NTNU

Kunnskap for en bedre verden

Department of Computer Science

TDT4173 - Assignment 3

Trigger word detection using Deep Neural Networks

Group:
Group 16

Authors:
Anders Bennæs (andebenn)
Simen Strømstad (siments)
Patrik Kjærran (patrikkj)

November 27, 2020

Abstract

The purpose of this project is to create three fully functional trigger word detectors and investigate how clever feature extraction can affect a model's predictive capability. Trigger word detectors should have good classification performance, low computational footprint and be robust to false triggers.

Three deep neural networks were implemented for binary classification; two recurrent- and one convolutional neural network. Raw audio is fed as input to one of the RNNs. This architecture serves as a baseline to evaluate the effects of preprocessing in the two other models, which both take preprocessed audio data in the form of a Log-Mel spectrogram as input. After all the models had been fully implemented and evaluated against objective metrics, they were empirically tested to address the real-world performance.

We found that the RNN taking Log-Mel spectrograms as input is superior to both the other models with regards to the task of trigger word detection. It has better accuracy and precision, a relatively low computational footprint and is less prone to false triggers. Since the baseline architecture performed substantially lower than both the other models, while at the same time having a higher computational footprint, we have concluded that the preprocessing pipeline had a positive effect on performance.

Keywords: Trigger word detection, RNN, CNN, Audio preprocessing, Log-Mel spectrogram

Link to GitHub repository: `https://github.com/patrikkj/marvin-models`

Link to video: `https://www.youtube.com/playlist?list=PLBPayIBBGNg9Rd0ENXhlUS_1vhBn3h4r2`

Table of Contents

List of Figures

List of Tables

1 Introduction

The idea behind this project originates from one of the authors asking *Siri* what the weather was going to be like. He found it fascinating how his phone is always listening passively until triggered, and started contemplating how the underlying technology works. Our way of interacting with computers has changed drastically in recent years, with the computer mouse and keyboard just recently being supplemented by touch screen technology for user input. Today, the use of voice commands to control computers and embedded electronic devices are increasing in popularity, and *trigger word detection* is a prerequisite for such systems.

This project addresses the task of detecting trigger words from a stream of audio data. A trigger word detection model aims to identify when a single word, or a certain phrase, is uttered. The goal of the project is to train a lightweight, offline, trigger word detection model that is capable of detecting the word “Marvin”. The final model is deployed to be used in applications, running in the background as a service on a computer or as a client-side web service. In contrast to full-scale voice assistant systems, there will only be a trigger word detector without further explicit interaction with the user.

The data used when training the different models is collected from a public dataset of speech commands utterances. This project alters the perspective of speech commands, as the chosen trigger word ”Marvin” is considered a negative sample in the original dataset. As the training data is labeled, this is to be considered a supervised learning approach. Further, it can be viewed as a binary classification task. This is because the trigger word detector fundamentally tries to label segments from a continuous audio stream as either *positive* or *negative*, indicating whether the segment contains the word ”Marvin”.

As indicated in section 2, recent research suggests that Deep Neural Networks (DNNs) are superior to traditional statistical models for the complex problem of recognizing speech. Because of this, the task is approached using DNN architectures, which are capable of processing raw audio directly. However, due to the noisy and sparse nature of raw audio, we are going to investigate the hypothesis that measures can be made to extract relevant information from the data, thus making it easier for complex models to learn good mappings for the task at hand. In order to test the hypothesis, three different models will be implemented. These are to be compared according to a set of evaluation methods, along with empirical evaluation.

The first model is a Recurrent Neural Network (RNN) that takes raw audio data as input. This architecture serves as a baseline for the other models presented and is referred to as the *Naïve RNN* for the remainder of this paper. The remaining two models are both provided input data which is preprocessed in accordance to the pipeline presented in section 3.1. The pipeline can be summarized by raw audio being transformed into a Log-Mel spectrogram, an intermediate representation that encodes the audio as an image in the time and frequency domain. The first model to utilize this input representation has a recurrent architecture similar to the one of the Naïve model, and is hereby referred to as *RNN*. The final model is a Convolutional Neural Network (CNN). The latter two models are to be compared against the baseline in order to identify the effects of the implemented preprocessing measures. Iterative tuning of hyperparameters is conducted in order to maximize the performance of each model before comparison.

As trigger word detection is an extensively studied field, we do not aim to solve a new problem, but rather contribute to existing research. We are therefore applying existing methods to a know problem. However, after examining a wide range of speech datasets, we believe the task outlined above is both unique and within scope.

