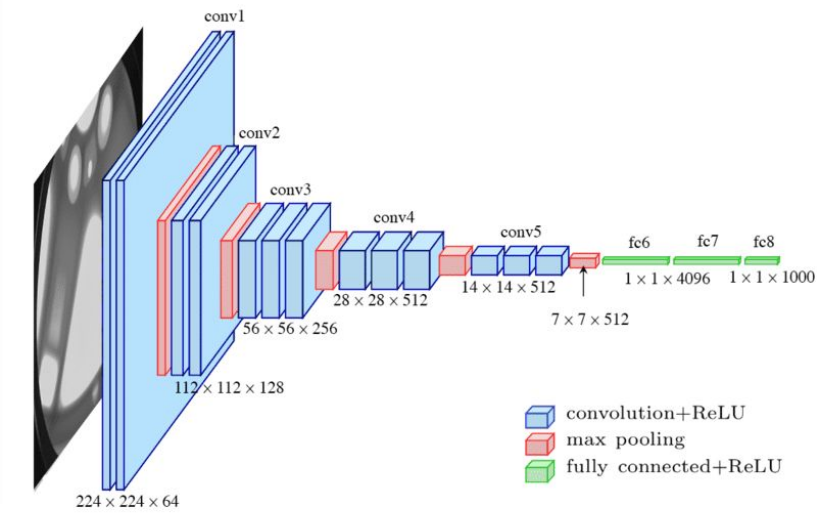


Here is the implementation of the described network, wrapped in the function "`simple_cnn`"

```
def simple_cnn(width, heigth, classes):
    model = Sequential()

    model.add(Conv2D(width, (3, 3), activation='relu', input_shape=(width, heigth, 1)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(width*2, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(width*2, (3, 3), activation='relu'))
    model.add(Flatten())
    model.add(Dense(width*2, activation='relu'))
    model.add(Dropout(0.35))
    model.add(Dense(classes, activation='softmax'))
    return model
```

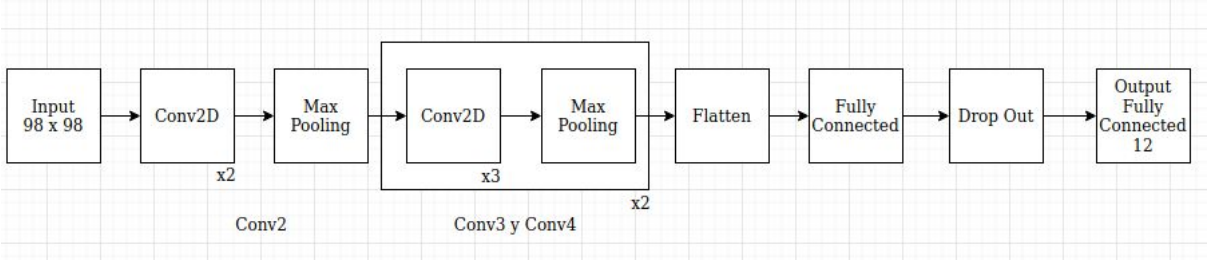
A custom VGG16

VGG16 is a convolutional neural network architecture for image classification proposed by K. Simonyan and A. Zisserman on the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition" [7], this architecture reaches 92.7% in the ImageNet dataset. The following image illustrates the structure of the network:



Max Ferguson, Researchgate

Basically, our custom model starts with an input with shape 98x98 and not 224x224, also, the first two convolutional layers “conv1” and the three from “conv5” were excluded, and the input change from 1000 to 10 that are the number of classes of our problem. The final diagram ended as follows:



Here is the implementation of the described network, wrapped in the function “`custom_vgg`”

```
def custom_vgg():
    model = Sequential()

    model.add(Conv2D(64, (3, 3), activation='relu', input_shape=(98, 98, 1)))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2, 2)))

    model.add(Conv2D(128, (3, 3), activation='relu'))
    model.add(Conv2D(128, (3, 3), activation='relu'))
    model.add(Conv2D(128, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2, 2)))

    model.add(Conv2D(256, (3, 3), activation='relu'))
    model.add(Conv2D(256, (3, 3), activation='relu'))
    model.add(Conv2D(256, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2, 2)))

    model.add(Flatten())
    model.add(Dense(512, activation='relu'))
    model.add(Dropout(0.4))
    model.add(Dense(12, activation='softmax'))
    return model
```

Benchmark model

As a reference, the accuracy of the top ranked 200 models on the public leaderboard of the Kaggle competition range from 0.87 to 0.91, with the first place having a score of 0.91060. It should be said that the testing dataset for the kaggle's competition is different from the test dataset provided for Google and the performance of the model once predictions are summited tends to be lower.

