Deep Learning: Project 2 Report

Maja Andrzejczuk, Maciej Orsłowski

Contents

1 Research Problem

The research centers on investigating speech command classification using three distinct models: ResNet-18, a custom Convolutional Recurrent Neural Network (CRNN), and the Vision Transformer (ViT) architecture, specifically the ViT-Base-16 model. The primary objective is to compare these architectures to evaluate their effectiveness in recognizing spoken commands from the Speech Commands Dataset (inversion 2017). This dataset contains various audio samples representing different commands. Each subfolder contains one-second long audio clips labelled with specific commands, including 'yes', 'no', 'up', 'down' and more, along with additional samples for 'silence' and 'unknown' classes.

To prepare the data for processing, we converted the audio into mel-spectrograms, allowing us to visualize sound as images. This transformation provides a more informative representation of the audio, facilitating easier analysis and feature extraction in machine learning models.

In our search for the right parameters for our models, we used Bayesian optimization. We explored parameters like batch size, learning rate, L2 regularization, and beta1 and beta2, which are key parameters for the ADAM optimizer. This method allowed us to efficiently optimize the hyperparameters, leading to improved model performance.

2 Instruction of the application

Our application comprises two Python Notebooks (.ipynb format). It requires Python 3.10 and the following packages:

- librosa (0.10.1),
- torch (2.2.1),
- torchvision (0.17.1),
- matplotlib (3.8.4),
- seaborn (0.12.2),
- bayesian-optimization (1.4.3),
- pandas (1.5.0),
- numpy (1.26.4).

The two notebooks are named `data_preparation.ipynb` and `models.ipynb`, and they should be run in this order. The first notebook contains the code to generate datasets which are used to train the models. The details of data preprocessing are described in section 4. The application expects the input data to be placed inside the `./data/raw/` folder, where `.` denotes the directory in which the notebooks reside. The `raw` folder is expected to contain all the data files, such as the `audio` folder and test/validation lists.

Only once the data is generated, the second notebook, `models.ipynb`, can be run. It contains all the code required to train the models, as well as various plots which visualise training processes and evaluate the models. Unfortunately, this notebook can only be run on Windows, due to bayesian-optimization package requiring this OS to work.

Both data preparation and models training rely on functions which involve randomness. However, in all cases, seeds are passed to random number generators to ensure reproducibility of our results across the entire application. On top of that, the model training is configured to utilise CUDA if a hardware that it runs on supports it, which can significantly shorten training times. The application automatically checks whether PyTorch can use CUDA and sets the device accordingly.

Before running the `models.ipynb` notebook, one has to keep in mind that training times can be very long, even when using CUDA. Some of the training code can run for many hours, with the entire notebook potentially taking days to complete on an average hardware. To alleviate this, all the results of lengthy operations are saved once these operations are completed. This includes model training, parameter optimisation and final predictors' evaluation. These results can then be loaded, which means a user does not have to run everything in the notebook at once to replicate our experiments. The code which loads the data is usually provided in the notebook, however some of the save paths might be different for different users as they may depend on a timestamp when the operation was completed (this was put in place for training results, to eliminate the risk of overwriting data).

3 Theoretical Introduction

3.1 Mel-spectrogram - audio transformation

The input data was originally stored in .wav files, which store uncompressed audio signals. These signals are mono (single channel), each 1 second long with 22050 samples. While it is possible to train models directly on these files, it is a common approach to transform them into spectrograms. A spectrogram is an image which represents the audio signal. Thus, classifying audio by using spectrograms turns this task into image classification.

Spectrograms utilise various audio-transformation techniques, understanding which is key to understanding spectrograms. The most important one is the Fourier transformation, which converts the signal from time domain into a frequency domain. A spectrogram is generated by applying the Fourier transformation on a window sliding across the whole signal. This is controlled by two parameters: `n_fft`, which determined the window size (i.e. how many samples are used for a single Fourier transformation), and `hop_length`, which indicates the step size for a window. For example, with `n_fft=2048` and `hop_length=256`, the first Fourier transform would be calculated for samples starting at 0 and ending at 2048, and the second one would be from samples 256 to 2304, etc. Each Fourier transformation returns a spectrum of frequencies, and these spectrums are then concatenated to form the spectrogram, meaning the spectrogram is a matrix, where the x-axis represents time, and the y-axis is a frequency domain. Spectrograms are usually visualised using heatmaps (Roberts 2024).

The type of spectrograms we use in our experiments is called a mel-spectrogram. It differs from a regular spectrogram in the frequency domain scale. A regular spectrogram utilises a linear scale, which is not natural for human ears when it comes to perceiving differences between different frequencies. The mel-spectrogram utilises a mel-scale, which converts frequency values in such a way, that in a spectrum which is audible for humans, equally distant frequencies sound equally different to human ears. Considering that what we are working with is human speech, this approach could potentially improve the models' ability to find patterns in the data.

3.2 CNN

Convolutional neural network (CNN) is a type of neural network that is useful when dealing with images. We explained it in detail in the Project 1 report, however, since we also use CNNs in this project, we provide a shorter description of how these networks work.

A convolutional layer is characterised by a number of input channels, a number of output channels, a filter size, a stride and a padding size. Images going into the network in our case have 3 channels, one for each color (RGB). A filter (or a kernel, as it is named in PyTorch) is a matrix, which usually has an equal width to height, and a depth equal to the number of channels in the input tensor. Its width and height are parameters that can be tweaked when designing a network architecture. Convolutional layers process an image by convolving over all chunks of the image which are of the same size as the filter, and taking a dot product of these chunks and the filter. This can be denoted as simply:

$$w^T x + b$$

where w is the filter, x is a chunk, and b is a bias. This results in a single number, so if applied over all chunks of the image, will eventually output a 2D matrix. For a square filter, if we denote filter size by f, image height and width by h and w respectively, stride by s, and padding size by p, the output of the convolutional layer is a matrix with its height and width equal to (respectively):

$$h_{out} = \frac{h - f + 2p}{s} + 1$$

$$w_{out} = \frac{w - f + 2p}{s} + 1$$

This operation can be repeated multiple times with multiple different filters, resulting in more output channels of the layer, and can help discover different features in the image. The elements of the matrix can also have an activation function applied to them.