2 Related Work

Speech recognition is an area of study that has existed for a long time [Cohen and Gannot, 2008]. Early research focused on recognizing simple vowels or digits. This was first accomplished at the Bell Labs in 1952, with a system restricted to classifying the utterance of one single digit at a time. In the late 60s, continuous speech recognition was explored, and in the early 80s, Hidden Markov models (HMMs) were introduced as a powerful tool for speech processing. HMMs were the dominant technology until the start of the 21st century when Deep Learning proved to yield better results [Kumar et al., 2018]. Since then, RNNs incorporating LSTM layers has been a popular choice of method. Within the field of speech recognition, a new domain has emerged, namely, trigger word detection. The two tasks share many similarities, but the environments in which they operate are somewhat different. Suitable applications for applying trigger word detection often require the model to run offline on low-energy devices, which has only recently become relevant due to technological advancements [Kumar et al., 2018].

2.1 Small footprint keyword spotting

The work on small footprint keyword spotting from the paper of Chen et al. [2014] highlights the importance of having a trigger word detection model with low memory- and computational cost. This is mainly because such applications should run on always-on processors. They propose a system that challenges keyword-filler HMMs, which in 2014 was an established method for the problem. The research was done in cooperation with Google, and their system is a standard *feed-forward* fully connected neural network, that is capable of detecting the key-phrase ”okay Google”. This is accomplished with high accuracy, low latency and small footprint, even when the system runs in computationally constrained environments. Chen et al. [2014] state that their system achieved 45% improvement relative to competitive HMMs.

2.2 Voice assistants

Trigger word detection models are part of an ever-growing technological industry. Large companies such as Google, Apple, Microsoft and Amazon have all implemented voice assistants in segments of their product line [Hoy, 2018]. Apple was first out with Siri in 2011, and the others quickly followed in order to keep up with the competition. Voice assistants have developed over time, and today they are capable of doing everything from answering simple questions to conducting bank transfers [Singh et al., 2020]. A commonality among these systems is that their trigger word detection is implemented using DNNs [Singh et al., 2020]. In order to provide further specific research on trigger word detection, two papers originating from Apple's Machine Learning research division are presented.

2.2.1 ”Hey Siri”

The trigger word detector of Siri applies a DNN to encode short sequences of audio to one among twenty sound classes [Sigtia et al., 2018]. These are from the set where the phonemes of ”Hey Siri”, silence and common background noises are included. The acoustic model outputs a probability distribution over the twenty sound classes, which can be interpreted as the individual score of a short sequence of audio. The continuous flow of output from this model is provided to an HMM, which is used to calculate a total confidence score that is evaluated against a predefined threshold for classification. The calculation of a total score is a recurrent process, and it is based on log-probabilities over the individual scores. It is sensitive to the order of the perceived sound classes since the model only is supposed to trigger if the inputs are detected in the correct order. The window size for the sequences resulting in a total score is dynamic, meaning that the HMM adapts segment length in accordance with the input. If the current total score at some point goes above the threshold, the model signals that a trigger word has been detected and the rest of the application is woken up [Sigtia et al., 2018].

2.2.2 Importance of avoiding false triggering in voice assistants

Many trigger word detectors run locally, and the main application does not connect to larger speech processing servers until triggered [Jeon et al., 2020]. In the case of a false positive being detected, the model will trigger and private audio can be processed without the speakers knowing. This is a problem regarding privacy since the users can never be completely sure whether the recorded data is misused. Voice assistants are implemented in many embedded devices, and because of data privacy, the mitigation of false triggers is of utmost importance. However, having high precision, and thus avoiding false positives, should not compromise accuracy, since one still wants a system that triggers appropriately. In their research, Jeon et al. [2020] propose a Bidirectional Lattice RNN in order to address this problem.

3 Data

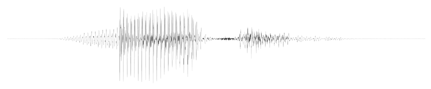
The data used in this project is from the *Speech Commands Dataset v.0.0.1* [Warden, 2018]. It contains 64 722 utterances from 1 881 speakers, of which 1 746 samples are labeled "Marvin". The rest represent 30 different words such as "House", "Sheila" and "Forward", thereby covering a wide range of phonemes. Data is split into train, validation and test sets by stratified sampling with ratios 60/20/20. The validation set is utilized in hyperparameter tuning, while the test set is used to evaluate the degree of generalization.