On the other hand, models implemented in the Google's TensorFlow tutorial [2], which include a low latency CNN, could reach an accuracy of between 85% and 90%, this could be the best reference for my custom model.

Methodology

Data Preprocessing

The input data and preprocessing process implemented is described in detail on the notebook 2_Preprocessing.ipynb. It follows the following pipeline, mostly inspired in the implementation done in the Google's solution, simplifying the scripts and making several editions to migrate the code to work with TensorFlow v2:

1. Single audio files are read and split into train, test and validation sets, using the hashing function provide by Google. Unknown class files are selected from the no core words to have a similar percentage than the other classes. Silence class is populated with files with zero values
2. Data augmentation is implemented, basically these transformations are applied:
    - Background noise is taken from the background files and added in a random percentage of the audios, volume of the noise could be also randomly adjusted.
    - Audio signal is time shifted, this could help the model to handle the fact that words information can be heard right at the beginning or at the end of the 1-second file
3. Preprocessed audio files are packed and saved into TensorFlow TFRecord files. I decided to implement this to improve the performance of the pipeline, as in one TFRecord file we can pack a big number of single audio files, as well handling samples from ' TFRecord Datasets' is more intuitive.
4. From this point TFRecords can be loaded into a 'Dataset' to feed the model. When training dataset is shuffled and random samples are sent to the trainer

Implementation

The implementation process, as described in the notebook *4_Model_Training.ipynb*, follows the next iterative process:

1. Load the training dataset from the TFRecords files, applying the function to obtain the desired feature.
2. Creating the models.
3. Define training parameters, optimizer, loss function and metrics.
4. Models are trained.
5. Performance is evaluated.

Loading the datasets / Getting the features

A function called `get_datasets` was implemented to obtain the datasets for the training, validation and testing process, this function takes as input the size of the batches and a transformation function to be applied to the audio files. The transformation function can be any of the three function we built to extract the features from the audio: `audio_to_spectogram`, `audio_to_melspectogram` or `audio_to_mfcc`.

```
def get_datasets(batch_size, transformation_fc):
    """get train, validate and test datasets and perform the parsed transformation function"""
    datasets = {}
    splits = ('train', 'validate', 'test')
    for split in splits:
        ds = get_dataset_from_tfrecords(batch_size=batch_size, split=split)
        ds = ds.map(transformation_fc)    #Transform audio to spectogram
        datasets[split] = ds
    return datasets['train'], datasets['validate'], datasets['test']
```

Here we have an example of the function in action, the transformation function applied was `audio_to_spectogram`, so as output we have a batch of 64 samples of a 124x124 image representing the audio's spectrogram (64, 124, 124, 1), plus their respective label (64, 1)

```
# get datasets
train_ds, validation_ds, test_ds = get_datasets(batch_size=64,
                                                transformation_fc=audio_to_spectogram)
print(train_ds,validation_ds, test_ds, sep='\n')
```

```
<MapDataset shapes: ((64, 124, 124, 1), (64, 1)), types: (tf.float32, tf.int64)>
<MapDataset shapes: ((64, 124, 124, 1), (64, 1)), types: (tf.float32, tf.int64)>
<MapDataset shapes: ((64, 124, 124, 1), (64, 1)), types: (tf.float32, tf.int64)>
```

Creating the models

stacked_fc_model: This model is created using the builder function `simple_stacked_fc_nn` described on the Algorithms and Techniques section. The image below shows the final structure of the network

```
#define model parameters
stacked_fc_model = simple_stacked_fc_nn(width=98,
                                        heigth=98,
                                        depth=3,
                                        units_x_layer=512,
                                        classes=12)
print(stacked_fc_model.summary())
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 9604)              0
_________________________________________________________________
dense (Dense)                (None, 512)               4917760
_________________________________________________________________
dense_1 (Dense)              (None, 512)               262656
_________________________________________________________________
dense_2 (Dense)              (None, 512)               262656
_________________________________________________________________
dropout (Dropout)            (None, 512)               0
_________________________________________________________________
dense_3 (Dense)              (None, 12)                6156
=================================================================
Total params: 5,449,228
Trainable params: 5,449,228
Non-trainable params: 0
_________________________________________________________________
None
```