Another concept related to CNNs are pooling layers, which are often used to compress the outputs of convolutional layers. The most commonly used one is a max-pool layer, which for each chunk of the image returns a maximum of all number in a chunk. In our solution, we always set both the stride and the kernel size to 2, meaning such layers halve the output sizes of the preceding convolutional layers.

Deep neural networks can suffer from the problem called the vanishing gradient, which means that the gradient propagated backwards approaches zero. This results in weights of initial layers not being updated, which makes the training process ineffective. Alternatively, a phenomenon called exploding gradient can occur, which is the opposite of vanishing gradient. Here, the gradients get very large, which can either result in overflows, or make the training unstable and cause gradient descent to diverge. ResNets are a type of deep neural network which attempt to avoid these issues by skipping some of the layers while training (He et al. 2015). ResNet-18, which we use in our experiments, is a type of ResNet which is 18 layers deep.

3.3 RNN and CRNN

Recurrent Neural Network (RNN) is a type of network which considers the order within the input data. It does so by maintaining a hidden memory. A regular multi-layer perceptron (MLP) can process only one input at the time. It also has no memory of the previous input it has processed and relies solely on its training. In contrast, an RNN can process input data chunk-by-chunk, and for each chunk that follows, some memory from the previous chunks is applied. This makes it suitable for dealing with objects such as time series, text or sound, where the order in the input sequence is a crucial feature of the data. There are a few variants of RNN architectures: one-to-one (which is just like MLPs); one-to-many: one input produces multiple outputs; many-to-one: multiple inputs produce a single output; and many-to-many: multiple inputs produce multiple outputs (Donges 2024).

A classic RNN architecture can be summarised using the following formula:

$$h_t = \tanh\left(\begin{bmatrix} U & W \end{bmatrix} \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix}\right) \tag{1}$$

where h_t can be interpreted as a hidden memory at step t, U represents the weights of recurrent connections within the hidden state, W represents the weights of connections between the input layer and the hidden state and x_t is an input at step t. This architecture however has its flaws. Firstly, the memory is usually only maintained for a couple of steps (max 10). Secondly, this architecture often suffers from the problem of either vanishing or exploding gradient.

A popular variant of RNN is a Long Short-Term Memory (LSTM), introduced by Hochreiter & Schmidhuber (1997). It mitigates the problem of vanishing gradient and also introduces a long-term memory by introducing a state cell c_t. Contrary to the classic RNN architecture, LSTM works very well in practice. Its long-term memory can persist for even as long as 100 steps.

The LSTM architecture can be summarised using the following three formulae:

$$\begin{bmatrix} i \\ f \\ o \\ g \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} \left(\begin{bmatrix} U & W \end{bmatrix} \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix}\right) \tag{2}$$

$$c_t = f \cdot c_{t-1} + i \cdot g \tag{3}$$

$$h_t = o \cdot \tanh(c_t) \tag{4}$$

The formula 2 is similar to 1 however here it calculates values of four gates. The first three function as follows: the "forget" gate f indicates whether the previous cell state should be erased or not, the "input" gate i determines whether the state should be updated, and the "gate" gate g decides how much should be written to the cell. They are all used to update the state cell at each step (formula 3). The final gate, the "output" gate o, is used when updating the hidden state, and it determines how much of the cell state should be passed to the hidden state (formula 4).

Convolutional Recurrent Neural Network (CRNN) is a type of neural network which integrates the ideas of Convolutional Neural Networks (CNN) and RNNs into one architecture. It helps to bring all the benefits of using RNNs into image processing, by adding recurrent layers after a convolutional block. This approach can be useful for spectrograms, because one of their dimensions is a time dimension.

3.4 Transformers and VIT

The Transformer architecture, originally introduced in the article 'Attention Is All You Need' (Vaswani et al. 2017), consists of two main components: the encoder and the decoder. For image processing applications, this architecture must be adapted from its original design, which was intended for text data.

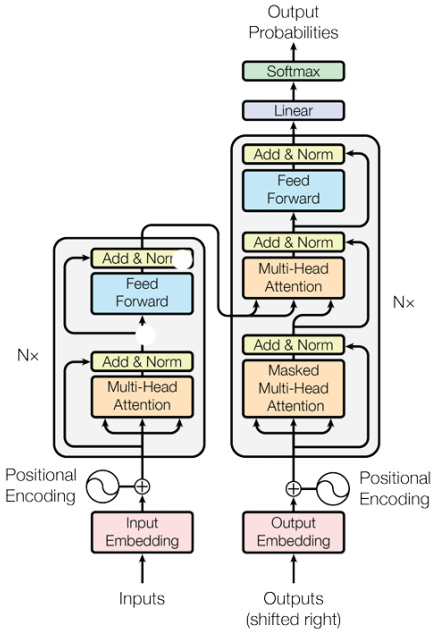

Figure 1: Transformer architecture. Image from the original paper (Vaswani et al. 2017).

A key adaptation involves the self-attention mechanism, which can effectively handle spatial relationships in data. The formulation of attention used in the Transformer can be generalized and is described by the following components and equation:

- Query (Q) - a vector that represents an item or a point in time for which we're trying to find relevant information
- Key (K) - the vector against which the query is compared to measure similarity or relevance.
- Value (V) - the vector that contains the information or content associated with a token.

