Deep Learning Course – Second Project Report

Speech Recognition - Transformers

Marta Szuwarska
01171269@pw.edu.pl
320662

Mateusz Nizwantowski
01161932@pw.edu.pl
313839

April 30, 2025

Contents

1 Introduction

This report presents our second project in the Deep Learning course in Data Science Masters at the Faculty of Mathematics and Information Science. The project aims to introduce us to transformers by solving speech commands recognition problem. On top of it we research other methods to solve this problem such as GRU and LSTM. We were informed that this work's goal is purely educational and research-focused, but truth be told this time we tried to create the best recognition model we could. We managed to achieve over 97.6 % of accuracy on validation set, which in our opinion is pretty good result.

For more details about the project and to access the code, please refer to the project repository: https://github.com/szuvarska/DeepLearning.

1.1 Project details

The exact topic of this project is *Speech commands classification with Transformers*. We have been given the kaggle `TensorFlow Speech Recognition Challenge`, there is data provided and we must use it for this project. Following the success of previous project we decided to use PyTorch as a deep learning framework. This time we focused more on final performance of the model. While project is fairly open in its definition there are areas and experiments that we have to cover:

1. Test and compare different network architectures (at least one of them should be a Transformer).
2. Investigate influence of parameters change on the obtained results.
3. Present confusion matrix (with appropriate discussion).
4. In case of accuracy or efficiency problems, start with a subset of classes (e.g., only the 'yes' and 'no' commands).
5. Pay special attention to the "silence" and "unknown" classes—test different approaches (e.g., a separate network for their recognition, under/oversampling, etc.).

1.2 Dataset description

As mentioned previously data comes form `TensorFlow Speech Recognition Challenge`. The challenge was held at the end of 2017, hosted by Google Brain and is designed to facilitate research in low-latency, real-time speech recognition on edge devices. In the challenge description there are strict requirements about performance on Raspberry Pi 3.

The dataset consists of over 65,000 one-second long utterances of 30 short words, recorded by thousands of different people. Each clip contains a single spoken English word recorded in various acoustic conditions. The audio samples are grouped into **30 classes**, but the core task focuses on identifying the following **12 keywords**:

1. Yes
2. No
3. Up
4. Down
5. Left
6. Right
7. On
8. Off
9. Stop
10. Go
11. Unknown (remaining words not in the target list)
12. Silence (background noise samples)

Each audio sample is stored as a `.wav` file and includes metadata such as speaker ID and recording environment. This makes the dataset particularly suitable for training models that generalize across speakers and noise conditions.

Here are 30 distinct spoken words present in this dataset and it is a mix of common words and auxiliary words, which are included to help build more robust models:

> `Yes`, `No`, `Up`, `Down`, `Left`, `Right`, `On`, `Off`, `Stop`, `Go`, `Zero`, `One`, `Two`, `Three`, `Four`, `Five`, `Six`, `Seven`, `Eight`, `Nine`, `Bed`, `Bird`, `Cat`, `Dog`, `Happy`, `House`, `Marvin`, `Sheila`, `Tree`, and `Wow`.

1.3 Project setup

The project is created using Python Jupyter notebooks. The core packages used in the project include:

- `torch` (2.6.0),
- `torchvision` (0.21.0),
- `matplotlib` (3.10.1),
- `numpy` (2.2.3),
- `timm` (1.0.15),
- `scikit-learn` (1.6.1),
- `seaborn` (0.13.2).

While additional dependencies were used throughout the project, we have chosen not to list them all here, as they are not essential to the main workflow. Throughout the project, we iterated on various methods. Initially, we again had planned to use Marimo notebooks, but after careful consideration - taking into account the tight deadline and our lack of prior experience with this new technology - we decided to stick with the more familiar Jupyter Notebooks.

Additionally, we organized our work with a well - structured directory setup in our GitHub repository to ensure smooth collaboration and clarity. All experiments are fully reproducible, as we set a seed at the beginning of the notebooks. To replicate the results, execute the notebooks in the order indicated by the numbers in the filenames.

Note that the data is not included in the repository; it should be downloaded from the source specified in the previous subsection. There is one notebook called `00_setup_data_split.ipynb` that is in responsible for data preparation and management. This "script" creates out of download from Kaggle data, a nice structure that is easy to work with, similar setup like in first project, with 3 directories: train, test, validation, with silence data created out of long audio tracks. We will describe this process in more detail in section dedicated to data reading 3.

The recommended execution order is as follows:

1. Notebooks in the `notebooks/data_exploration` directory,
2. Notebooks in the `notebooks/model_building` directory,
3. Notebooks in the `notebooks/transformer_buidling` directory.

However, the order of execution does not affect the final results. To start, ensure that Python is installed and all dependencies listed in the `requirements.txt` file are met.

When it comes to the hardware, we used our own desktop PC. We only will focus at the most important components: GPU, CPU, RAM and storage.

1.
   - GPU: Nvidia 4070 Super (slightly underclocked)
   - CPU: AMD Ryzen 7600
   - RAM: 32GB 6000MHz
   - SSD: 2TB read 7000MB/s; write 6000MB/s
2.
   - GPU: Nvidia 1050Ti 4GB
   - CPU: AMD Ryzen 3800X
   - RAM: 16GB 3600MHz
   - SSD: 512 GB read 3500MB/s; write 3000MB/s

2 Theory

2.1 Waveform

A waveform is a one-dimensional representation of an audio signal, typically encoding the variation in air pressure (or an analogous signal) over time as captured by a microphone. In digital audio processing, the waveform is a sequence of amplitude values sampled at a fixed rate (e.g., 16 kHz). It serves as the raw input in many speech processing pipelines. While waveforms contain all the necessary information, they can be difficult for models to interpret directly due to their high temporal resolution and lack of explicit frequency decomposition [1].

2.2 Spectogram

A spectrogram is a two-dimensional time-frequency representation of a waveform, obtained by applying the Short-Time Fourier Transform (STFT). It shows how the spectral content of a signal changes over time, with one axis representing time, the other representing frequency, and the color or intensity indicating amplitude. Spectrograms are a commonly used input representation in audio classification and speech recognition tasks, as they make important features like formants and harmonics more accessible to machine learning models [2].

2.3 MFCC Spectogram

A Mel-Frequency Cepstral Coefficient (MFCC) spectrogram is a time-frequency representation that captures the short-term power spectrum of a signal based on a nonlinear Mel scale of frequency. It is derived by applying the Short-Time Fourier Transform (STFT), mapping the powers of the spectrum onto the

Mel scale, taking the logarithm of the powers, and then applying a Discrete Cosine Transform (DCT) to decorrelate the features. The resulting coefficients, typically the lower ones, represent the broad spectral shape and are stacked over time to form a 2D representation. MFCC spectrograms are widely used in speech and audio processing tasks, as they approximate human auditory perception and provide compact, informative features for machine learning models [3].

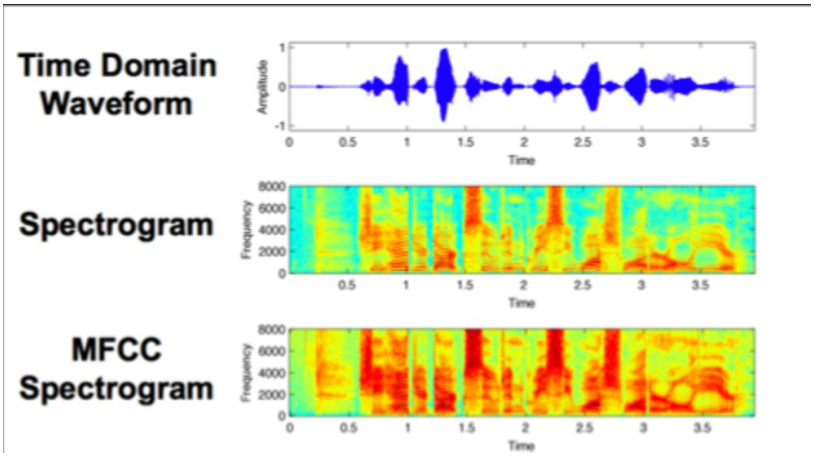


Figure 1: Visualization of the mentioned data representation

2.4 Transformer

The Transformer is a deep learning architecture introduced by Vaswani et al. [4], based on the self-attention mechanism, which allows the model to weigh the importance of different positions in the input sequence dynamically. Unlike recurrent models, Transformers process the entire sequence in parallel, making them more computationally efficient and better suited to capturing long-range dependencies. Though originally developed for natural language processing, Transformers have recently been applied to audio tasks, where they can model temporal patterns in speech more flexibly than convolutional networks [5].

2.5 Strided convolutions

Strided convolutions are a variation of the standard convolution operation in which the filter is applied at intervals larger than one, effectively downsampling the input. For instance, a stride of 2 reduces the spatial (or temporal) dimension by half. Strided convolutions are commonly used for reducing the computational burden and increasing the receptive field of deeper layers [6]. However, large strides may lead to loss of fine-grained information, which is especially relevant when dealing with short or dense signals like speech.

2.6 Self-attention

Self-attention is the core mechanism of Transformer architectures. It allows each position in the input to attend to all other positions, enabling the model to dynamically capture relationships between distant elements in the sequence [4]. In audio tasks, self-attention enables the model to integrate temporal information more effectively than fixed-size convolutional kernels.

2.7 Positional encoding

Since Transformers lack recurrence or convolution, they do not inherently encode information about the order of input tokens. Positional encodings are added to the input embeddings to provide the model with information about the relative or absolute position of each element in the sequence. This is crucial for processing speech or time-series data, where the temporal structure carries essential semantic information [4].

3 Data preparation

3.1 Data transformation

We created a custom script that creates out of download from kaggle data, a nice structure that is easy to work with, similar to setup from first project where you had 3 directories: train, test, validation. Each of these directories has the same structure in it. Sub-directories named after the class name for example: "Down", "Stop". We prefer to do it once, save it in local directory and not worry about it in the future.

3.2 Handling silence

There are also six special files, in directory `_background_noise_`, they are significantly longer than the other one. In order to work with them we had to split them. To do this we used `pydub` Python package. Those tracks were 60 seconds long. We processed each background audio file by splitting it into overlapping chunks of 1000 ms duration. This was achieved by sliding a 1-second window over the audio with a hop size of 200 ms. We did this to have more samples and avoid excessive imbalance.