Due to limited computational resources and a highly disproportionate ratio of negative to positive samples, half of the negative samples are discarded by random selection. This is shown in Table 1, with corresponding ratios. To further balance the ratio of the minority class, the dataset which is fed to the model is generated at runtime by randomly interleaving positive and negative samples with a ratio of 1:9 (10%). This is equivalent to providing a *class weight* for the respective classes in the loss function, but has the added benefit of increasing the probability of a random batch containing a positive sample. An issue with traditional class weighting on unbalanced datasets is that some batches may end up with no positive samples for the model to learn from [Seiffert et al., 2008]. Note that resampling is only performed on the training split, as it is essential that the validation and test data come from the same probability distribution for their losses to be comparable when addressing the models' degree of generalization.

Table 1: Dataset characteristics.

Split		**Original**			**Cropped**			**Resampled**		
Name	Ratio	Positive	Negative	Ratio	Positive	Negative	Ratio	Positive	Negative	Ratio
Train	0.6	1048	37786	1:36	1048	18893	1:18	1994	17946	1:9
Validation	0.2	349	12595	1:36	349	6298	1:18	349	6298	1:18
Test	0.2	349	12595	1:36	349	6297	1:18	349	6298	1:18
Total	1	1746	62976	1:36	1746	31488	1:18	2692	30541	1:11

The training samples all represent one second of audio. Each contains 16-bit mono-channel raw PCM audio sampled at a rate of 16 kHz, encapsulated in a *.wav* file. The raw PCM values describe oscillation in air pressure at a given time. This can be plotted in two dimensions, resulting in a basic waveform. When loaded into memory, the data is normalized to a scale of -1 to 1. The waveform of a "Marvin" training sample is visualized in Figure 1a, while a contrasting example of the word "Sheila" is displayed in Figure 1b. Manually collected data for empirical testing is sampled by dividing a continuous audio stream into one-second intervals, still with a sample rate of 16 kHz. A *sliding window* method with a step length of 200 ms is used to ensure that every word is fully included in at least one audio clip.


(a) Example of "Marvin".

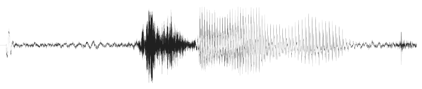

(b) Example of "Sheila".

Figure 1: Waveform of input.

Raw PCM audio is directly used as input to the Naïve RNN model. However, as this format has low information density, we have conducted a series of data preprocessing steps to extract the features of the raw PCM audio, while at the same time lowering input size. Reduction in input dimensionality is essential to achieve a low computational footprint. The resulting format is an image with a time and a frequency dimension, which can be used as input to the sophisticated RNN and CNN models.

3.1 Data preprocessing

Both the RNN and CNN implementations are used to classify images. Pictures of a waveform do not sufficiently differentiate features, hence, preprocessing has to be conducted. Below is the preprocessing pipeline that one second of audio undergoes before being used in the two classification models.

3.1.1 Spectrogram

After loading audio data into memory, the first preprocessing step is to convert the audio tensor into a spectrogram. A spectrogram is a visualization of the presence of specific frequencies at a given time, with higher amplitude giving brighter colors. It encodes patterns more clearly than a waveform, which makes it easier for models to learn a mapping from input to output [Wyse, 2017]. The conversion is done by computing a series of *Short-Time Fourier Transforms*[1] on the audio tensor. This is done to keep some of the temporal information, which a traditional Fourier Transform over the whole interval would eliminate.

[1]For the interested reader: https://ccrma.stanford.edu/~jos/sasp/Short_Time_Fourier_Transform.html

We have experimented with a wide range of Fourier transform configurations and arrived at a frame size of 512 and a step size of 256. This makes a nice compromise between time and frequency resolution while minimizing computational footprint [Amatriain et al., 2002]. A sample rate of 16 kHz results in 61 transformations for each input. Every transformation gives a function for the frequency strengths in the given time segment, which is represented as a single vertical strip. Horizontal concatenation of these 61 strips composes the full spectrogram. A visualization for an input of "Marvin" is displayed in Figure 2.

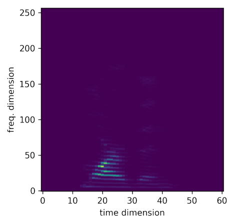


Figure 2: Spectrogram.

3.1.2 Mel spectrogram

From Figure 2, it is evident that most of the information is present in the lower frequency spectrum. This is because humans communicate within this range, and there is little to no information contained in higher frequencies [Madikeri, 2011]. In order to emphasize the area of relevant frequencies, the input is further processed by applying a Mel filter bank. When applied to the spectrogram, segments of increasing size are put into bins, summarizing the energy level of that segment. The segment size increases with higher frequencies, thus giving the lower ones a higher weight. This is also analogous to human communication, as small frequency variations are not distinguished - more so in higher frequencies. We used 64 Mel bins, with the result visualized in Figure 3.

