Deep Learning project report

Motivation:

The goal of this project is to implement a compact deep learning model/s to classify a set of 8-10 keywords accurately such that the model can be reliably deployed on a resource constrained embedded device. The model can then perform keyword inference locally such that one or a combination of those keywords can be used to perform a set of smart domestic tasks, i.e. - switching of lights, changing temperature, etc. Consequently it will be unnecessary to perform inference for the set of local commands on the server side, saving bandwidth and cloud resources for tasks which are non-local (requiring internet connectivity) and bypassing network reliability issues for such local commands.

Methodology:

The speech recognition network is implemented using Tensorflow's speech recognition example at the following repository:
https://github.com/tensorflow/tensorflow/tree/r1.15/tensorflow/examples/speech_commands

The speech model is trained on words taken from Google's speech commands dataset(https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781789132212/5/ch05lvl1sec42/google-speech-commands-dataset), which consists of 65,000 one-second long audio clips where each clip contains one of 30 different words spoken by different subjects in realistic environments. Each word has a folder containing all its one-second long wave file examples. 8-10 words from the dataset are used to train the speech model, whereas examples from words not selected are used to train an 'unknown' word label. A 'silence' label is also trained to detect no spoken word. It should be noted that words falling outside these 30 words can also be trained using this script as long as it contains sufficient example clips and the clips are formatted and organized like the existing words.

Each second of raw audio captured from the device's microphone is converted to a 2D spectrogram of features using an FFT algorithm and fed as input to the model. By deafult, each spectrogram consists 40 rows and 49 columns, were each row is a 30-ms audio slice split into 43 frequency buckets using FFT. The 2D array for one second of audio data is built by running FFT on 49 consecutive audio slices with each slice overlapping the last by 10-ms. Both the audio slice size and slice stride are tunable parameters, and hence the spectrogram input shape can be changed.

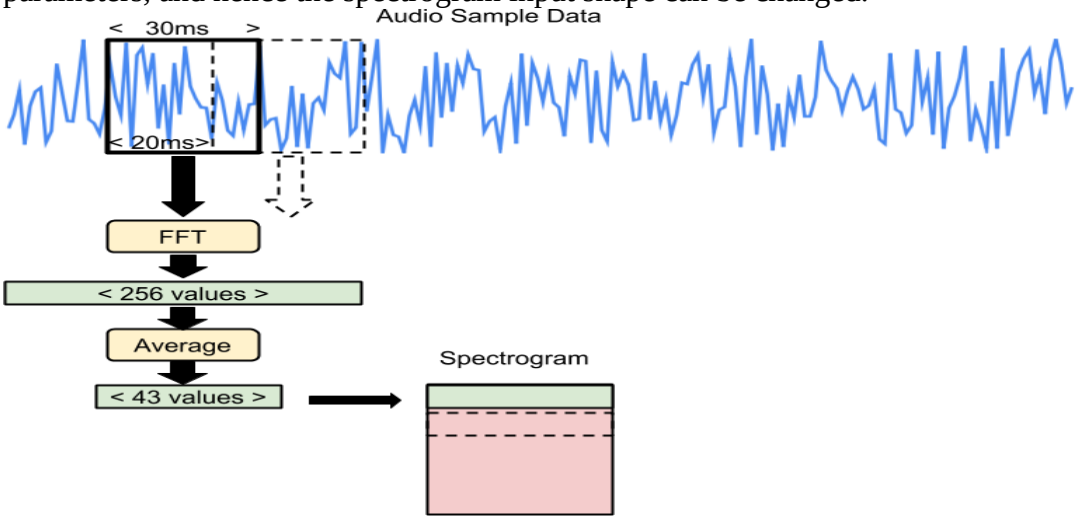

Figure 7-4 from **[1]**. *Diagram of audio samples being processed*

The model then performs inference on the one-second spectrogram data and outputs logits for each keyword it classifies. Since audio data is continously streaming and multiple such inferences are performed per second on overlapping streaming data, a command recognizer class,recognize_commands.h, is used to recognize the spoken command by averaging the score for all words over an averaging window which contains a minimum number of inferences (3). If the average score for a spoken word is above a threshold value, it is detected as the spoken word. The averaging window size and detection threshold are both tunable parameters.

Speech recognition model:

The model architectures used to train the keyword classifier are defined in the file 'models.py' . I added my convolutional model 'cnn' defined by the 'create_cnn_model' function in the 'models.py' file with the aim of optimizing model tradeoff between size and performance for small footprint deployment on embedded platforms. The model outline is given below:

(fingerprint_input)
v
[Conv2D layer]
v
[Relu]
v
[Batch norm layer]
v
[Conv2D layer]
v
[Relu]
v
[Batch norm layer]
v
[Maxpool layer]
v
[Output/logits layer]
v