After chunking, we split the resulting audio segments into **training** (80%), **validation** (10%), and **test** (10%) subsets. To avoid data leakage from overlap we didn't split it randomly, instead to training goes first 80% of audio length, next 10% to validation and remaining tracks goes to test set. The chunks were then saved into separate directories under a `silence` label, preserving the source filename for traceability.

3.3 Data loading

To efficiently manage audio data, we developed a custom `SpeechCommandsDataset` class. Each instance of this class scans a specified root directory, recursively collecting paths to all `.wav` files contained within labeled subdirectories. Each subdirectory name is treated as a class label. Audio samples are loaded using `torchaudio.load`, and each waveform is either truncated or zero-padded to a fixed length of 16,000 samples (corresponding to one second at 16 kHz). This ensures that all input tensors share the same dimensions, which is necessary for batch processing in PyTorch.

The dataset supports optional waveform transformations, applied after padding and truncation. This makes it suitable for both preprocessing and data augmentation. It also provides a label-to-index mapping through the `class_to_idx` property. During training and evaluation, `SpeechCommandsDataset` is wrapped in `DataLoader` objects with a batch size of 16. The training loader uses `shuffle=True` to ensure data randomness across epochs, and both loaders are configured with `num_workers=6` to enable parallel data loading for performance optimization.

3.4 Handling unknown classes

To reflect open-set recognition scenarios, the dataset supports two modes for labeling: `original` and `modified`. In the `original` mode, each subdirectory label is treated as an individual class. In contrast, the `modified` mode reclassifies any label not included in a predefined list of known commands (e.g., *yes*, *no*, *up*, etc.) into a single `unknown` class. This reduces the total number of classes and enables the model to distinguish between familiar and unfamiliar commands.

However, grouping many out-of-vocabulary commands into a single `unknown` class introduces class imbalance, with the `unknown` class having significantly more samples than the others. To counteract this, we calculate class weights that are inversely proportional to the number of samples in each class. These weights are normalized and passed to `nn.CrossEntropyLoss`, ensuring that underrepresented classes are not overwhelmed during optimization. This reweighting scheme is particularly important when using the `modified` mode and leads to improved performance across both major and minor classes.

4 Experiments

4.1 Transformer

The goal of this project is to evaluate how well different Transformer-based models can learn from time-frequency representations of audio and to compare their performance against more conventional approaches. The following sections present the experimental setups, training strategies, and results obtained using various Transformer configurations tailored to the speech command recognition task.

4.1.1 Early architecture

Our initial model architecture consists of three main components: a feature extraction frontend based on mel-spectrograms and convolutional layers, a Transformer encoder to model temporal dependencies, and a linear classification head. The total number of trainable parameters is approximately 2.15 million (calculated by torchinfo package).

Raw waveform inputs are sampled at 16 kHz and first transformed into a mel-spectrogram using `torchaudio.transforms.MelSpectrogram`. This produces a time-frequency representation of shape $(64, 101)$ for each input, where 64 is the number of mel bands and 101 is the number of time frames. These are then converted to decibel scale using `AmplitudeToDB`.

The resulting spectrogram is passed through a lightweight convolutional block composed of two convolutional layers. Each convolution is followed by batch normalization and a ReLU activation. Specifically, the first convolution maps the input from 1 to 32 channels, and the second from 32 to 64 channels, preserving the spatial resolution throughout (stride=1). The output of this CNN block is a tensor of shape $(64, 64, 101)$.

To interface with the Transformer encoder, the CNN output is flattened across the frequency and time dimensions, resulting in a sequence of 6464 tokens (64×101). Each token is projected to a 256-dimensional embedding space via a linear projection layer. A learnable `[CLS]` token is prepended to the sequence, increasing its length to 6465.

The core of the model consists of a Transformer encoder with 4 identical layers. Each `TransformerEncoderLayer` uses 4 attention heads, a feedforward hidden dimension of 512, GELU activations, and dropout of 0.1. The Transformer operates on the full input sequence, including the `[CLS]` token.

Finally, the output corresponding to the `[CLS]` token is passed through a fully connected layer that maps from 256 to 12 output classes. Details of the architecture are depicted in Figure 2.

Model summary:

- Total parameters: **2,147,404**
- Input size (batch size 16): **1.02 MB**
- Forward/backward pass memory: **4607.58 MB**
- Estimated total memory usage: **4.6 GB**
- Total multiply-add operations: **1.96 GigaOps**

Despite its simplicity, this architecture suffers from significant computational cost due to the very long input sequences (6465 tokens), leading to slow training and memory inefficiency.

```
Layer (type:depth-idx)                        Output Shape              Param #
==========================================================================================
SpeechCommandTransformer                      [16, 12]                  256
├─MelSpectrogram: 1-1                         [16, 64, 101]             --
│    └─Spectrogram: 2-1                       [16, 201, 101]            --
│    └─MelScale: 2-2                          [16, 64, 101]             --
├─AmplitudeToDB: 1-2                          [16, 64, 101]             --
├─Sequential: 1-3                             [16, 64, 64, 101]         --
│    └─Conv2d: 2-3                            [16, 32, 64, 101]         320
│    └─BatchNorm2d: 2-4                       [16, 32, 64, 101]         64
│    └─ReLU: 2-5                              [16, 32, 64, 101]         --
│    └─Conv2d: 2-6                            [16, 64, 64, 101]         18,496
│    └─BatchNorm2d: 2-7                       [16, 64, 64, 101]         128
│    └─ReLU: 2-8                              [16, 64, 64, 101]         --
├─Linear: 1-4                                 [16, 6464, 256]           16,640
├─TransformerEncoder: 1-5                     [16, 6465, 256]           --
│    └─ModuleList: 2-9                        --                        --
│    │    └─TransformerEncoderLayer: 3-1      [16, 6465, 256]           527,104
│    │    └─TransformerEncoderLayer: 3-2      [16, 6465, 256]           527,104
│    │    └─TransformerEncoderLayer: 3-3      [16, 6465, 256]           527,104
│    │    └─TransformerEncoderLayer: 3-4      [16, 6465, 256]           527,104
├─Linear: 1-6                                 [16, 12]                  3,084
==========================================================================================
Total params: 2,147,404
Trainable params: 2,147,404
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 1.96
==========================================================================================
Input size (MB): 1.02
Forward/backward pass size (MB): 4607.58
Params size (MB): 4.38
Estimated Total Size (MB): 4612.98
==========================================================================================
```

Figure 2: Architecture of early transformer model.

4.1.2 Training strategy

The training strategy for the transformer model is designed to ensure efficient learning while minimizing overfitting. The strategy incorporates key elements such as loss computation, optimization, model evaluation, and early stopping, as described below.

During training, the model is fed waveforms through the training data loader. These waveforms are preprocessed to remove the unnecessary channel dimension, resulting in inputs with the shape `(batch_size, samples)`. The model's performance is evaluated based on its ability to predict labels for these inputs.

Hyperparameters: By default, the model uses `CrossEntropyLoss` as the criterion for classification, with the Adam optimizer and a learning rate of 0.001. A learning rate scheduler, `StepLR`, is employed to decrease the learning rate by a factor of 0.1 every 10 epochs, helping the model to converge effectively in later training stages.

The training loop consists of multiple epochs, and for each epoch, the model is trained in batches. Each batch is processed as follows:

- The model's weights are updated using backpropagation, with the loss computed by comparing the model's output with the true labels.

- The optimizer updates the model's parameters using gradients computed from the loss.
- The accuracy of the model is calculated by comparing the predicted labels with the true labels.

In addition to the training phase, the model is evaluated on the test set at the end of each epoch. During evaluation, the model is set to evaluation mode, where layers such as dropout are turned off. The test loss and accuracy are computed for the test data, providing insight into the model's generalization ability.

Early Stopping: To prevent overfitting, early stopping is implemented. If the test accuracy does not improve after a specified number of consecutive epochs (patience, by default 3), training is halted. This strategy helps avoid unnecessary training and saves computational resources.

The following metrics are tracked during training:

- **Train Loss**: The loss computed on the training dataset.
- **Train Accuracy**: The accuracy of the model on the training dataset.
- **Test Loss**: The loss computed on the test dataset.
- **Test Accuracy**: The accuracy of the model on the test dataset.

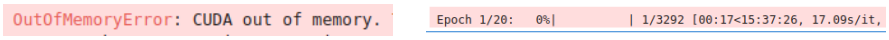
The model is trained on a GPU to speed up the training process, ensuring efficient computation.

4.1.3 Early results

Initially, the training dataset was highly imbalanced, containing approximately 32,550 samples in the *unknown* class and around 1,800 samples per each of the remaining classes. As a result, the model failed to learn meaningful distinctions between classes, instead defaulting to always predicting *unknown*. This led to a deceptively high initial accuracy of around 63% that remained unchanged throughout early training attempts.

Upon recognizing this issue, we corrected it by assigning appropriate class weights in the cross-entropy loss function, as described in Section 3.4. Since this change addressed the imbalance, we do not revisit the issue further and instead focus on the results obtained from balanced training.

We used the architecture outlined in the previous subsection, training for up to 20 epochs using default hyperparameters. Initially, we attempted to train with a batch size of 16 and an embedding dimension of 256 on a GeForce GTX 1050 Ti GPU. However, this configuration led to frequent `CUDA out of memory` errors (see Figure 3a). Reducing the embedding dimension to 64 allowed the training to proceed, but the estimated time for a single epoch exceeded 15 hours (Figure 3b).



(a) Screenshot of `CUDA out of memory` error.