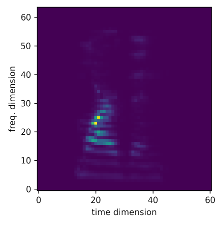


Figure 3: Mel spectrogram.

However, Mel spectrograms are not sufficiently detailed. One highly effective operation to handle this is *log transformations*. Sound level is not perceived on a linear scale, and transforming the energy levels is therefore more representative of human communication [Madikeri, 2011]. By computing the log transform of the Mel spectrogram, the features of the data become significantly clearer. This is visualized in Figure 4, which is the final format of input for both the RNN and CNN.

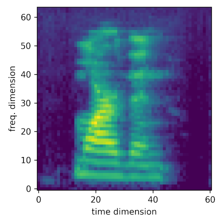


Figure 4: Log-Mel spectrogram.

3.2 Data augmentation

In addition to being preprocessed, the data is augmented at runtime when training the RNN and CNN. The augmentation pipeline is inspired by Google Brain's augmentation for speech recognition [Park et al., 2019]. It is performed by masking a randomly located fraction of size 1-10 in both the time- and frequency spectrum, visualized in Figure 5. This prevents the model from being too reliant on specific frequencies or timesteps while providing diversity to the input images.

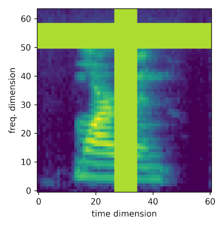


Figure 5: Masked random fraction of frequency and time spectrum.

3.3 Flow of data

The flow of data during training of the models is described in Figure 6. While the Naïve RNN takes a plain waveform as input, the RNN and CNN evaluate preprocessed data in the form of a Log-Mel spectrogram. The preprocessing and augmentation pipeline is built into the first two layers of the model[2], such that these computations can be performed at runtime on the GPU while the data is loaded into memory. The data augmentation layer is given a boolean flag by the model, indicating whether the model is training or evaluating, such that augmentation is disabled for non-training inferences. In the next section, the three specific models are explained in detail.

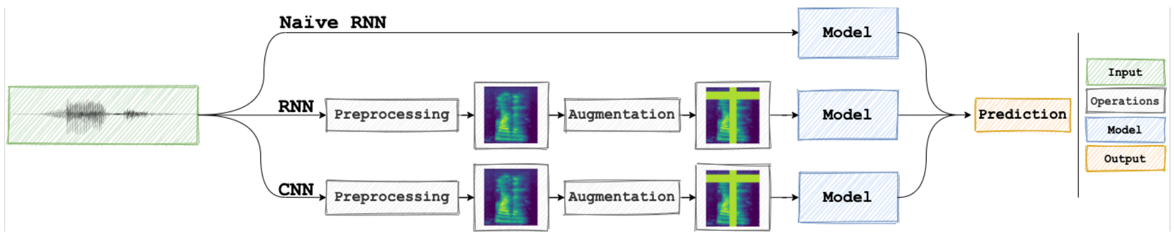


Figure 6: Flow of data during training.

[2]https://github.com/patrikkj/marvin-models/blob/main/scripts/layers.py

4 Methods

The methods selected for this project are two RNNs and a CNN. RNNs are suited for capturing patterns in sequential data [Goodfellow et al., 2016] and are chosen as we are working with audio. CNNs excels at processing data that takes the form of a grid-like pattern, and this architecture is therefore chosen as an alternative to interpret Log-Mel spectrograms. The models are all implemented in Python using the TensorFlow Keras API [Abadi et al., 2016]. First, we present commonalities in the workflow of all three models, before specific implementations are covered in detail.

4.1 Workflow

Training objective and optimization

The training objective for all three models has been to minimize *binary crossentropy loss*, calculated as the sum of negative log-probabilistic differences between labels and predictions. The loss J is formalized in Equation 1, where N denotes the number of elements in a batch, y_n the label for sample n and $\hat{y}_n$ its corresponding prediction:

$$J(y,\hat{y}) = -\frac{1}{N}\sum_{n=1}^{N}[y_n \cdot log(\hat{y}_n) + (1-y_n)\cdot log(1-\hat{y}_n)] \tag{1}$$

This appears to be a reasonable minimization objective, considering that the problem is modeled as a binary classification task. In order to minimize the loss function, we have experimented with three gradient-based optimizers, namely Adam, Stochastic Gradient Descent (without momentum) and RMSProp. Adam was selected from an early stage, as it provided the most stable results, is scale-invariant and we have found it to be more forgiving in terms of learning rate configurations. Most runs were executed with a learning rate of 0.001 and an exponential decay rate for first- and second-order momentum of 0.9 and 0.999 respectively, as suggested by Kingma and Ba [2019].