The 'cnn' model consists of two successive convolutional layers with 64 and 48 kernels respectively. The larger number of kernels and single strides in the first convolutional layer are meant to extract maximum information from the input while the smaller number of filters and larger stride in the 2nd convolutional layer is supposed to capture more discriminative features. I use strides of 2 in one dimension for the second convolutional layer kernels as an alternative to a pooling layer. This is because normally a pooling layer would downsize the output of the first convolutional layer to a greater degree and therefore would not retain as much information. The second convolutional layer is followed by a maxpooling layer which connects to the output layer to obtain the logits for each word. The maxpooling layer is meant to enable further feature discrimination as well as downsize the second convolutional layer so as the reduce the number of parameters connecting to the output layer. Batch normalization layers are introduced after each convolutional layer to optimize/stabilize training

As I require a model with a small footprint, I did not introduce more layers to the model so as to make it compact yet reliable. Other than adjusting model parameters such as configurations of each layer, number of training steps,learning rate and dropout rate; training input parameters such as the amount and volume of background noise mixed in, time shifting to add distortion, number of frequency bins in each spectrogram window slice as well the size and stride of each spectrogram window were also tweaked to train an optimal model.

I also added and tested a recurrent network model ‘rnn’ defined in ‘create_rnn_model’ to evaluate it’s performance. It consists of 3 lstm layers with 118 units in each layer. I compared both my models against the baseline ‘conv’ model provided as the default model for training the keyword spotter in the ‘create_conv_model’ function.

Experimental setup:

All models are trained to classify 8 keywords in consideration of feasible implementation in terms of compute intensiveness and size. The window size for the audio input is set at 30-ms with a window stride of 20-ms to create the spectrogram for 1 second of audio. 40 frequency/feature bins are used to represent each spectrogram window. 10% of the training data is used to train the ‘unknown’ word label and 10% to train the ‘silence’ label. 10% of the wave files are used as the test set and 10% as the validation set .

Background noise (volume 0.1 with max. being 1) is introduced to 80% of the training samples. The training audio is randomly shifted in time within a range of 100-ms. The first 23000 training steps have a learning rate of 0.001 and the following 7000 steps have a smaller learning rate of 0.0001 for fine-tuning. Training configuration parameters are kept constant for all models and are specified in the ‘train.py’ file and the corresponding training commands in the README file. The experimental results are evaluated in the next section.

Results:

The results of training and evaluating my ‘cnn’ model are presented below. The results obtained from training both the ‘cnn’ model and the baseline ‘conv’ model are presented in the ‘model_results’ file in this directory.

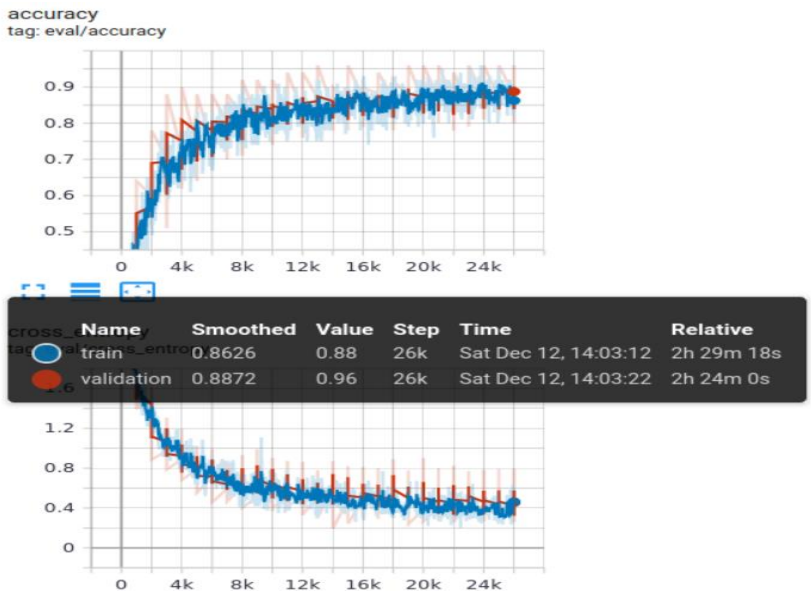

Fig: Accuracy and cross entropy graphs from training 'cnn' model obtained from tensorboard logs. As can be seen, at training step=26k validation accuracy is about 96%.

Number of parameters:

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 40, 37, 64)	2624
batch_normalization_1 (Batch	(None, 40, 37, 64)	256
dropout_1 (Dropout)	(None, 40, 37, 64)	0
conv2d_2 (Conv2D)	(None, 16, 34, 48)	122928
batch_normalization_2 (Batch	(None, 16, 34, 48)	192
dropout_2 (Dropout)	(None, 16, 34, 48)	0
max_pooling2d_1 (MaxPooling2	(None, 15, 33, 48)	0
flatten_1 (Flatten)	(None, 23760)	0
dropout_3 (Dropout)	(None, 23760)	0
dense_1 (Dense)	(None, 10)	237610

Total params: 363,610
Trainable params: 363,386
Non-trainable params: 224

Confusion matrix:

```
[[321   2   1   0   1   0   0   0   0   2]
 [  0 211   9  29  12   8  16  23  14   5]
 [  2  10 384   8   0   3   8   2   1   1]
 [  0  10   3 361   3  16   5   2   3   2]
 [  1  15   0   5 369   5   4   2   8  16]
 [  1  16   5  42   4 330   4   1   2   1]
 [  2   8  21   3  10   3 359   6   0   0]
 [  1  16   1   0   0   3  14 359   2   0]
 [  3  19   2   2   6   5   0   1 355   3]
 [  1   6   0   6  19   6   0   1   5 367]]
```