The attention mechanism, $A(Q, K, V)$, is computed as:

$$A(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Further enhancing the Transformer's capability is the Multi-head Attention (MHA) mechanism, which allows the model to attend to different parts of the input sequence independently. Multi-head attention enables the model to perform better in complex scenarios by processing various features of the input data in parallel, thus making it a powerful component in the Transformer architecture.

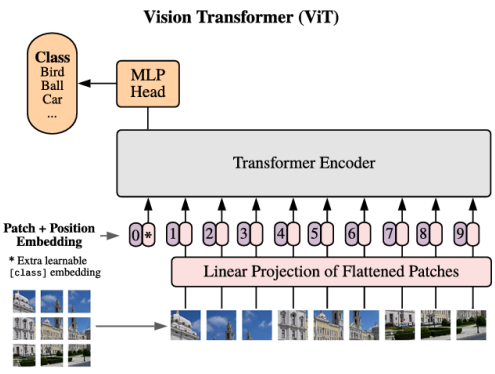

Figure 2: Vision Transformer architecture

To process images, the Transformer needs to convert 2D image data into a sequence of patches, treating them similarly to words in a text. This transformation involves breaking the image into smaller pieces, with each patch being processed as a single token. This approach was introduced and thoroughly described in Dosovitskiy et al. (2021).

Once this sequence is created, the encoder part of the Transformer processes and encodes the features of each patch, similar to how an encoder handles words in a text. This allows for the extraction of meaningful information, building a representation that captures both local and global features of the image.

4 Data preprocessing

The dataset employed for this project originates from the TensorFlow Speech Recognition Challenge (inversion 2017). It consists of raw audio recordings of spoken words categorized into various classes. For processing the audio data, we utilized a library known as Librosa, which is commonly used for audio feature extraction. The primary method involved generating mel-spectrograms, which are visual representations of the frequency content of an audio signal over time. These mel-spectrograms were then saved as image files.

The dataset includes text files that outline the division of files into training, testing, and validation sets. We followed this structure to correctly allocate the images into their respective folders.

We identified a list of known classes, which include 'up', 'down', 'left', 'right', 'on', 'off', 'yes', and 'no', and established paths to the corresponding data directories. In addition to these known classes, the dataset features 'silence' and 'unknown' categories. We independently developed the 'silence' class by extracting segments from background noise recordings and supplementing them with additional silence to achieve uniform file lengths. The 'unknown' class was created by randomly selecting audio files from categories not included in our predefined list of known classes. We took around the same number of samples from each category. We also ensured that, in the end, the number of samples in the 'unknown' and 'silence' classes combined was around the same as the number of samples in the other classes.

Before images are passed into models, they are scaled to be of a size of 224x224 pixels. This particular image shape is required by the Vision Transformer. Since the generated mel-spectrograms are roughly square, this operation does not skew the data (see Figures 3 and 4; otherwise, if they were rectangles not shaped like squares, they would have to be stretched in one direction). This resizing is done for all models to ensure a fair comparison.

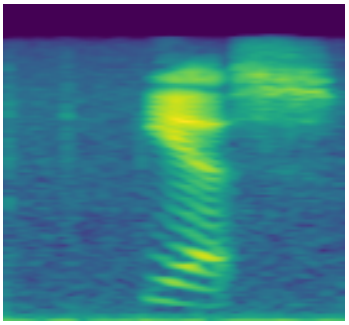

Figure 3: Example mel-spectrogram (original)

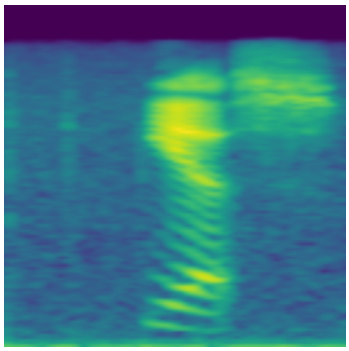

Figure 4: Example mel-spectrogram (after resize)

5 Experiments

One of the most important choices to make when using neural networks is the choice of their architecture. In this research, we explored solutions using Convolutional Neural Networks (CNN), a custom-built Convolutional Recurrent Neural Network (CRNN), and a Transformer. The aim of these experiments was to evaluate how each architecture performs, which would help us choose the best ones for our research.

The first step during experiments was to train the models on the 8 known classes and the 'other' class, making it 9 classes in total. The 'other' class was created by grouping together samples from

'silence' and 'unknown' class.

5.1 Convolutional Neural Network

The first architecture we tested was a Convolutional Neural Network. In our solution, we decided to use the ResNet-18 architecture initialized with random weights, as this architecture gave us excellent results in previous projects.

5.1.1 Setup

During the implementation of the model, we set the initialization parameters so that the model would begin training with random, untrained weights. Subsequently, we manually added a final layer to the network to establish the number of outputs as 9. The loss function employed throughout the training process was cross-entropy loss.

5.1.2 Initial Training

The initial training was launched with the specified parameters:

- batch size: 64,
- optimiser: Adam,
- learning rate: 10^{-3}.

5.1.3 Hyperparameters Tuning

Knowing that a crucial aspect of finding an optimal solution is selecting the appropriate parameters for training, we explored potential solutions using Bayesian optimization. During the search, we set the ranges of the target parameter values as follows:

- batch size: $\{2^3, 2^4, 2^5, 2^6\}$,
- learning rate: $[10^{-4}, 0.1]$,
- L2 regularisation coefficient: $[10^{-5}, 10^{-2}]$,
- beta1 (Adam): $[0, 0.99]$
- beta2 (Adam): $[0, 0.99]$

The Bayesian Optimisation algorithm was configured to sample 4 random hyperparameter combinations and then run 10 iterations of optimisation, which gives a total of 14 trainings, each one taking 10 epochs.