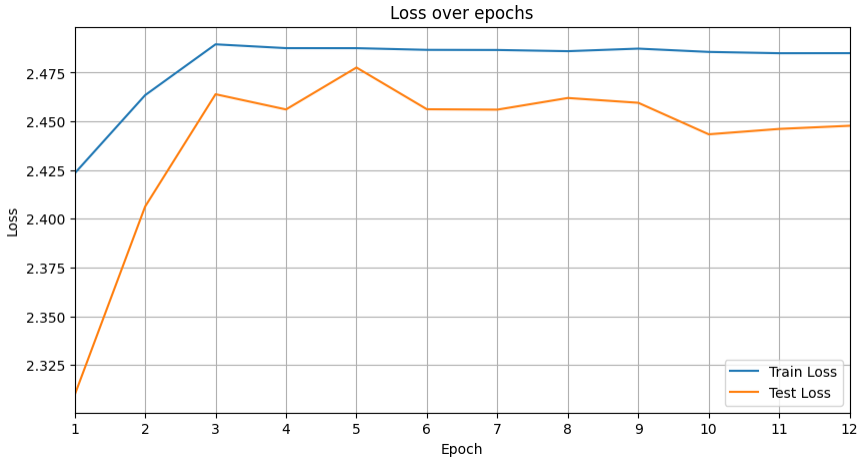
(b) Screenshot of ETA for 1 epoch exceeding 15 hours.

Figure 3: Encountered problems during initial attempts of training transformers.

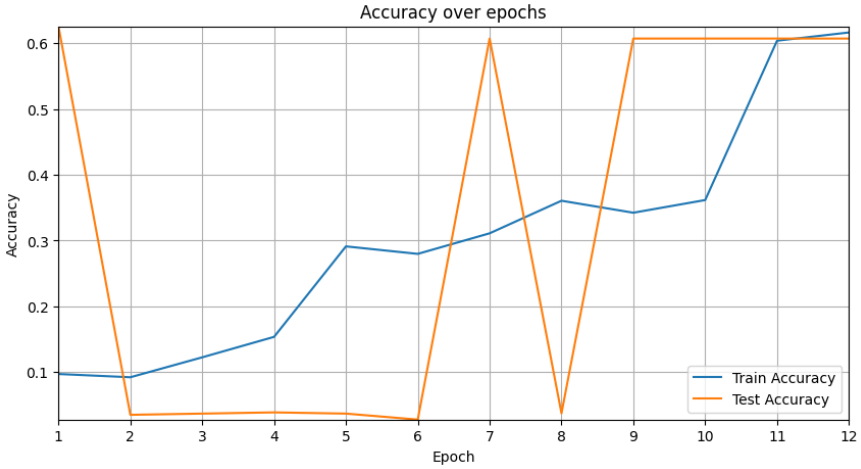
To accelerate training, we switched to a GeForce RTX 4070 Super, which enabled us to use a batch size of 32 and an embedding dimension of 128. With this setup, we successfully trained the model for 12 epochs before early stopping, completing the run in approximately 13 hours.

Despite improved training conditions, the results were inconsistent and, at times, counterintuitive. While training accuracy increased steadily (Figure 4), the test accuracy remained highly unstable. This discrepancy suggests potential issues such as overfitting, poor generalization, or instability in learning dynamics.

Training was stopped early at epoch 12. The best test accuracy of 60.71% was observed in several epochs, while the best corresponding training accuracy at that point was only 34.23%. This gap suggests the model may have learned to exploit the structure of the test set without genuine improvement in class-level discrimination. These findings point toward the need for further investigation into model regularization, evaluation robustness, and possibly the quality of the dataset split.



(a) Loss over time.



(b) Accuracy over time.

Figure 4: Loss and accuracy over time for an early transformer model with a batch size of 32 and an embedding dimension of 128.

4.1.4 Strided architecture

To address the computational inefficiencies and underfitting observed in the original model, we introduced a number of architectural modifications targeting both the convolutional frontend and the sequence modeling pipeline. The overall structure of the model remains similar, with a feature extraction stage followed by a Transformer encoder and a classification head. However, key changes were made to reduce sequence length, improve efficiency, and enable training of

deeper or wider networks under GPU memory constraints.

The most significant difference lies in the design of the convolutional block. In the original architecture, both convolutional layers used a stride of 1, which preserved the full temporal and frequency resolution of the mel-spectrogram. This resulted in an output of shape $(64, 64, 101)$ from the CNN, and consequently a flattened sequence length of 6464 tokens before the Transformer encoder. Such a long sequence incurred substantial memory and compute overhead due to the quadratic complexity of self-attention.

In the revised architecture, we introduced downsampling by setting a stride of 2 in the second convolutional layer. This reduced the spatial dimensions of the CNN output to $(64, 16, 26)$, effectively decreasing the sequence length from 6464 to 416. This sequence is projected to a 256-dimensional embedding space, and a learnable `[CLS]` token is prepended as before, resulting in a final sequence length of 417 for the Transformer encoder. This modification dramatically improves training speed and memory efficiency while preserving sufficient resolution for learning discriminative features.

The Transformer encoder itself remains unchanged in structure, consisting of 4 layers with 4 attention heads, a model dimension of 256, and a feedforward width of 512. However, due to the reduced input sequence length, the computational cost of the Transformer is significantly lower in the revised model. This enables training with the same batch size and embedding dimension as before, while requiring far less memory per forward/backward pass.

Depending on the input data (original classes or modified classes), the output dimension of the final linear layer was adjusted to match the number of target classes (12 or 31). Results for both variants are presented later in this report.

Despite these modifications, the total number of parameters remains nearly the same—approximately 2.15 million. However, the estimated number of multiply-add operations has dropped from 1.96 gigaflops to just 148.75 megaflops for original classes and 148.68 megaflops for modified classes, and the total memory footprint has decreased from approximately 4.6 GB to 312 MB. These improvements make the model significantly more scalable and better suited for experimentation with larger architectures or longer training schedules. Details of the architectures are depicted in Figure 5 and 6.

Model summary for original classes:

- Total parameters: **2,152,287**
- Input size (batch size 16): **1.02 MB**
- Forward/backward pass memory: **307.11 MB**
- Estimated total memory usage: **312.53 MB**
- Total multiply-add operations: **148.75 MegaOps**

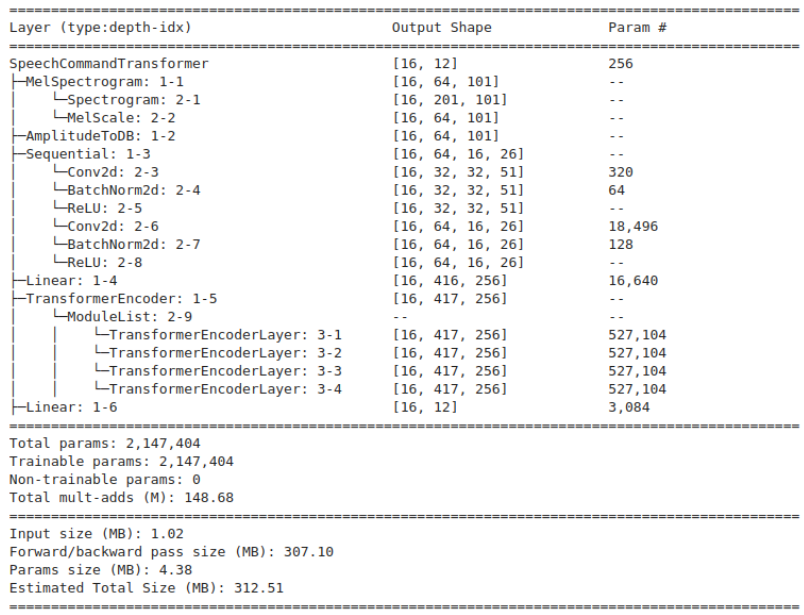
Model summary for modified classes:

- Total parameters: **2,147,404**
- Input size (batch size 16): **1.02 MB**
- Forward/backward pass memory: **307.10 MB**
- Estimated total memory usage: **312.51 MB**
- Total multiply-add operations: **148.68 MegaOps**

Overall, this revised architecture maintains the expressive power of the original Transformer while solving its inefficiency and underfitting issues through careful architectural optimization and dimensionality reduction.

```
==========================================================================================
Layer (type:depth-idx)                        Output Shape              Param #
==========================================================================================
SpeechCommandTransformer                      [16, 31]                  256
├─MelSpectrogram: 1-1                         [16, 64, 101]             --
│    └─Spectrogram: 2-1                       [16, 201, 101]            --
│    └─MelScale: 2-2                          [16, 64, 101]             --
├─AmplitudeToDB: 1-2                          [16, 64, 101]             --
├─Sequential: 1-3                             [16, 64, 16, 26]          --
│    └─Conv2d: 2-3                            [16, 32, 32, 51]          320
│    └─BatchNorm2d: 2-4                       [16, 32, 32, 51]          64
│    └─ReLU: 2-5                              [16, 32, 32, 51]          --
│    └─Conv2d: 2-6                            [16, 64, 16, 26]          18,496
│    └─BatchNorm2d: 2-7                       [16, 64, 16, 26]          128
│    └─ReLU: 2-8                              [16, 64, 16, 26]          --
├─Linear: 1-4                                 [16, 416, 256]            16,640
├─TransformerEncoder: 1-5                     [16, 417, 256]            --
│    └─ModuleList: 2-9                        --                        --
│    │    └─TransformerEncoderLayer: 3-1      [16, 417, 256]            527,104
│    │    └─TransformerEncoderLayer: 3-2      [16, 417, 256]            527,104
│    │    └─TransformerEncoderLayer: 3-3      [16, 417, 256]            527,104
│    │    └─TransformerEncoderLayer: 3-4      [16, 417, 256]            527,104
├─Linear: 1-6                                 [16, 31]                  7,967
==========================================================================================
Total params: 2,152,287
Trainable params: 2,152,287
Non-trainable params: 0
Total mult-adds (M): 148.75
==========================================================================================
Input size (MB): 1.02
Forward/backward pass size (MB): 307.11
Params size (MB): 4.40
Estimated Total Size (MB): 312.53
==========================================================================================
```