Final test accuracy = 87.1% (Number of test samples=3924)

I compare these results to the baseline 'conv' model provided for good keyword spotting performance. The 'conv' model uses 940,000 weight parameters and requires over 800 million FLOPs to perform each inference. While this should enable the 'conv' model to provide good performance, it is too compute intensive to run constantly on resource constrained embedded devices and therefore not feasible as a keyword spotting model. In contrast, my model uses about 363000 weight parameters and performs 14 million FLOPs per inference. Assuming a system running a maximum of 10 inferences per second, an embedded MCU with reasonable compute resources can feasibly run a model performing upto about 20Mops/inference (reference in Table 3 in **[2]**). Thus, my model will be able to run in interactive speeds on limited-resource devices with reasonable compute performance.

The model can be quantized for eight-bit deployment by running the training script with the 'quantize' flag set to 'True'. The size of my model without quantization is about 1.4MB whereas it is about 370KB with quantization. While quantization led to little degradation in test accuracy, the quantized model is small enough to fit within the constraints of embedded platforms. For instance, a number of Cortex M4/M7 boards have 1-2 MB flash (Table 1 in **[2]**). However additional SRAM memory may be required than is present in such some of these example boards to run this model.

Despite the small footprint of my model, its test accuracy is similar to that of the baseline model accuracy of 87.2%. Thus, my model is highly optimized for efficient keyword spotting performance on embedded systems. This result also shows there is scope for improvement in terms of model performance-size tradeoff.

I did not include the results of my 'rnn' model in this evaluation as it's performance is inferior to that of my 'cnn' model (it has about 81% test accuracy). However, the performance results for the 'rnn' model can be obtained by following the corresponding instructions in the README file. The inferior performance of the 'rnn' model may be because recurrent networks are more suitable for capturing longer time-series dependencies than is the case for a 1-second long spectrogram time-series input.

Streaming accuracy:

As a keyword spotting application run on a continuous stream of audio rather than individual clips, I evaluate model perfromance on streaming data. In this case, it is is done by applying the model repeatedly at different offsets and averaging the results over a short window to produce a smoothed

prediction. I evaluate my model on streaming data by running the ‘test_streaming_accuracy’ file. This uses the ‘Recognize commands’ class mentioned before to run through a long-form input audio, try to spot words, and compare those predictions against a ground truth list of labels and times.

I need a long audio file to test streaming data along with labels showing where each word was spoken. I generate synthetic test data by running the ‘generate_streaming_test_wav,py’ file to create a 5 minute .wav file with similar confiurations I used for training and with words roughly every two seconds, and a text file containing the ground truth of when each word was spoken.

Then I run the test streaming accuracy script (test_streaming_accuracy.py) with my model on the generated audio file to evaluate streaming performance. I use an averaging window of 500-ms to smooth prediction results (with minimum 3 inferences within that window). The detection threshold for a prediction within that window is set at 0.7 (max. is 1). The ‘time-tolerance-ms’ flag is 1500-ms, which is the maximum time given to the model to recognize a word since it is spoken. The ‘suppression_ms’ flag is set at 500-ms, and prevents subsequent word detections for the specified duration after an initial one is found. Each of these parameters are configurable.

The streaming accuracy results of my model is as follows:

67% matched: 63% correct, 4% wrong, 0% false positive

Among the 67% of words my model recognized (about 30% of words belong to the ‘unknown’ label) in the generated audio file, 63% of words were correctly classified, 4% were wrong and there were no false positives. The baseline ‘conv’ model had similar performance. The incorrect classifications may be caused by discrepancies arising from simulating streaming performance on synthetic data. However, the model has fairly good streaming performance on the synthetic audio file and this a good demonstration of applying the model on streaming audio data and obtaining reasonable performance by using appropriate/optimized parameters for the streaming function.

Conclusion:

The results obtained for my convolutional model show it can be feasible to train an optimized keyword spotting model to perform on-device inference for local commands. Although there needs further improvement in model performance, the fact that my model achieved similar classification performance as the baseline model despite its compactness indicates there is scope to further improve keyword spotting performance while maintaining reasonable model resource requirements. The keyword spotting model can be applied on streaming data to detect commands based on a single word or a combination of consecutive words. The fact that the convolutional model performs better than the recurrent model in this case may be because convolutional layers efficiently model the local temporal and spectral correlation of speech features present in its 2D spectrogram image. I am further exploring the model architecture space to leverage advantages provided by different types of network layersin order to improve on the ‘cnn’ model. For example, I am exploring the implementation of a combined convolutional and recurrent model to potentially exploit the characteristics of both types of networks and achieve better results.

References:

[1] TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers 1st Edition by Pete Warden and Daniel Situnayake

[2] Hello Edge: Keyword Spotting on Microcontrollers, Yundong Zhang, Naveen Suda, Liangzhen Lai and Vikas Chandra