5.1.4 Results

The initial training produced surprisingly good results, with validation accuracy exceeding 0.9 just after 3 epochs, eventually reaching the value of 0.953. The training also did not take long - in fact, this model turned out to be the fastest one to train.

Despite the fact that we already had a very accurate model after the initial training, we still decided to run the Bayesian optimisation, hoping that we could get a model with validation accuracy even closer to 1. However, the hyperparameter optimisation did not improve the results by a lot, which is why eventually we chose the initially trained model as our final CNN. Its training history is visualised in Figures 5 (accuracy) and 6 (loss). From these figures, it is clear that the model learned rapidly, and further training beyond the third epoch did not improve its results.

We can have a closer look at this model's performance by looking at its evaluation on the test dataset, presented as a confusion matrix in Figure 7. ResNet excelled at identifying nearly all classes, achieving accuracy below 0.95 only on three classes ('down', 'no', and 'other'). The 'no' class, on which the model performed the worst, with accuracy of 0.91, was most often mistaken with 'down', 'up' and 'other' classes. The fact that the 'other' class was not the one with the worst results can be surprising, as this would seem to be the most varied class, meaning finding a pattern for it should be the most difficult (and in fact was for all the other models, which are described in later sections).

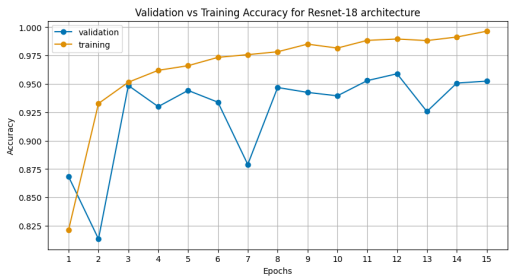

Figure 5: Resnet-18 model accuracy per epoch during training

Figure 6: Resnet-18 model loss per epoch during training

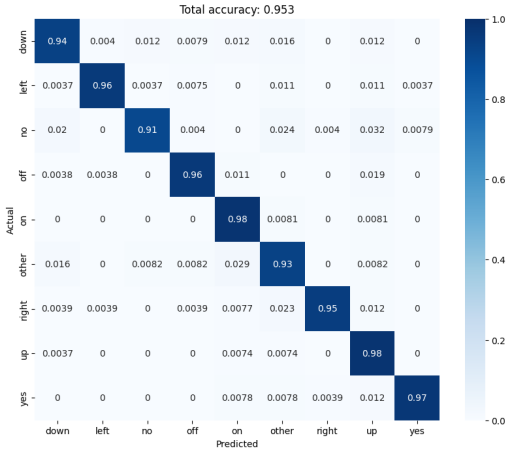

Figure 7: Confusion matrix for ResNet-18 model among known + 'other' classes

5.2 Convolutional Recurrent Neural Network

A spectrogram is not just a regular image, since its x-axis is a time dimension. This means that their classification could be an ideal use case for a recurrent network. That is why, for the second architecture, we use a CRNN.

5.2.1 Setup

The architecture we use is our custom design, which we tailored to achieve the best possible performance on this particular task. It is structured as follows:

1. Three pairs of convolutional layers, each pair followed by a 2x2 max-pool layer. In each pair, the first layer has more channels on the output than on the input, while the second layer maintains the number of channels. All convolutional layers use 3x3 kernels, and ReLU activations, with the other parameters left default. The first pair takes RGB images and has 32 channels on the output. The second pair increases the number of channels to 64, and then the final pair to 128. Outputs of this convolutional block are shaped 24x24.
2. An LSTM with 256 features in the hidden state and the other parameters left as default.
3. A fully connected layer with softmax activation. The number of outputs can be configured upon initialisation of the network and was set to 9 for this experiment.

5.2.2 Initial Training

The initial training was performed using the following parameters:

- batch size: 32,
- optimiser: Adam,
- learning rate: 10^{-4}.

5.2.3 Hyperparameters Tuning

The next step was to optimise the hyperparameters. Again we use the same technique as before, that is Bayesian optimisation. This time, the search spaces were defined as follows:

- batch size: $\{2^3, 2^4, 2^5, 2^6\}$,
- learning rate: $[10^{-5}, 10^{-3}]$,
- L2 regularisation coefficient: $[10^{-5}, 10^{-2}]$,
- beta1 (Adam): $[0, 0.99]$
- beta2 (Adam): $[0, 0.99]$

These ranges are the same as before, except for learning rate. This was changed due to the difficulties encountered during initial training, as this architecture emerged as more prone to hyperparameter changes compared to ResNet-18. We also realised that it worked better with lower learning rate values.

The Bayesian optimisation was, just like before, configured to first sample 4 random parameter sets and then perform 10 optimisation iterations. Each training lasted for 10 epochs.

5.2.4 Results

Interestingly, as was the case for ResNet-18, the Bayesian Optimisation did not improve the initial model results, with the best architecture eventually being the one which we trained with manually-set parameters at the start. The most optimal model found through hyperparameter optimisation achieved the validation accuracy of 0.829, while the initial model at the same point achieved the value of 0.911. The best model's training statistics are visualised in Figures 8 and 9.