Output bias initialization

By default, the bias terms of dense layers in TensorFlow are initialized to zero. For this reason, the models are expected to predict uniformly among the two classes. As we know the ratio of positive to negative classes is 1:9 for the training set, the model will achieve a significantly lower initial loss if being configured to consider the class imbalance. To accomplish this, we set the output bias of the last dense layer to -2.197. This speeds up convergence as the model does not have to spend the first few epochs learning the class distribution bias. The calculations and further reasoning behind the bias initialization can be found in Appendix B.

Early stopping

To prevent overfitting on complex model configurations, we use early stopping with patience of 20 epochs, monitoring validation loss during training. This ensures that the model stops training once it starts overfitting, and the model weights are reverted to the best configuration if early stopping occurs.

Hyperparameters and training

We started off defining hyperparameter domains, inspired by common configurations for similar tasks [Sak et al., 2015, Krizhevsky et al., 2017]. Based on these domains, we conducted a broad random search on up to 6 GPUs, running in parallel on Google Colaboratory. Each instance continuously logged model configurations in addition to progression during training to remote cloud storage. Through the analysis of configuration performances, we have been able to progressively shrink the search space. This was achieved by eliminating specific values and excluding bad-performing combinations of hyperparameters, through imposing constraints on the sampling space. Further, good performing regions in the remaining parameter space were more densely sampled by repeating this process using a smaller, fine-grained search space. Finally, the best-performing model from the last iteration was selected for each of the architectures.

Random sampling was selected as it is a common approach, and proven by Bergstra and Bengio [2012] to be superior to an exhaustive grid search when the latter option is computationally infeasible. A significant drawback to both methods is the computational cost required. This also applies to other approaches for algorithmic hyperparameter tuning. However, as random sampling appears to be surprisingly effective despite its simplicity, this alternative was chosen.

A link to the notebooks containing the code for training and hyperparameter tuning of all three models can be found here.

4.2 Implementations

All operations are performed on batches utilizing vectorization, but for simplicity, the operations are described as if performed on a single sample. A link to the final models can be found here.

4.2.1 Baseline model - Naïve Recurrent Neural Network

The first model is a *naïve* approach to solving the problem using a RNN, where the raw PCM audio is fed as input to the model. First, the input tensor of length 16 000 is split into 16 equally sized chunks, referred to as t_i in Figure 7. These chunks are downscaled by a 1-dimensional convolution along the time axis. Given a stride of 4 and a kernel size of 24, the number of temporal components for each chunk is reduced from 1000 to 245. The output from the convolutions is further processed by a four-layer network of GRU cells, with 128 units per layer. Internal state is passed from one GRU cell to the next within the same layer, illustrated as the rightwards arrows in Figure 7. With the exception of the last recurrent layer, GRU cells are configured to also propagate their state as input to the corresponding cell within the next layer. Outputs across layers are visualized by arrows pointing upwards. In this manner, the last GRU cell in the 4th layer along the temporal axis encodes all information relevant for classification. This state is passed on to a single-unit dense layer with a sigmoid activation, outputting a predicted value between 0 and 1. The architecture can be categorized as a *many-to-one* RNN, as the model outputs a single prediction, formed by interpreting a sequence of inputs.

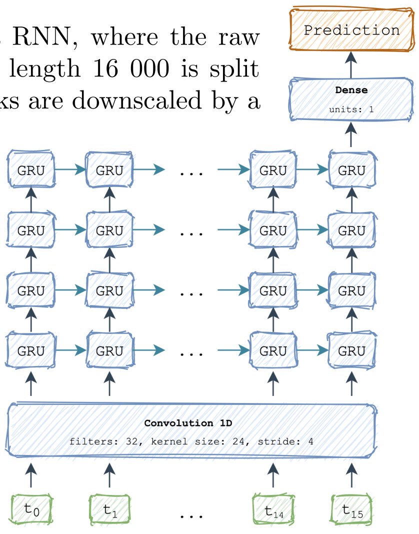


Figure 7: Naïve RNN architecture.

4.2.2 Recurrent Neural Network on Log-Mel Spectrograms

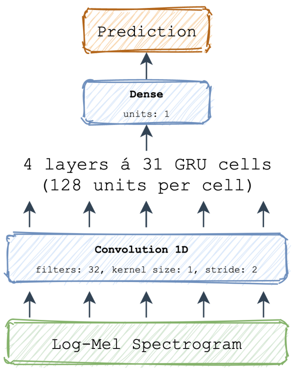


Figure 8: RNN architecture.