Figure 5: Architecture of a strided transformer model for original classes.



```
==========================================================================================
Layer (type:depth-idx)                        Output Shape              Param #
==========================================================================================
SpeechCommandTransformer                      [16, 12]                  256
├─MelSpectrogram: 1-1                         [16, 64, 101]             --
│    └─Spectrogram: 2-1                       [16, 201, 101]            --
│    └─MelScale: 2-2                          [16, 64, 101]             --
├─AmplitudeToDB: 1-2                          [16, 64, 101]             --
├─Sequential: 1-3                             [16, 64, 16, 26]          --
│    └─Conv2d: 2-3                            [16, 32, 32, 51]          320
│    └─BatchNorm2d: 2-4                       [16, 32, 32, 51]          64
│    └─ReLU: 2-5                              [16, 32, 32, 51]          --
│    └─Conv2d: 2-6                            [16, 64, 16, 26]          18,496
│    └─BatchNorm2d: 2-7                       [16, 64, 16, 26]          128
│    └─ReLU: 2-8                              [16, 64, 16, 26]          --
├─Linear: 1-4                                 [16, 416, 256]            16,640
├─TransformerEncoder: 1-5                     [16, 417, 256]            --
│    └─ModuleList: 2-9                        --                        --
│    │    └─TransformerEncoderLayer: 3-1      [16, 417, 256]            527,104
│    │    └─TransformerEncoderLayer: 3-2      [16, 417, 256]            527,104
│    │    └─TransformerEncoderLayer: 3-3      [16, 417, 256]            527,104
│    │    └─TransformerEncoderLayer: 3-4      [16, 417, 256]            527,104
├─Linear: 1-6                                 [16, 12]                  3,084
==========================================================================================
Total params: 2,147,404
Trainable params: 2,147,404
Non-trainable params: 0
Total mult-adds (M): 148.68
==========================================================================================
Input size (MB): 1.02
Forward/backward pass size (MB): 307.10
Params size (MB): 4.38
Estimated Total Size (MB): 312.51
==========================================================================================
```

Figure 6: Architecture of a strided transformer model for modified classes.

4.1.5 Results for original classes

We evaluated the performance of the strided convolutional architecture described in the previous subsection on the original classes. The experiments focused on optimizing the training configuration to achieve reliable and generalizable results. Table 1 summarizes the outcomes of key training runs.

Table 1: Summary of performance results for the transformer models based on the the strided convolutional architecture for original classes.

no.	**optimizer**	**learning rate**	**epochs**	**accuracy**		**train time**
				train	**test**	
1	AdamW	0.000005	10	87.62	83.88	4h 25m
2	Adam	0.001	9	3.44	3.53	4h 50m
3	AdamW	0.000005	20	93.04	87.60	14h 17m

In the first experiment, we trained the model using the AdamW optimizer with a learning rate of 5×10^{-6} for 10 epochs. The results were encouraging, with training accuracy increasing consistently from 27.96% in the first epoch to 87.62% by the tenth. Correspondingly, test accuracy improved steadily, reaching 83.88% by the final epoch. This suggested that the model was able to learn meaningful representations and generalize well to unseen data.

Subsequently, we tested the same model using the Adam optimizer with a learning rate of 10^{-3}. This configuration failed to produce meaningful learn-

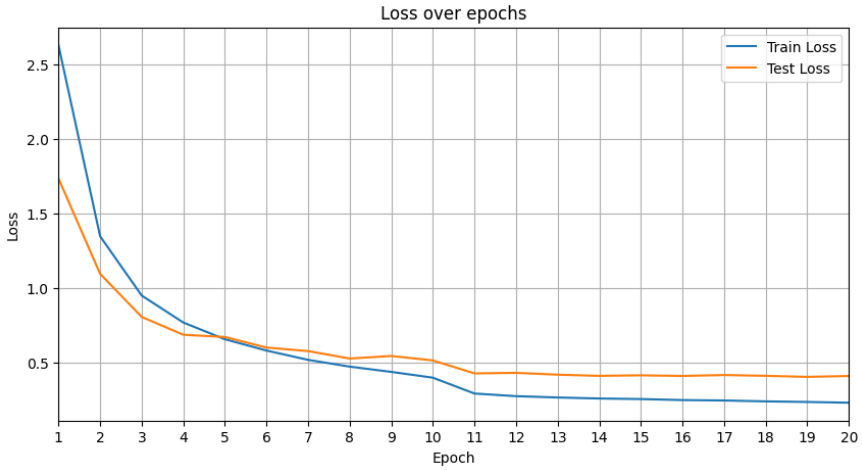
ing. Training and test accuracies stagnated around 3.5%, indicating that the model was unable to escape poor initial states or converge effectively. Therefore, training was halted early after 9 epochs.

These results confirmed that AdamW was significantly more effective for this architecture and task. Encouraged by the previous success, we conducted a longer training session with AdamW for 20 epochs to examine whether additional training would yield further improvements.

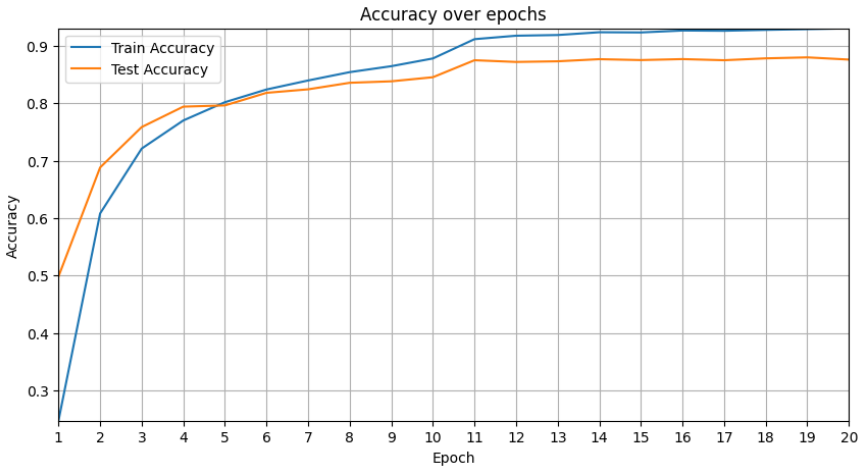
Initially, both training and test accuracies improved substantially, but after the 10th epoch, the training accuracy continued to rise while test accuracy plateaued around 87–88%. This indicated the onset of overfitting. Despite this, the final model achieved 93.04% training accuracy and 87.60% test accuracy, which remains our best-performing configuration for this task.

For this best model, we present accuracy and loss curves in Figure 7. Additionally, we include confusion matrices for both training and test datasets in Figure 8. Due to the large number of classes (31), confusion matrix values are normalized, scaled by 100, and rounded to integers for readability.

The confusion matrices reveal several interesting patterns. The *silence* class is recognized with 99% accuracy across both training and test sets, indicating that the model has effectively learned to detect non-speech input. However, phonetically similar words such as *go* and *no*, or *tree* and *three*, are frequently confused. This is an expected outcome, given the auditory similarities between such terms. These observations point toward potential improvements in data augmentation or model refinement to better handle such close distinctions.

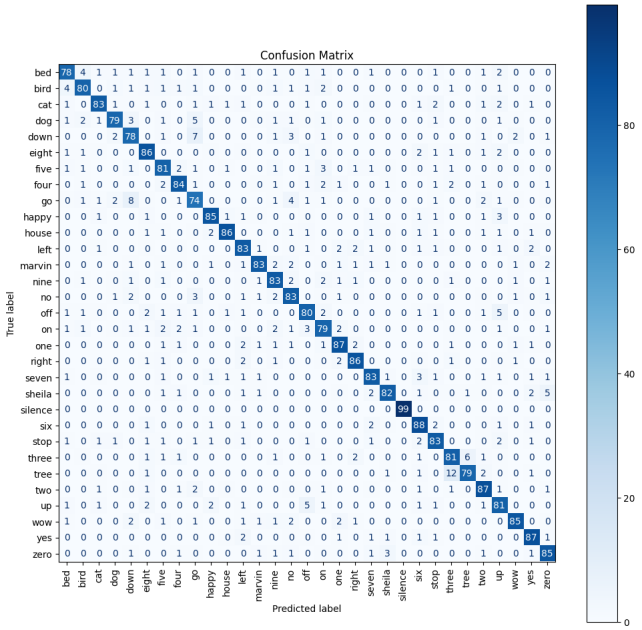


(a) Loss over time.

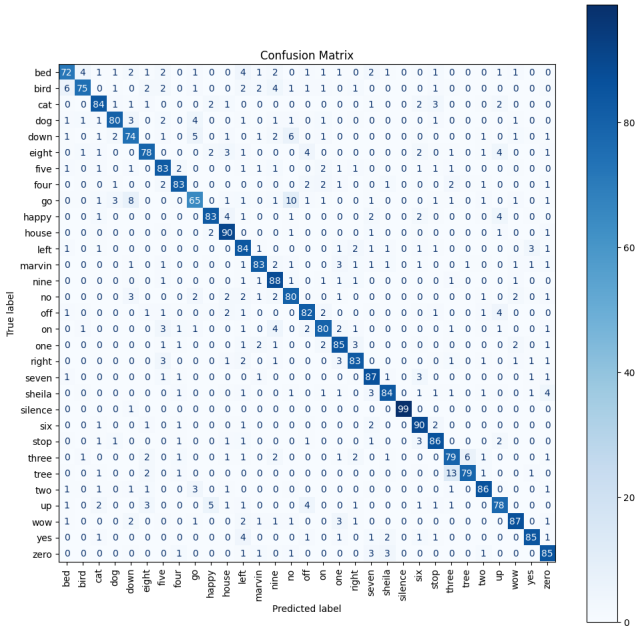


(b) Accuracy over time.

Figure 7: Loss and accuracy over time for the best strided transformer model for original classes with AdamW optimizer trained for 20 epochs.



(a) Train dataset



(b) Test dataset

Figure 8: Confusion matrices for the best strided transformer model for original classes with AdamW optimizer trained for 20 epochs.

4.1.6 Results for modified classes

In this series of experiments, we evaluated model performance on the dataset with modified classes. The same architecture and training procedures were used as in the original classification setting. Summary of the results is provided below in Table 2.