cnn_3l_model: This model is created using the builder function `simple_fcnn` described on the Algorithms and Techniques section. The image below shows the final structure of the network

```
#Creates a 3 conv layers model, input 98x98, 12 output nodes
cnn_3l_model = simple_cnn(98, 98, 12)
print(cnn_3l_model.summary())
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 96, 96, 98)        980
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 48, 48, 98)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 46, 46, 196)       173068
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 23, 23, 196)       0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 21, 21, 196)       345940
_________________________________________________________________
flatten_3 (Flatten)          (None, 86436)             0
_________________________________________________________________
dense_8 (Dense)              (None, 196)               16941652
_________________________________________________________________
dropout_3 (Dropout)          (None, 196)               0
_________________________________________________________________
dense_9 (Dense)              (None, 12)                2364
=================================================================
Total params: 17,464,004
Trainable params: 17,464,004
Non-trainable params: 0
_________________________________________________________________
None
```

custom_vgg_model: This model is created using the builder function `custom_vgg` described on the Algorithms and Techniques section. The image below shows the final structure of the network

```
custom_vgg_model = custom_vgg()
print(custom_vgg_model.summary())
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_9 (Conv2D)            (None, 96, 96, 64)        640
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 94, 94, 64)        36928
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 47, 47, 64)        0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 45, 45, 128)       73856
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 43, 43, 128)       147584
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 41, 41, 128)       147584
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 20, 20, 128)       0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 18, 18, 256)       295168
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 16, 16, 256)       590080
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 14, 14, 256)       590080
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 7, 7, 256)         0
_________________________________________________________________
flatten_4 (Flatten)          (None, 12544)             0
_________________________________________________________________
dense_10 (Dense)             (None, 512)               6423040
_________________________________________________________________
dropout_4 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_11 (Dense)             (None, 12)                6156
=================================================================
Total params: 8,311,116
Trainable params: 8,311,116
Non-trainable params: 0
_________________________________________________________________
None
```

Defining the training parameters

The following training parameters were defined for all models:

- Optimizer: Adam optimization algorithm.
- Loss function: sparce categorical cross entropy
- Metrics: Multiclass accuracy

Here is the way we assign the parameters to the models:

```
stacked_fc_model.compile(optimizer='adam',
                         loss='sparse_categorical_crossentropy',
                         metrics=['accuracy'])
```

Training the models

Once defined the models and assigned the training parameters, the model's method fit is used to start the training process, here we pass the number of epochs, the training and the validation datasets.

```
history = model.fit(train_ds, epochs=5, validation_data=validation_ds)


Epoch 1/5
1008/1008 [==============================] - 287s 285ms/step - loss: 0.0520 - accuracy: 0.9858 - val_loss: 0.9217 - val_accuracy: 0.8689
Epoch 2/5
1008/1008 [==============================] - 286s 283ms/step - loss: 0.0500 - accuracy: 0.9859 - val_loss: 0.8287 - val_accuracy: 0.8764
Epoch 3/5
1008/1008 [==============================] - 284s 281ms/step - loss: 0.0445 - accuracy: 0.9871 - val_loss: 0.8407 - val_accuracy: 0.8747
Epoch 4/5
1008/1008 [==============================] - 283s 280ms/step - loss: 0.0399 - accuracy: 0.9884 - val_loss: 0.8613 - val_accuracy: 0.8709
Epoch 5/5
1008/1008 [==============================] - 281s 279ms/step - loss: 0.0439 - accuracy: 0.9876 - val_loss: 0.8105 - val_accuracy: 0.8733
```

Spectrograms vs Mel-Spectrograms vs MFCC

The next tables show a comparative analysis between the three features extracted from the audio files and the three proposed models, it presents their performance on the training, validation and testing sets. All models were trained using the same parameters:

- Optimizer: Adam optimization algorithm.
- Loss function: sparce categorical cross entropy
- Metrics: Multiclass accuracy
- Epochs:4
- batch size: 64

Accuracy on training set

Model / Input	Spectrograms	Mel-Spectrograms	MFCC
stacked_fc_model	0.8777	0.6688	0.6758
cnn_3l_model	0.9697	0.9411	0.9274
custom_vgg_model	0.9466	0.9499	0.9454

Accuracy on validation set

Model / Input	Spectrograms	Mel-Spectrograms	MFCC
stacked_fc_model	0.6967	0.6270	0.6332
cnn_3l_model	0.8366	0.8573	0.8468
custom_vgg_model	0.8723	0.8781	0.8587

Accuracy on testing set

Model / Input	Spectrograms	Mel-Spectrograms	MFCC
stacked_fc_model	0.6817	0.6226	0.6338
cnn_3l_model	0.8389	0.8447	0.8434
custom_vgg_model	0.8739	0.8824	0.8668

Some thought about the results:

- The custom VGG model outperform the other models on the training and validation sets.
- Accuracy using the Mel-spectrograms tend to be slightly better than those using just the spectogram.
- Models using just the spectrogram tend to overfit more, as its accuracy on the training and validation shows bigger differences.
- Model's performance using MFCC is lower in most of the cases.