Using the Naïve model as a foundation, the other RNN utilizes a more compact data input representation in the form of the Log-Mel spectrograms outlined in section 3. Similarly to the Naïve RNN, a time-distributed, 1-dimensional convolution operation is performed. In this case, horizontal strips of the spectrogram are convoluted in isolation, as the kernel size along the temporal axis is 1. The entire frequency axis is collapsed by the convolution, extracting features across all frequencies for a time-step in the resulting 32 filters. Notice that the stride along the time dimension is 2, which in combination with a kernel of size 1 indicates that the convolution fully discards every other strip along this axis. Further, the convoluted output is passed on to a four-layer network of GRU cells, identical to the architecture described in section 4.2.1. The output of the recurrent layers is also passed to a single-unit dense layer, which ultimately outputs the prediction.

4.2.3 Convolutional Neural Network on Log-Mel Spectrograms

The last model is a CNN with two convolution blocks (Conv1D + Batch Normalization + MaxPooling) followed by a flattening layer and two dense layers interconnected by a dropout layer. The purpose of the two convolution layers is to extract increasingly abstract features from the input spectrogram. Both convolution layers use ReLU as an activation function, and are followed by Batch Normalization layers that aids in regularization. This normalization also eliminates the covariance shift caused by the outputs from ReLU not being centered around zero [Ide and Kurita, 2017]. The second convolution layer has 128 filters, as opposed to 64 for the first layer. This pattern of doubling the number of filters for each layer was chosen when experimenting with a different number of convolution layers. The Max Pooling layers use a pool size of 3, effectively shrinking the input with a factor of 3 as it only retains the sharpest features extracted by the convolution. This operation significantly lowers the computational footprint. For each sample, the flattening layer unfolds the two-dimensional output from the convolutional blocks to a one-dimensional tensor. A mapping from this tensor to the prediction of a value between 0 and 1 is learned by the dense layers, which are connected with a regularizing dropout layer. A pattern of halving the number of units for each layer was used when experimenting with a different number of dense layers.

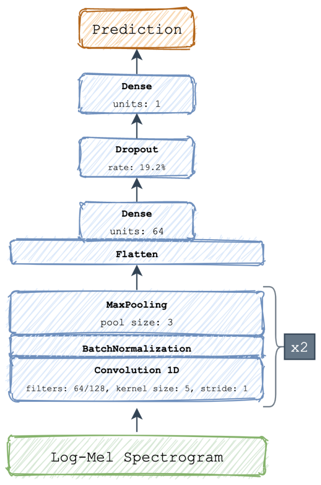


Figure 9: CNN architecture.

5 Results

We first evaluate the three models with respect to classification performance. Further, they are evaluated based on computational footprint and robustness to false triggers. The additional methods are chosen due to the mentioned aspects of a trigger word detector, namely that they often run on low-energy devices and are prone to data privacy regulations. All results are generated by the notebook found here.

5.1 Model configurations

Models were trained on the data described in section 3, using the architectures presented in section 4.2. The final model configurations were derived by applying the incremental tuning procedure outlined in section 4.1. Tuning was guided by looking at accuracy, precision and recall, in combination with the total number of model parameters and computational footprint for inference on batches of 1, 10 and 100 elements, both on CPU and GPU. All hyperparameters are summarized in Table 2. For ease of reproducibility, we have also included all other parameters and relevant configuration details for the three models in Appendix A.

Table 2: Overview of final hyperparameters.

Attribute	Naïve RNN	RNN
n_timesteps	16	—
conv_filters	32	32
conv_kernel_size	24	1
conv_stride	4	2
recurrent_cell	gru	gru
recurrent_layers	4	4
recurrent_units	128	128
batch_size	256	64
steps_per_epoch	77	311
learning_rate	0.000316	0.001
optimizer	adam	adam
num_epochs	95	73

(a) Hyperparameters for RNN models.

Attribute	CNN
conv_filters	64
conv_kernel_size	5
conv_stride	1
batch_size	256
steps_per_epoch	77
learning_rate	0.001
optimizer	adam
conv_layers	2
maxpool_size	3
dense_layers	1
dense_units	64
dropout	0.192
num_epochs	66

(b) Hyperparameters for CNN model.

5.2 Objective evaluation

All results within this section are based on a classification threshold of 0.5, implying that any prediction below 0.5 results in a negative classification by the model.

5.2.1 Classification performance

The final models were evaluated using *accuracy*, *precision*, *recall* and *false positive rate* (FPR). Due to the class imbalance, we have included several metrics beyond accuracy in order to quantify the models' performance on the two classes independently. Precision is included as it provides a clear indication of the quality of a model's positive predictions, whereas recall, also known as *true positive rate* (TPR) is included to evaluate the models' ability to capture positive instances. The FPR is evaluated as it encapsulates the important aspect of false triggers.