Table 2: Summary of performance results for the transformer models based on the the strided convolutional architecture for modified classes.

no.	**optimizer**	**learning rate**	**epochs**	**scheduler**	**pos embed**	**accuracy**		**train time**
						train	**test**	
1	AdamW	0.000005	10	StepLR	no	69.88	69.67	4h 6m
2	Adam	0.001	20	StepLR	no	37.33	60.71	1h 30m
3	AdamW	0.000005	20	StepLR	no	85.23	83.04	4h 50m
4	AdamW	0.000005	20	-	no	84.75	77.84	5h 3m
5	AdamW	0.000005	20	StepLR	yes	82.24	80.91	6h 54m
6	AdamW	0.000005	20	CosineAnnealingLR	no	85.83	82.11	14h 16m

The model trained with the Adam optimizer once again underperformed compared to AdamW. Although it managed to improve test accuracy to 60.71%, the training was unstable and the training accuracy stagnated at 37.33% before early stopping was triggered at epoch 6. This suggests that the optimizer struggles to generalize in this setting as well.

In contrast, AdamW yielded significantly better results. With the same learning rate and training for 20 epochs using a StepLR scheduler, the model achieved a peak test accuracy of 83.04%, closely matching its training accuracy of 85.23%.

Training without a scheduler resulted in more erratic accuracy trends and somewhat reduced test accuracy (77.84%), highlighting the importance of proper learning rate control. This instability is evident in the performance fluctuations throughout training despite gradual accuracy improvement.

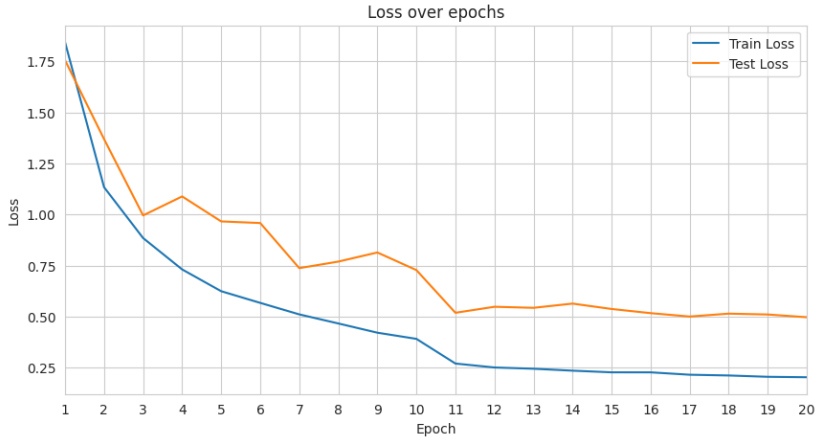
Adding positional embeddings slightly reduced the test accuracy (80.91%) compared to the baseline without positional embeddings. This suggests that positional information may not be necessary or even beneficial for this particular sppech recognition task, possibly due to the relatively short and fixed-length input sequences.

The use of a CosineAnnealingLR scheduler led to more stable training but did not surpass the performance achieved with StepLR. The model trained under this regime achieved a best test accuracy of 82.11%, showing that while the learning rate schedule smoothened training, it did not offer a distinct advantage over StepLR in terms of final accuracy.

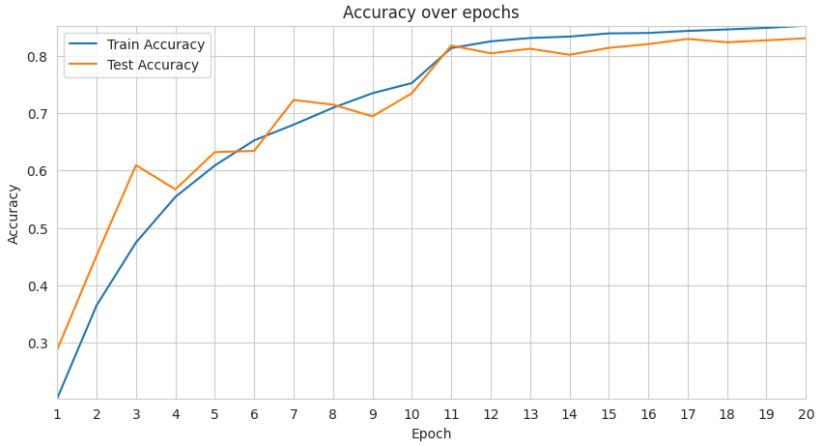
Overall, the best-performing model configuration was identical to that used in the original multi-class setup: the AdamW optimizer with a low learning rate of 5e-6, trained for 20 epochs with a StepLR scheduler, and without positional embeddings. This setup achieved 83.04% test accuracy and is used for further

analysis in this report, with training and testing curves and confusion matrices provided in the Figures 9 and 10 below.

A deeper look into the confusion matrices revealed that the *silence* class is predicted with nearly perfect accuracy for both the training and testing sets. The *unknown* class, however, remains the most difficult to classify, frequently being confused with multiple other classes. This is likely due to its heterogeneous nature, as it aggregates various distinct words, making it less internally consistent. Other notable confusion trends include frequent misclassification of the command *go* as *down* or *no*, likely due to acoustic similarities.

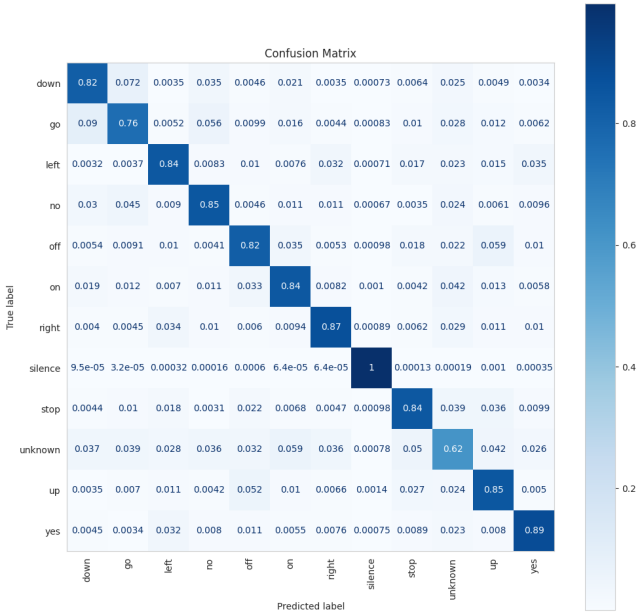


(a) Loss over time.

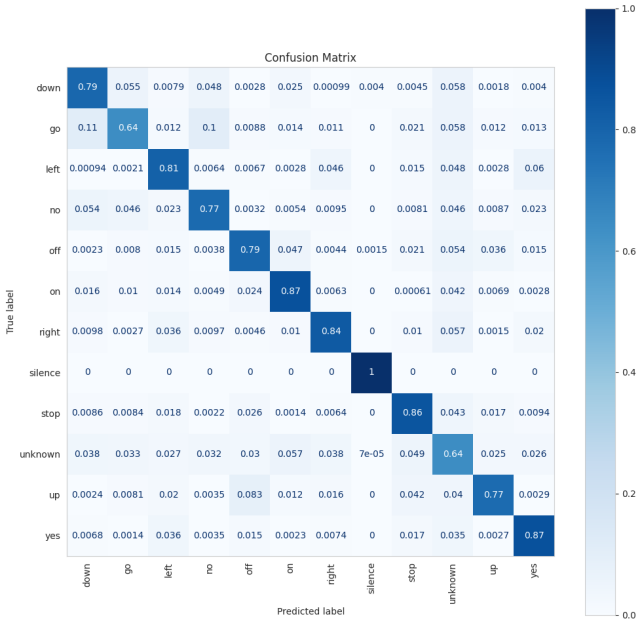


(b) Accuracy over time.

Figure 9: Loss and accuracy over time for the best strided transformer model for modified classes with AdamW optimizer, STEPLR scheduler and no position embedding trained for 20 epochs.



(a) Train dataset



(b) Test dataset

Figure 10: Confusion matrices for the best strided transformer model for modified classes with AdamW optimizer, STEPLR scheduler and no position embedding trained for 20 epochs.

4.1.7 Different architecture

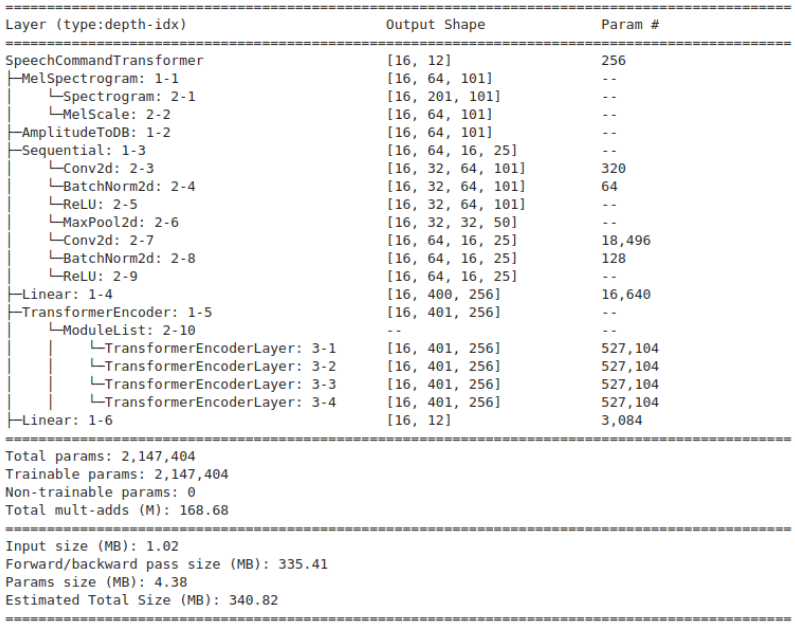
A further variation of the improved model involved modifying the convolutional frontend to incorporate a max pooling layer. In this version, the first convolutional layer uses a stride of 1, while the second convolutional layer uses a stride of 2 to perform spatial downsampling. Additionally, a `MaxPool2d` layer with a kernel size and stride of 2 is inserted between the two convolutions. This results in two stages of spatial resolution reduction: one through pooling and one through stride.

The output of the CNN in this variant has shape $(64, 16, 25)$, producing a flattened sequence of length 400 before the projection step. After projection to a 256-dimensional embedding and addition of the `[CLS]` token, the sequence fed into the Transformer encoder has length 401. This architecture maintains the same Transformer configuration as in previous experiments (modified classes), with 4 layers, 4 attention heads, and a feedforward width of 512.