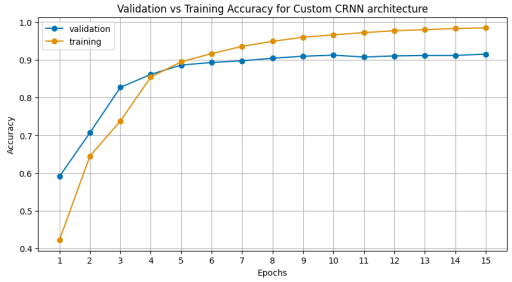

Figure 8: Custom CRNN accuracy per epoch during training

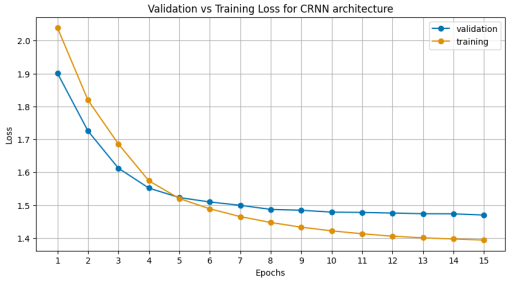

Figure 9: Custom CRNN loss per epoch during training

The final model was trained for 10 epochs, since there were signs of overfitting and no further improvement in validation loss when we tried to train it for longer than that. It is evaluated on the test dataset in Figure 10.

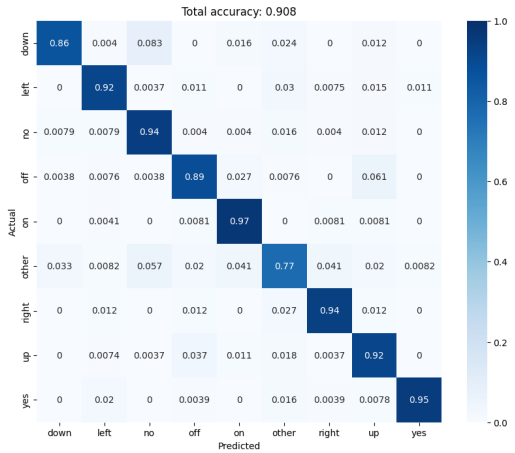

Figure 10: Confusion matrix for Custom CRNN model among known + 'other' classes

Looking at the confusion matrix, we can see that the 'other' class was the most difficult to distinguish from other classes. This is not surprising considering this class is the most varied of all, containing samples from 22 different words. The other class which CRNN found difficult to recognise was 'down', which was mistaken around 8% of the times with 'no'. Interestingly, there was no such issue in the opposite scenario, meaning that in the cases when the model was not sure between the two classes it would then usually classify the command as 'no'. Another common mistake made by this model was misclassifying 'off' as 'up' (around 6%). It was however very good at recognising 'on' and 'yes' classes (around 97% and 95% respectively).

5.3 Transformer

The last architecture we used was the Vision Transformer (ViT). Transformers are known for their ability to handle image data thanks to their built-in self-attention mechanism. By transforming our audio files into an image format, we were able to test the capabilities of this architecture.

5.3.1 Setup

In our solution, we decided to test the VIT-b-16 architecture. It is one of the smaller available VIT architectures, known as the 'base' model, with a 16x16 patch size.

5.3.2 Initial training

Initially, the ViT architecture with its original parameters was not conducive to the model's learning process for recognizing all eight 'known' classes, resulting in very long training times and low accuracy. Consequently, we decided to train the model exclusively on the 'yes', 'no', and 'other' classes, for which we managed to achieve the validation accuracy value of around 0.5. These results were surprisingly weaker than those obtained with previous network architectures. Due to long training times, we decided not to tweak the parameters manually any further in the initial training, and rather move straight to the hyperparameter optimisation.

5.3.3 Hyperparameters Tuning

A crucial aspect of training with the ViT architecture was hyperparameter tuning. We initially attempted this on all eight classes. However, it proved ineffective due to the excessive load on our available machines, combined with the prolonged duration of single epoch execution. Consequently, we decided to optimize parameters only for the model predicting the 'yes', 'no', and 'other' classes, searching within the following range of target parameter values as follows:

- batch size: $\{2^3, 2^4, 2^5\}$,
- learning rate: $[10^{-4}, 0.1]$,
- L2 regularisation coefficient: $[10^{-5}, 10^{-2}]$,
- beta1 (Adam): $[0, 0.99]$
- beta2 (Adam): $[0, 0.99]$

These ranges are similar to the ones used earlier, but a crucial difference is a smaller maximum batch size. The reason for not testing a batch size of 2^6 was an insufficient memory on the GPU that was used by us to train this model, which meant that the batch size of 2^5 was the maximum our hardware would allow us.

The Bayesian optimisation was again first drawing 4 random hyperparameter configurations and then optimising for 10 iterations. This time, each training lasted for 15 epochs.

Additionally, we decided to conduct another test by training a model predicting all eight 'known' classes plus 'other', using the parameters obtained from the Bayesian optimization of the model that predicts only 'yes', 'no', and 'other'.

5.3.4 Results

During the optimisation, we noticed that most models which trained on a low learning rate (10^{-4}) achieved good result during iterations. We were tempted to repeat the optimisation process, this time also trying even lower values of learning rate, however unfortunately due to very long training times this was not a feasible option. Still, this optimisation led us to very well-performing models, and the best results were achieved using the following parameters:

- batch size: 2^4,
- learning rate: 10^{-4},
- L2 regularisation coefficient: 10^{-5},
- beta1: 0.7268,
- beta2: 0.99

After training a model with these hyperparameters for 30 epochs, we reached a validation accuracy of 0.855 in the three-class scenario. Then, training a model for nine classes using hyperparameters obtained from the optimization of a model trained on a smaller number of classes proved to be a very good idea. It led to achieving the validation accuracy of more than 0.75, as seen on graph 11. This approach demonstrated that parameters optimized for a simpler model can effectively scale to more complex scenarios, providing a solid foundation for further experimentation and refinement in model training. As the final model, we use this one but trained only for 9 epochs, as further training (as can be seen on graphs) led to overfitting.