Note that the resampling step performed on the training data shifts the class distribution towards a less skewed ratio. This means that training set metrics are not directly comparable to validation- and test metrics, as they do not come from the same underlying distribution.

Table 3: Evaluation metrics for all three models.

Split	Model	Accuracy	Precision	Recall	FPR
train	**rnn**	0.999699	1	0.994275	0
	cnn	0.999799	0.998092	0.998092	0.000106
	rnn_naïve	0.998897	0.997093	0.981870	0.000159
val	**rnn**	0.998496	1	0.971347	0
	cnn	0.996239	0.973684	0.954155	0.001429
	rnn_naïve	0.991575	0.950769	0.885387	0.00254
test	**rnn**	0.998044	1	0.962751	0
	cnn	0.996991	0.971347	0.971347	0.001588
	rnn_naïve	0.992477	0.960000	0.893983	0.002064

Table 3 shows that the RNN is superior in terms of classification accuracy, with 99.80% on the test set. The CNN has a slightly lower accuracy of 99.70%, while the Naïve RNN comes last with 99.25%. We can also see that the RNN has a precision of 1 on all three datasets, which indicates that it is very confident in its positive classifications. This is further substantiated by the remarkable FPR of 0. The two other models both perform worse in regards to precision and FPR, with the CNN having better results than the Naïve RNN. The RNN has a recall of 96.28% on the test set, indicating that it successfully captures most of the positive utterances. Here, the CNN performs slightly better, while the Naïve RNN once more is inferior to both the other models.

5.2.2 Robustness to false triggers

To evaluate the models' robustness concerning false triggers, we ran inference on 10 minutes of white noise. Besides, we extracted 10 minutes of randomly sampled audio clips from the LibriSpeech dataset, which is a large corpus of English speech derived from verbally transcribed audiobooks [Panayotov et al., 2015]. Both inferences were executed using a sliding window with a step length of 200 ms, and the results are shown in Table 4 below.

Table 4: False triggers on 10 minuets of audio.

Model	**White noise**	**LibriSpeech**
Naïve RNN	3	81
CNN	0	56
RNN	0	33

One can see that the Naïve RNN performed poorly, since it falsely reported *white noise* as a trigger word on three occurrences and had 81 false triggers on the LibriSpeech dataset. The CNN and RNN did not trigger from white noice, however they falsely triggered 56 and 33 times on the other dataset respectively.

5.2.3 Computational footprint

To quantify the models' computational footprints, we profiled inference time on the 10 minutes of audio from the LibriSpeech dataset. Table 5 shows the averaged inference time per second of audio over the total 10 minute audio clip, performed on a Nvidia Tesla T4 GPU. Three different batch sizes are included to illustrate the effects of vectorization if multiple examples are processed simultaneously.

Table 5: Computational cost of the three models.

Model	Batch size: 1 t/s [ms]	Batch size: 32 t/s [ms]	Batch size: 256 t/ex [ms]	Num. of parameters
Naïve RNN	71.365	3.150	1.213	371 873
CNN	11.407	1.202	0.932	103 489
RNN	24.413	1.423	0.967	361 633

We can see that the Naïve RNN performs considerably worse than its counterparts for all three batch sizes. In a real-time application the model will likely operate on a single window at a time, as batch processing by caching multiple windows would infer latency proportional to the number of cached windows. Inference on batches containing a single window takes 71.4 ms per second of audio for the Naïve RNN, as opposed to 22.4 ms for the RNN and 11.4 ms for the CNN.

5.3 Achievable configurations

The *receiver operating characteristic* (ROC) curve visualized in Figure 10 shows the compromise between a model's FPR and TPR. By plotting TPR over FPR, one can identify the span of configurations which can be achieved by tuning the classification threshold.

As illustrated in the figure, there are diminishing returns to tweaking the threshold away from the top-left corner, as the rate of decrease for the opposing characteristic is very high in both directions. Lowering the classification threshold would imply a higher TPR at the cost of FPR. This is because a lower threshold does not require the model to be as confident in its prediction for it to be classified as true. The trade-off is relevant for trigger-word detection models, because of the mentioned importance of avoiding false triggers.

When considering the ROC curve, the RNN and CNN models appear to be strictly better than the Naïve RNN model.