The overall parameter count remains consistent at approximately 2.15 million, while the number of multiply-add operations is slightly higher than in the stride-based downsampling variant but still significantly lower than in the original model. The total memory usage also remains manageable at around 340 MB. Details of this architectures are depicted in Figure 11.

Model summary:

- Total parameters: **2,147,404**
- Input size (batch size 16): **1.02 MB**
- Forward/backward pass memory: **335.41 MB**
- Estimated total memory usage: **340.82 MB**
- Total multiply-add operations: **168.68 MegaOps**



```
==========================================================================================
Layer (type:depth-idx)                        Output Shape              Param #
==========================================================================================
SpeechCommandTransformer                      [16, 12]                  256
├─MelSpectrogram: 1-1                         [16, 64, 101]             --
│    └─Spectrogram: 2-1                       [16, 201, 101]            --
│    └─MelScale: 2-2                          [16, 64, 101]             --
├─AmplitudeToDB: 1-2                          [16, 64, 101]             --
├─Sequential: 1-3                             [16, 64, 16, 25]          --
│    └─Conv2d: 2-3                            [16, 32, 64, 101]         320
│    └─BatchNorm2d: 2-4                       [16, 32, 64, 101]         64
│    └─ReLU: 2-5                              [16, 32, 64, 101]         --
│    └─MaxPool2d: 2-6                         [16, 32, 32, 50]          --
│    └─Conv2d: 2-7                            [16, 64, 16, 25]          18,496
│    └─BatchNorm2d: 2-8                       [16, 64, 16, 25]          128
│    └─ReLU: 2-9                              [16, 64, 16, 25]          --
├─Linear: 1-4                                 [16, 400, 256]            16,640
├─TransformerEncoder: 1-5                     [16, 401, 256]            --
│    └─ModuleList: 2-10                       --                        --
│    │    └─TransformerEncoderLayer: 3-1      [16, 401, 256]            527,104
│    │    └─TransformerEncoderLayer: 3-2      [16, 401, 256]            527,104
│    │    └─TransformerEncoderLayer: 3-3      [16, 401, 256]            527,104
│    │    └─TransformerEncoderLayer: 3-4      [16, 401, 256]            527,104
├─Linear: 1-6                                 [16, 12]                  3,084
==========================================================================================
Total params: 2,147,404
Trainable params: 2,147,404
Non-trainable params: 0
Total mult-adds (M): 168.68
==========================================================================================
Input size (MB): 1.02
Forward/backward pass size (MB): 335.41
Params size (MB): 4.38
Estimated Total Size (MB): 340.82
==========================================================================================
```

Figure 11: Architecture of a strided transformer model with max pooling for modified classes.

Due to time constraints and the high training time of each configuration, we were not able to test additional variations as originally planned. Specifically, we intended to evaluate a three-layer convolutional frontend as well as an alternative pooling configuration where max pooling would be applied after the second convolutional layer rather than the first. These could potentially improve performance by either increasing representational capacity or better aligning temporal resolution with Transformer input requirements.

Although not all architectural variants could be evaluated within the scope of this project, the tested configurations offer valuable insights into the trade-offs between efficiency, model depth, and sequence dimensionality.

4.1.8 Results for different architecture

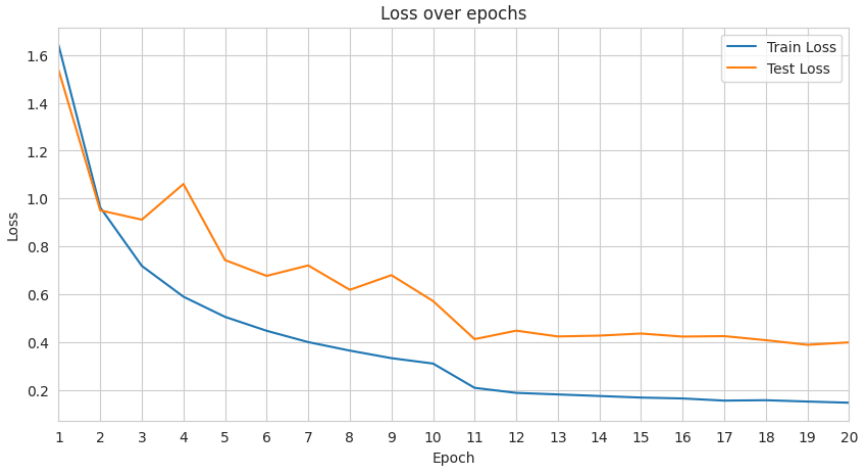
For this experiment, we use modified the convolutional neural network architecture to use a stride of 1 in the first convolutional layer and a stride of 2 in the second convolutional layer with a max pooling layer between them, as described in the previous section. This change was motivated by the desire to preserve more fine-grained spatial information early in the network, which is especially beneficial when working with audio spectrograms, where early patterns may carry important temporal and frequency cues.

The training and test performance across 20 epochs is illustrated in Figure 12, along with confusion matrices in Figure 13. As shown, the model exhibits

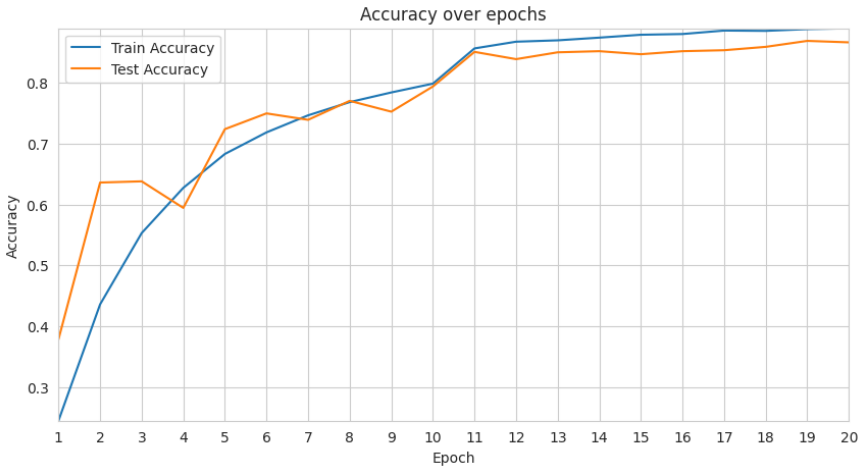
rapid performance gains in the first five epochs, followed by a more gradual but stable increase in both training and test accuracy. Notably, the test accuracy surpasses 85% by epoch 11 and peaks at 86.67% by epoch 20, with a corresponding training accuracy of 88.98%. These results represent the best performance achieved among all tested configurations.

Compared to previous architectures, this model not only achieved higher final accuracy but also demonstrated improved training stability, as seen in the smooth convergence of the training loss. The test accuracy curve, although somewhat chaotic in the initial epochs, stabilizes significantly after epoch 10. Detailed confusion matrices confirm that the model performs exceptionally well on the *silence* class, with near-perfect predictions. The *unknown* class remains the most challenging, though it shows a noticeable improvement in accuracy from 0.64 to 0.71 when compared with the previous experiment.

This experiment demonstrates the critical importance of early feature preservation in convolutional architectures for audio classification. The use of stride 1 in the first convolutional layer, in conjunction with a max pooling operation, appears to balance feature retention and dimensionality reduction effectively.

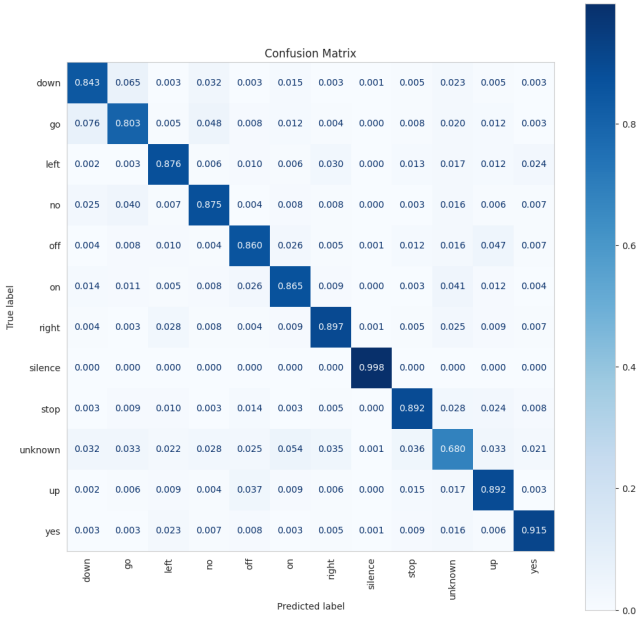


(a) Loss over time.

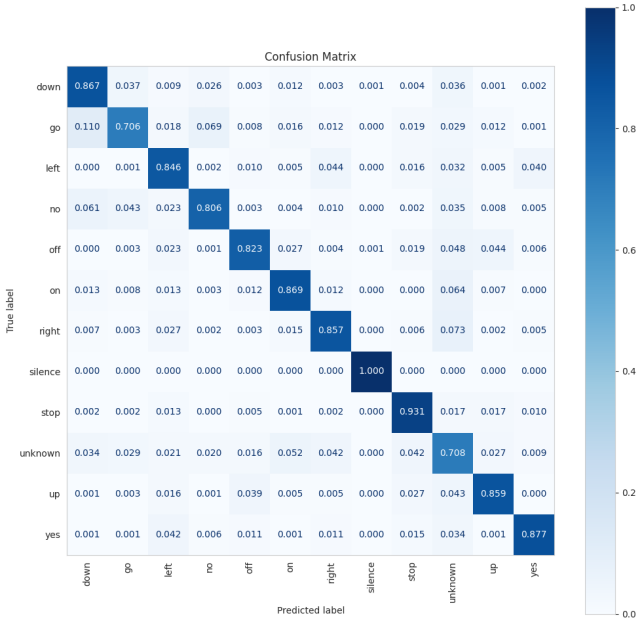


(b) Accuracy over time.

Figure 12: Loss and accuracy over time for the transformer model with a different architecture (stride=1 for first convolutional layer, stride=2 for second convolutional layer and max pooling between them) for modified classes trained for 20 epochs.