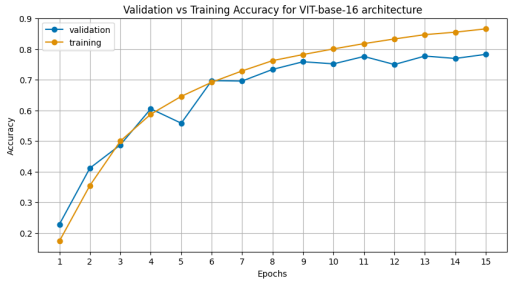

Figure 11: VIT-b-16 model accuracy per epoch during training

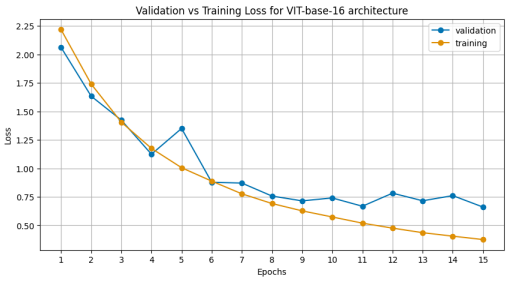

Figure 12: VIT-b-16 model loss per epoch during training

The final model has been evaluated on a test dataset, and the results are presented in Figure 13. As was the case with the CRNN, this model also struggled the most at recognising the 'other' class. The other classes it had difficulty recognising were 'no' and 'off', where for both the accuracy did not exceed 0.7. Interestingly, it performed the best on the 'down' class, where its accuracy reached 0.86 - this was the known class that the CRNN struggled the most with. Some of the common mistakes that ViT made were confusing 'left' for 'right' (11%), 'no' for 'down' (12%, note that this behaviour is opposite of that of CRNN), 'off' for 'on' or 'up' (both 12%), and 'yes' for 'left' (12%).

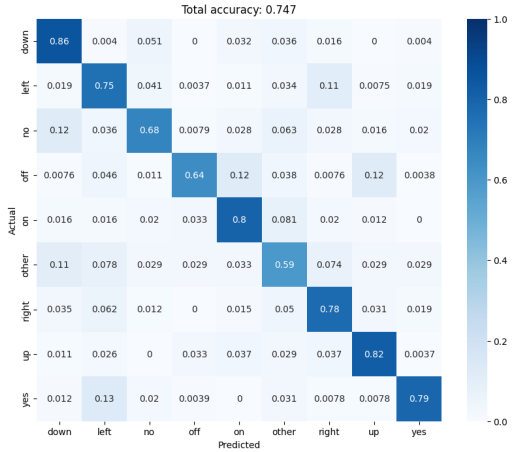

Figure 13: Confusion matrix for Custom VIT model among known + 'other' classes

5.4 Silence vs Unknown detection

In our ongoing efforts to enhance speech recognition capabilities, we focused on distinguishing between 'silence' and various 'unknown' sounds. This chapter details the specific methodologies and outcomes of our approach.

5.4.1 Setup

To identify whether a sound from the 'other' class is silence or simply a different, unknown sound, we trained additional models specifically for this purpose. For training, we used audio samples from classes that are not on the 'known' list, as well as samples of silence. As mentioned earlier, due to the limited number of silence audio samples, we included the reverse of each silence recording to double the data available. To predict the 'unknown' class, we selected random samples from each of the remaining classes, ensuring a balanced dataset across all classes to help our solution handle diverse files effectively. In testing our solutions, we used all three architectures that we applied to predicting the 'known' classes. This approach allowed us to evaluate the consistency and robustness of our models across different scenarios.

5.4.2 Results

For each architecture, we achieved a 100% accuracy rate in predicting the correct class in test dataset as seen on confusion matrices (16, 17 and 18). These results were also achieved after 2 or 3 epochs (depending on the model; see Figures 14 and 15). This meant that there was no need for any hyperparameter optimisation.

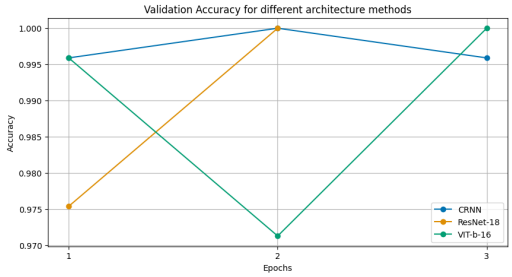

Figure 14: Comparison of accuracy per epoch during training for different architecture methods

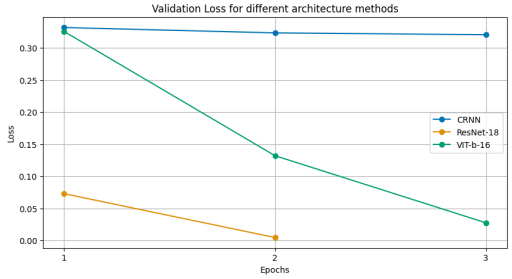

Figure 15: Comparison of loss per epoch during training for different architecture methods

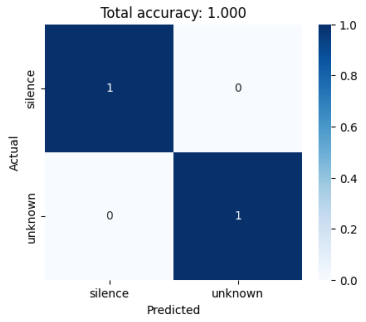

Figure 16: Confusion matrix for 'other' class using CRNN

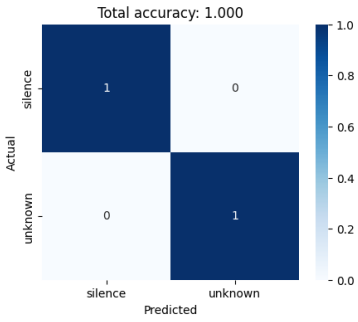

Figure 17: Confusion matrix for 'other' class using ResNet-18

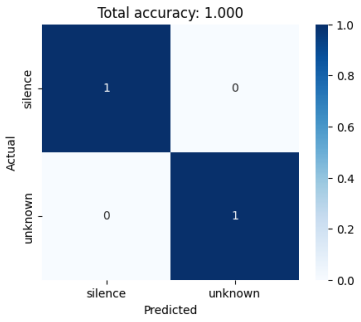

Figure 18: Confusion matrix for 'other' class using ViT

We believe that this exceptional performance is largely due to the distinctive spectral features of silence compared to other classes. The spectrogram of silence distinctly lacks the complex patterns observed in the spectrograms of the sound-containing classes. Visually, the silence class spectrogram appears significantly more uniform, which likely simplifies the classification task for the models (see Figures 19 and 20). Furthermore, the inclusion of reversed silence samples might have contributed to the models' ability to generalize well within this specific class by enhancing the diversity of training data.