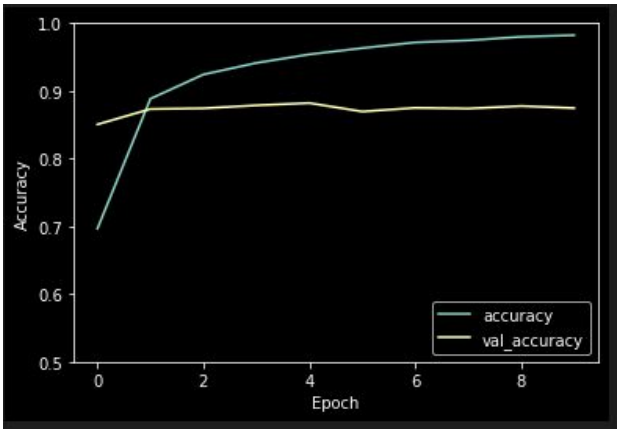
We will choose the custom VGG model not only because its performance but also because it has almost half of the parameters of the 3 layers CNN model.

Refinement

After choosing a model and the feature that gave us the best performance, we tested some changes in several parameters with the aim of further improve the model's performance.

Number of epochs

During the initial evaluation of the models we executed the training using 4 epochs, after increasing this number to 10, as we can see in the graphic below, the model's performance did not improve, indeed accuracy on the training set got stuck around 0.87, also it is evident that after the 5th epoch the model start overfiting.



Batch size

Three batches sizes were tested 64, 128 and 254. No significant differences were found on the final model's performance, neither in the difference on accuracy between the training and validation sets.

Drop out rate

In the custom model, right after the last convolution layer, a Dropout layer was placed, it helped the model to avoid overfiting. Several values were tested for the dropping rate on this layer, from 0.2 to 0.6 and at the end the value of 0.5 seams to be the best point, as bigger values affect the models' accuracy.

It seems that tuning the training parameters did not have a great impact on the final model performance. It will be interesting to explore other changes such as, further improvement on the model's architecture or include in the data augmentation process a method to normalize the volume of the audio files.

Results

Model Evaluation and Validation

The model with the best performance over all was the "custom_vgg_model", it reached a score of 0.88. Using the Mel-spectrograms gave us the best performance and after testing several hyperparameters as discussed in the previous section, these were the final used:

- Optimizer: Adam optimization algorithm.
- Loss function: sparce categorical cross entropy
- Metrics: Multiclass accuracy
- Epochs:4
- batch size: 64
- Dropout rate: 0.5

```
# Accuracy on testing set
test_loss, test_acc = custom_vgg_model.evaluate(test_ds, verbose=2)
print(f'The accuracy on the testing set is {test_acc}')

The accuracy on the testing set is 0.8865489363670349
```

Justification

Custom VGG Model vs Google Tutorial Examples

The custom VGG model reached almost the top score claimed for the Google's tutorial models which was 0.90. The final model's score is also some points above our initial goal that was 0.85, is far to say that our main goal was fulfilled.

It should be stated that the data augmentation functions applied to the audio files such as, time shifting and background mixing described in the notebook *2_Preprocessing.ipynb,* gave to the model a real boost, going from a maximum of 0.84 (using the same model but without data augmentation) to 0.88, this improvement is a sign of the ability of the model to generalize better.

As stated earlier it will be interesting to explore other options to make the model more robust, for example, further improvement on the model's architecture or include in the data augmentation process a method to normalize the volume of the audio files.

References

[1]. Warden P. (2017, August 24). Launching the Speech Commands Dataset. Retrieved from: https://ai.googleblog.com/2017/08/launching-speech-commands-dataset.html

[2] TensorFlow tutorials, Simple Audio Recognition. 2017. https://github.com/tensorflow/docs/blob/master/site/en/r1/tutorials/sequences/audio_recognition.md#simple-audio-recognition

[3] TensorFlow Speech Recognition Challenge. 2017. https://www.kaggle.com/c/tensorflow-speech-recognition-challenge/overview

[4] TensorFlow Speech Recognition Challenge, Evaluation. 2017. https://www.kaggle.com/c/tensorflow-speech-recognition-challenge/overview/evaluation

[5] Sainath T, Parada C. Convolutional Neural Networks for Small-footprint Keyword Spotting. INTERSPEECH 2015.

[6] Nair Pratheeksha, The dummy's guide to MFCC, 2018. https://medium.com/prathena/the-dummys-guide-to-mfcc-aceab2450fd

[7] Simonyan K, Zisserman A, Very Deep Convolutional Networks for Large-Scale Image Recognition, ICLR 2015.