(a) Train dataset



(b) Test dataset

Figure 13: Confusion matrices for the transformer model with a different architecture (stride=1 for first convolutional layer, stride=2 for second convolutional layer and max pooling between them) for modified classes trained for 20 epochs.

4.2 Other models

We also tried other approaches to deal with the recognition of silence and the unknown. The method we researched the most is GRU. A gated Recurrent Unit is a type of recurrent neural network (RNN) designed to model sequential data. It was introduced as a simpler alternative to the more complex LSTM (Long Short-Term Memory) networks.

GRUs use gating mechanisms—specifically the update gate and the reset gate to control the flow of information and manage the hidden state. This helps them capture long-range dependencies in sequences while being computationally more efficient than LSTMs. GRUs are particularly useful in tasks like speech recognition, language modelling, and time-series prediction, and this is why we decided to use them.

4.2.1 Simple GRU model on 30 classes

The first model we tried proved that there is potential in this approach. It scored over 82% of accuracy on 30 classes. Truth be told, we were surprised that the accuracy was so high.

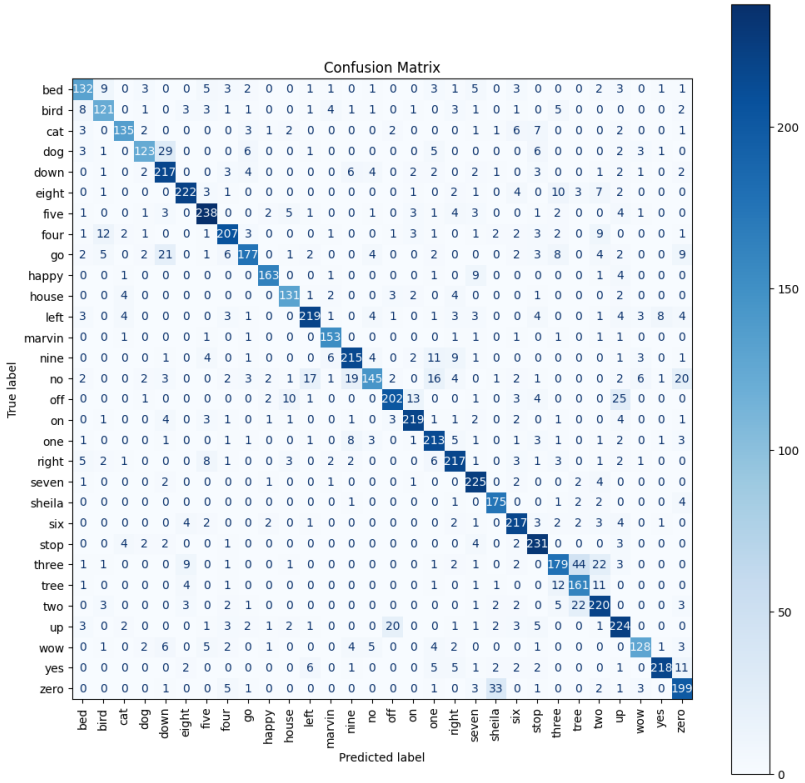


Figure 14: Convolution matrix for the first model on original classes

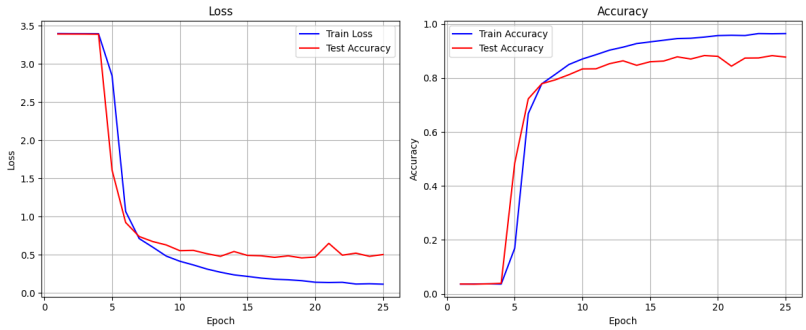


Figure 15: Loss and Accuracy for epochs for the first model on original classes

This model was simple compared to transformers, and training was almost instant. Here is the architecture of this model:

```
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
SpeechCommandGRU                         [32, 30]                  --
├─Sequential: 1-1                        [32, 40, 39]              --
│    └─Conv1d: 2-1                       [32, 32, 39]              2,592
│    └─ReLU: 2-2                         [32, 32, 39]              --
│    └─Conv1d: 2-3                       [32, 40, 39]              3,880
│    └─ReLU: 2-4                         [32, 40, 39]              --
├─GRU: 1-2                               [32, 39, 256]             427,008
├─Linear: 1-3                            [32, 30]                  7,710
==========================================================================================
Total params: 441,190
Trainable params: 441,190
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 541.23
==========================================================================================
Input size (MB): 0.20
Forward/backward pass size (MB): 3.28
Params size (MB): 1.76
Estimated Total Size (MB): 5.25
```

Figure 16: Architecture of initial GRU model

4.2.2 Simple GRU model on 12 classes

After that we switched to proper assessment and we considered only 12 classes including silence and unknown. For this setup we got 93.6%. Here is confusion matrix for this experiment.

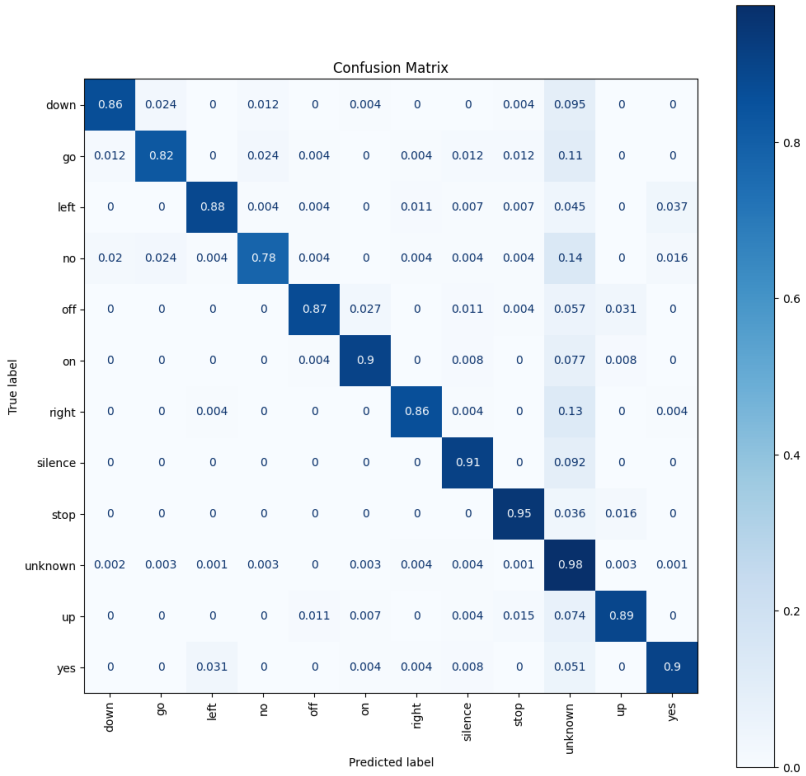


Figure 17: Confusion matrix for original model on modified classes

We tested different optimizers like AdamW and SGD and weight decay, but they didn't make a significant difference, and even in the case of SGD, they did not converge. We added then batch normalization and dropout. This model achieved 95.36%, almost a 2% improvement, which is huge.

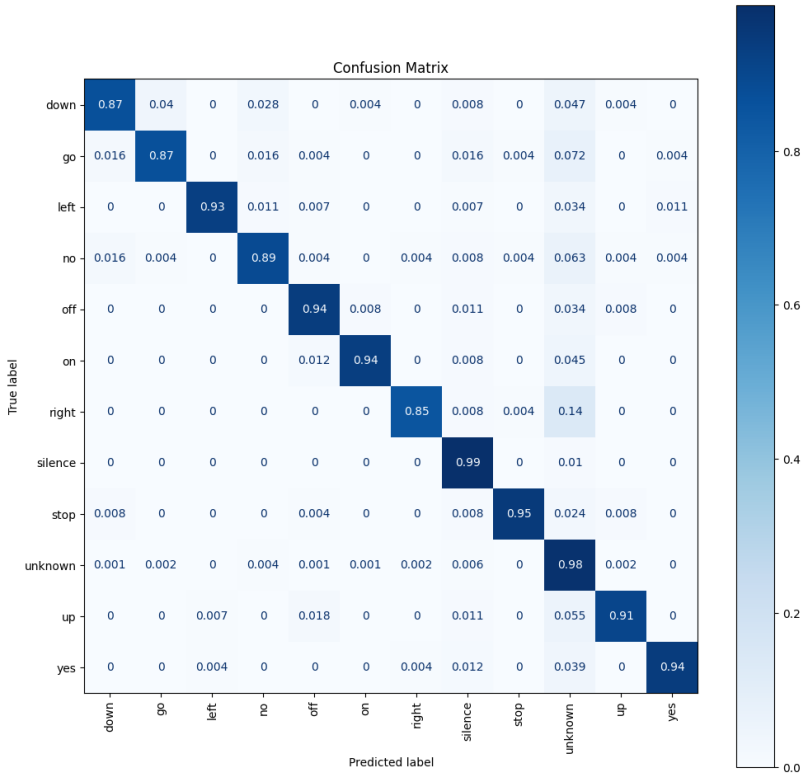


Figure 18: Confusion matrix for a model with added batch normalization and dropout