5.5 Merging the models

This section explores the integration of models for recognizing 'known' classes with those for classifying 'silence' and 'unknown'. This merging allows for a comprehensive approach to handling various inputs, improving the accuracy and flexibility of our system. The results show that this combined approach can achieve high accuracy, providing a reliable classification.

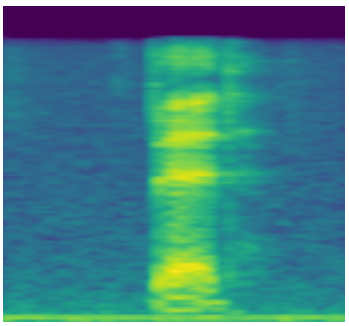

Figure 19: Example spectrogram from 'unknown' class

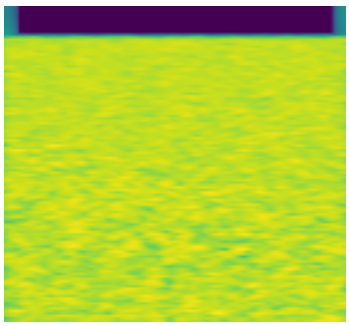

Figure 20: Example spectrogram from 'silence' class

5.5.1 Setup

To facilitate the integration of models recognizing the 'known' classes with a classifier for 'others' into 'silence' and 'unknown', we have created an additional class called `MergedModel`. This setup enables the merging of these two models by initially passing the spectrogram through the model predicting 'known' classes. If this model predicts the class as 'other', the input is then directed to a model that differentiates between 'silence' and 'unknown'. Due to the excellent performance observed in distinguishing between 'silence' and 'unknown', we decided to merge the models based on their architecture.

5.5.2 Results

For the final models, we achieved satisfactory results. The best model in our case was the ResNet model, which quickly and accurately learned to recognize classes with an accuracy of 0.95. This model achieved high results, with an accuracy above for all classes except the 'unknown' class, which reached an accuracy of 0.89. This is still a high result for this class compared to other models.

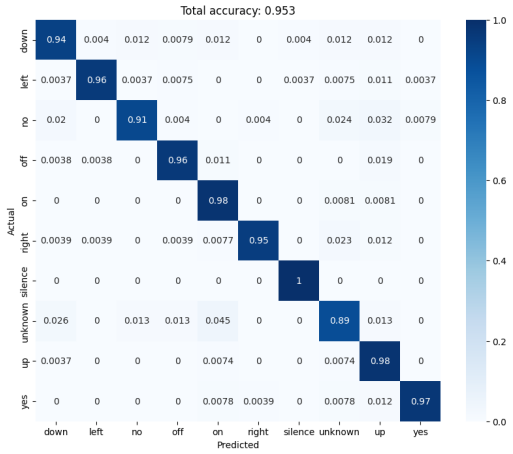

Figure 21: Resnet-18 merged

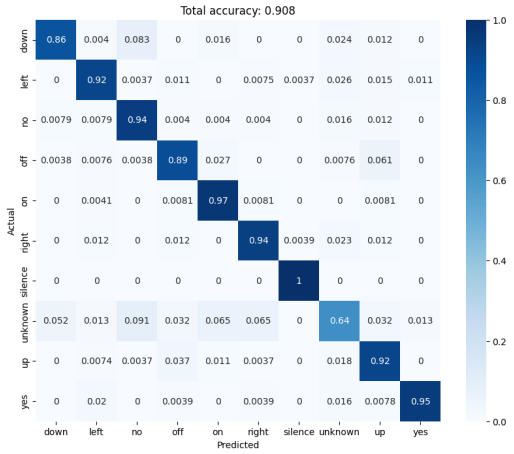

Figure 22: Custom CRNN merged

With the classification using a model based on CRNN architecture, we achieved an accuracy of 0.90. The model had the most difficulty with the 'unknown' class, where the accuracy was 0.64. Among the misclassifications outside the 'unknown' class, the most frequently confused classes were 'no' and 'down'. Additionally, the actual 'right' class was often misidentified as 'on,' while the 'on' class, apart from being confused with 'unknown,' was also mistaken for 'up'. We can see that classes are not misinterpreted symmetrically—if one class is confused with another, it does not necessarily mean the reverse is true.

With the Vision Transformer (ViT) model, we achieved an accuracy of 0.74. A major issue with this architecture is the model's inability to classify 'unknown' classes correctly. All classes, except for the 'silence' class, are often misclassified as 'unknown'. Other errors include frequent classification of the 'no' class as 'down' and 'yes' as 'left'. We can observe that in this model, classification errors are not symmetrical. The model does not misclassify the 'left' class as 'yes'

as often, but much more frequently as 'right'. The model performs best in recognizing the classes 'down,' 'on,' and 'up,' with an accuracy above 0.8.