4.2.3 GRU with MFCC

After that, we switched gears and decided to look for a different approach. Instead of using raw sound waves, we switched to MFCC. To make it work, we had to change the data set. Because our input changed dimensionality, we adjusted the architecture. Now, it featured 2D conv layers. This model was huge in terms of parameter count: 4.5 mld. Surprisingly, it trained really fast, and thanks to this, we were able to quickly iterate and improve our solution. It was a more pleasant experience than working with transformers.

```
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
SpeechCommandCRNN                        [2, 12]                   --
├─Sequential: 1-1                        [2, 128, 40, 101]         --
│    └─Conv2d: 2-1                       [2, 32, 40, 101]          320
│    └─BatchNorm2d: 2-2                  [2, 32, 40, 101]          64
│    └─ReLU: 2-3                         [2, 32, 40, 101]          --
│    └─Conv2d: 2-4                       [2, 64, 40, 101]          18,496
│    └─BatchNorm2d: 2-5                  [2, 64, 40, 101]          128
│    └─ReLU: 2-6                         [2, 64, 40, 101]          --
│    └─Conv2d: 2-7                       [2, 128, 40, 101]         73,856
│    └─BatchNorm2d: 2-8                  [2, 128, 40, 101]         256
│    └─ReLU: 2-9                         [2, 128, 40, 101]         --
├─GRU: 1-2                               [2, 101, 256]             4,328,448
├─Dropout: 1-3                           [2, 256]                  --
├─Linear: 1-4                            [2, 12]                   3,084
==========================================================================================
Total params: 4,424,652
Trainable params: 4,424,652
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 1.62
==========================================================================================
Input size (MB): 0.03
Forward/backward pass size (MB): 29.37
Params size (MB): 17.70
Estimated Total Size (MB): 47.10
==========================================================================================
```

Figure 19: Architecture of model that uses MFCC

Again, we saw improvement; now, the model achieved 96.93%. Many changes contributed to this, such as the usage of scheduler CosineAnnealingLR and AdamW. Here is the confusion matrix for this try:

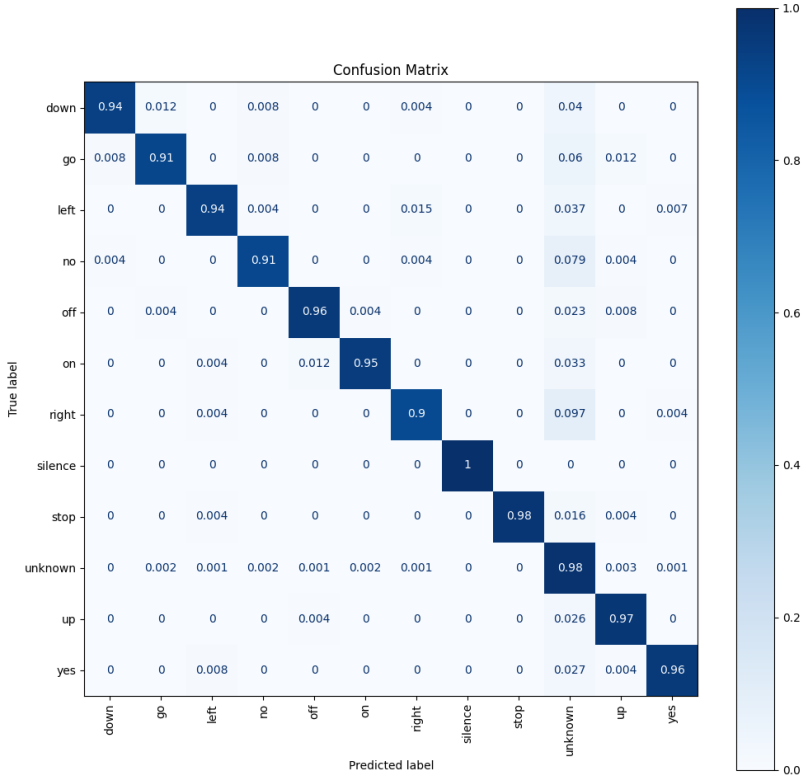


Figure 20: Confusion matrix of model that uses MFCC

4.2.4 Improved GRU with MFCC

The best model we got had 98.02% on the test set and 97.62% on the validation set. To achieve this, we implemented masking, which covered parts of MFCC plots. This resulted in decreased performance on the train test, but the model became more robust. Architecture is way more complex than it was in the beginning. We replaced ReLU with SiLU, added an attention layer and made the linear layer deeper. This model had 1.5 mln parameters. Here is the architecture of this model:

```
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
SpeechCommandCRNN_v2                     [2, 12]                   --
├─Sequential: 1-1                        [2, 128, 10, 25]          --
│    └─Conv2d: 2-1                       [2, 32, 40, 101]          320
│    └─BatchNorm2d: 2-2                  [2, 32, 40, 101]          64
│    └─SiLU: 2-3                         [2, 32, 40, 101]          --
│    └─MaxPool2d: 2-4                    [2, 32, 20, 50]           --
│    └─Conv2d: 2-5                       [2, 64, 20, 50]           18,496
│    └─BatchNorm2d: 2-6                  [2, 64, 20, 50]           128
│    └─SiLU: 2-7                         [2, 64, 20, 50]           --
│    └─MaxPool2d: 2-8                    [2, 64, 10, 25]           --
│    └─Conv2d: 2-9                       [2, 128, 10, 25]          73,856
│    └─BatchNorm2d: 2-10                 [2, 128, 10, 25]          256
│    └─SiLU: 2-11                        [2, 128, 10, 25]          --
├─GRU: 1-2                               [2, 25, 256]              1,379,328
├─Attention: 1-3                         [2, 256]                  --
│    └─Linear: 2-12                      [2, 25, 1]                257
├─Sequential: 1-4                        [2, 12]                   --
│    └─Linear: 2-13                      [2, 128]                  32,896
│    └─ReLU: 2-14                        [2, 128]                  --
│    └─Dropout: 2-15                     [2, 128]                  --
│    └─Linear: 2-16                      [2, 12]                   1,548
==========================================================================================
Total params: 1,507,149
Trainable params: 1,507,149
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 145.54
==========================================================================================
Input size (MB): 0.03
Forward/backward pass size (MB): 7.31
Params size (MB): 6.03
Estimated Total Size (MB): 13.37
```

Figure 21: Architecture of best GRU model

Here is the confusion matrix for this model; we achieved a perfect score for silence and really high results for the unknown classes. The other classes confused was "right", which is similar to "night" from the unknown class, "go" and "up", both of which are short words and proved to be harder than longer ones. The lowest accuracy was on the "down" command, which is very surprising for us; it was mistaken with "go", "no", and "unknown" classes.

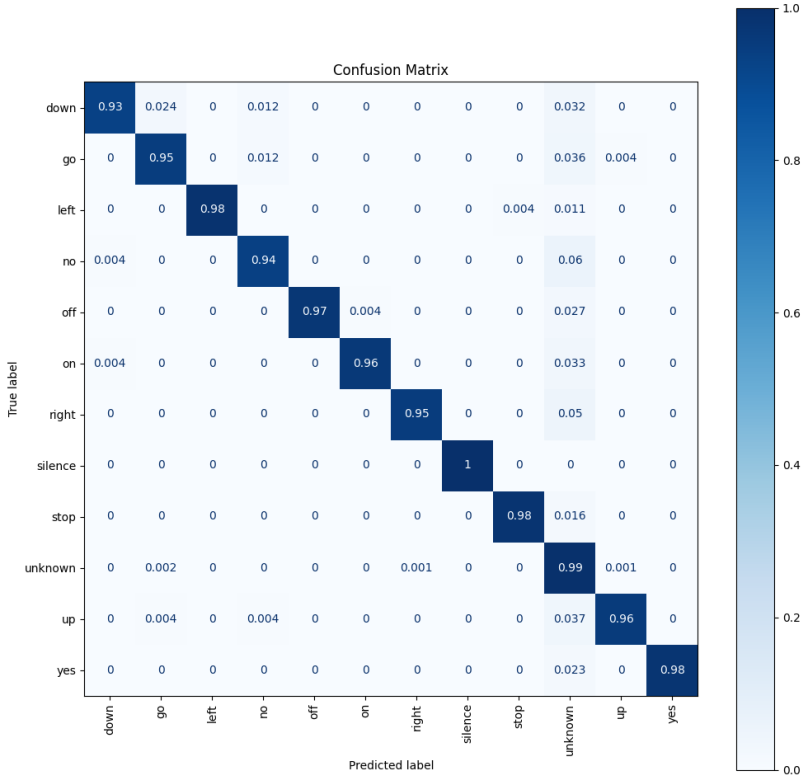


Figure 22: Confusion matrix of best GRU model

Table 3: Best GRU model on modified classes (performance over 10 runs)

Set	**Accuracy**		**Loss**	
	Mean	**Std**	**Mean**	**Std**
Train	95.42%	0.09%	0.1352	0.0026
Test	98.02%	0.08%	0.0721	0.0021
Validation	**97.62%**	0.07%	0.0892	0.0046

This journey with GRU was a nice experience; this family of models was easier to work with. We didn't see out-of-memory errors as frequently, and training was faster, which allowed us to gain intuitions quicker and obtain satisfactory results. We also think that those models were more forgiving when it comes to the selection of hyper-parameters. If we had more time, we think that we would be able to create an even better model. Of course, we did not describe every model; the solution was created iteratively, and in the report, we only focused on "snapshots". Truth be told, a lot of that work was "lost" because of

Jupyter Notebooks' nature with rerunning of cells; we did interactively change parameters to see the results. But in the end, we tried to clean them and tell a story of our research and findings.

5 Summary

We really enjoyed the project; we learned much and built on foundations created in the first project. We strengthened our understanding of the PyTorch framework and explored various aspects of speech recognition using transformers and other methods like GRU. While we didn't manage to implement all the ideas, we created a few models that performed really well, and we are happy with the results we obtained. Even though we were upset at the beginning when our models did not converge, we persevered, and in the end, we built something that we are proud of; on top of that, we learned a lot about transformers and speech detection. That being said, we see room for improvement in our solution; we could test on "real" test data, and we could submit our model for an official assessment. We had ideas about data augmentation using background noise and many more. While this project concludes here, it has certainly sparked a continued interest in the field, and we want to research it further in our free time.

References

[1] H. Purwins *et al.*, "Deep learning for audio signal processing," *IEEE Journal of Selected Topics in Signal Processing*, 2019.

[2] S. Hershey *et al.*, "Cnn architectures for large-scale audio classification," *ICASSP*, 2017.

[3] S. B. Davis and P. Mermelstein, "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences," in *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 357–360, IEEE, 1980.

[4] A. Vaswani *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[5] Y. Gong, Y.-A. Chung, and J. Glass, "Ast: Audio spectrogram transformer," *Interspeech*, 2021.

[6] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv preprint arXiv:1603.07285*, 2016.