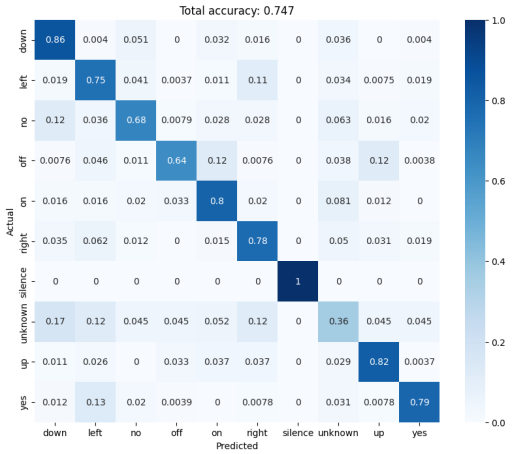

Figure 23: ViT-b-16 merged

The one class that no model misclassified was the 'silence' class. As mentioned earlier, it might be due to spectrograms from this class being clearly distinct from all other classes (see Figure 20). All models achieved accuracy of 1 on silence class, with ViT also being the only one not to mistake any other class for 'silence'.

Our approach to integrating models for recognizing 'known' classes with those for classifying 'silence' and 'unknown' has shown promising results. By employing a merged model strategy, we were able to achieve high accuracy in classifying most categories. The ResNet-based model excelled, achieving an overall accuracy of 0.95, while the CRNN-based model delivered a commendable accuracy of 0.90, despite struggles with the 'unknown' class. The Vision Transformer (ViT) model, although exhibiting issues with misclassifications, achieved a respectable accuracy of 0.74.

A key takeaway is that misclassification errors are not always symmetrical. A class frequently misidentified as another does not guarantee the reverse is true. This highlights the importance of carefully examining the unique patterns in misclassification. The consistent accuracy across all models for the 'silence' class underscores its distinctiveness, which can serve as a foundation for further optimization.

6 Conclusions

Eventually, we were able to achieve satisfactory results on all models predicting in a 10-class variant, which we consider a success. The final accuracy values on testing data were 0.953 for a CNN (ResNet-18), 0.908 for a CRNN (custom), and 0.747 for a Transformer (ViT). These results prove that it is not always the most complex architecture that is the best fitted for any task, as ResNet-18, which was the best model overall, was also the quickest one to train.

This project helped us gather experience in working with recurrent networks when building our own Convolutional Recurrent Neural Network architecture. We also gained skills and understanding of dealing with transformers by utilising and training the Vision Transformer. Unfortunately, the high computational cost of training transformers meant it was impossible for us to train ViT to its full potential, or try a larger architecture. Still, we were able to find a way to optimise training times by working with fewer classes, and later utilising the same parameters for a more complex scenario. While this approach definitely has its limitations, it was the only feasible strategy considering the time and hardware constraints. Admittedly, it did help us significantly improve the model's performance and eventually achieve results of around 75% accuracy in 10-class classification, which we consider a success.

Additionally, the balanced dataset across all classes, including the diverse "unknown" sounds, ensured that our models were well-trained to handle different acoustic characteristics without bias towards more frequently occurring features. This preparation was crucial for maintaining high performance and consistency in our tests, demonstrating the adaptability and reliability of our architectures in real-world audio processing scenarios.

References

David, L. (2023), 'Self-Attention: A step-by-step guide to calculating the context vector'.
URL: *https://medium.com/@lovelyndavid/self-attention-a-step-by-step-guide-to-calculating-the-context-vector-3d4622600aac*

Donges, N. (2024), 'A complete guide to Recurrent Neural networks (RNNs)'.
URL: *https://builtin.com/data-science/recurrent-neural-networks-and-lstm*

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J. & Houlsby, N. (2021), 'An image is worth 16x16 words: Transformers for image recognition at scale'.

He, K., Zhang, X., Ren, S. & Sun, J. (2015), 'Deep residual learning for image recognition'.

Hochreiter, S. & Schmidhuber, J. (1997), 'Long Short-Term memory', *Neural computation* **9**(8), 1735–1780.
URL: *https://doi.org/10.1162/neco.1997.9.8.1735*

inversion, Julia Elliott, M. M. P. W. R. (2017), 'Tensorflow speech recognition challenge'.
URL: *https://kaggle.com/competitions/tensorflow-speech-recognition-challenge*

Nouman (2021), 'Writing CNNs from Scratch in PyTorch'.
URL: *https://blog.paperspace.com/writing-cnns-from-scratch-in-pytorch/*

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. & Chintala, S. (2019), PyTorch: An Imperative Style, High-Performance Deep Learning Library, *in* H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox & R. Garnett, eds, 'Advances in Neural Information Processing Systems 32', Curran Associates, Inc., pp. 8024–8035.
URL: *http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf*

Roberts, L. (2024), 'Understanding the MEL Spectrogram - Analytics Vidhya - Medium'.
URL: *https://medium.com/analytics-vidhya/understanding-the-mel-spectrogram-fca2afa2ce53*

Rosebrock, A. (2023), 'PyTorch: Training your first Convolutional Neural Network (CNN) - PyImageSearch'.
URL: *https://pyimagesearch.com/2021/07/19/pytorch-training-your-first-convolutional-neural-network-cnn/*

Samar, A. (2023), 'Fine-Tuning a Pre-Trained ResNet-18 Model for Image Classification with PyTorch'.
URL: *https://alirezasamar.com/blog/2023/03/fine-tuning-pre-trained-resnet-18-model-image-classification-pytorch/*

Tam, A. (2023), 'Using dropout regularization in PyTorch models'.
URL: *https://machinelearningmastery.com/using-dropout-regularization-in-pytorch-models/*

Tong Yu, H. Z. (2020), 'Hyper-parameter optimization: A review of algorithms and applications'.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. & Polosukhin, I. (2017), 'Attention is all you need', *CoRR* **abs/1706.03762**.