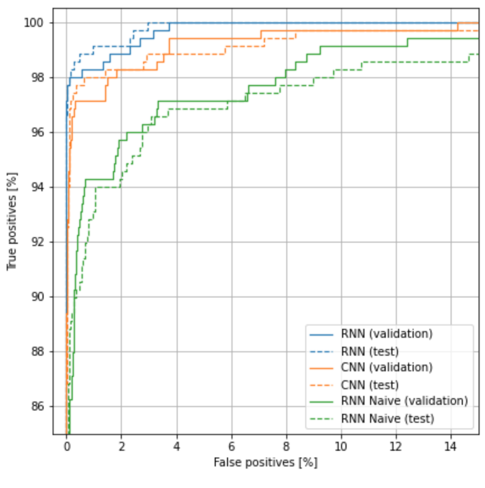


Figure 10: ROC curves for all models.

5.4 Results from empirical testing

We created an interactive Jupyter Notebook for empirical testing, which can be found here.

5.4.1 Empirical testing by the authors

Figure 11 illustrates the result of the RNN after one of the authors says "Hey, Marvin! Has Siri told you about Marvin?". The use of a sliding window is clearly visualized as the audio clip is divided into overlapping segments. When plotting the result, the dots on the left-hand side are colored based on the model's prediction for that window on a continuous scale from dark red (confident negative) to dark green (confident positive). If the predicted value is above the predefined threshold of 0.5, it is classified as positive and the segment is given a green color. As one can see, the model successfully detects both utterances of "Marvin" in the sentence. Because the complete word appears in several overlapping windows, the model classifies positive multiple times. Despite the fourth window being classified as negative, the dot is yellow, thus indicating that the prediction is close to the threshold.

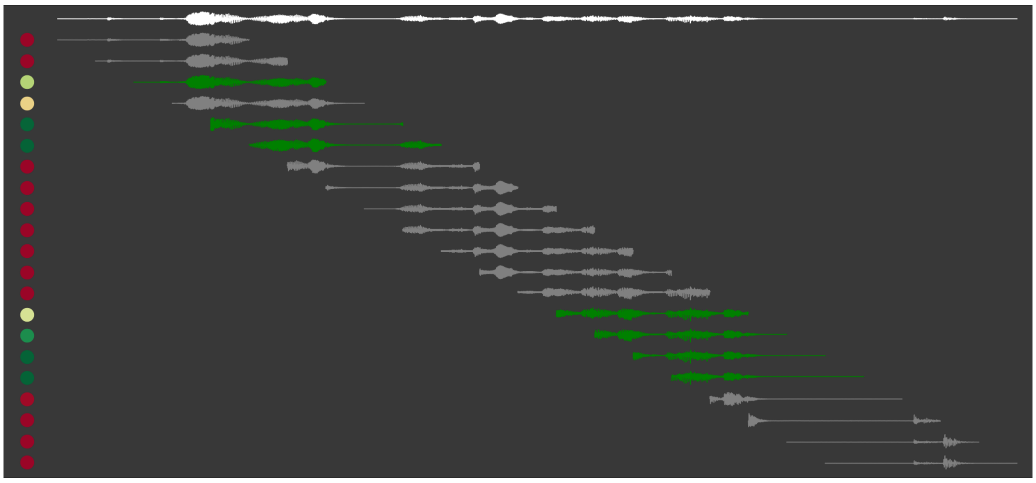


Figure 11: Example result from empirical testing

Figure 11 further illustrates an interesting phenomena worth highlighting. If saying "Hey Marvin", the model seems to consistently be more confident on the last few windows containing the utterance (darker green). This is likely due to that the model is trained to recognize "Marvin" as a standalone word, such that windows containing both "Hey" and "Marvin" might be classified as negative, or positive with lower confidence. This leads to the model having a hard time correctly classifying pronunciations where there is little or no gap between the two words.

After several rounds of testing the models, some empirical results could be drawn regarding classification performance and robustness to false triggers. It was difficult to separate the performance of the CNN and RNN taking Log-Mel spectrograms as input, but they were both notably more accurate than the Naïve RNN. The Naïve model did not correctly detect the utterance of "Marvin" as often as the two others, while at the same time being prone to triggering unexpectedly.

5.4.2 Empirical testing of accents

We reached out to 18 acquaintances from all around the world, who were given the task of testing the RNN model by pronouncing increasingly obscure variations of the phrase "Hey Marvin" in their local accent. All the audio files can be found here. Naturally, some of the pronunciations were not recognized. However, the model was able to detect most utterances of "Marvin", and it was always able to detect at least one of the utterances from every speaker. This indicates a high recall, as a large proportion of true positives were detected.

5.5 Discussion

As presented above, the three models have all been evaluated using different methods, with the results providing diverse insight into relevant factors for trigger word detection.

The objective evaluation of classification performance shows that the RNN taking Log-Mel spectrograms has the best performance. Its accuracy, precision and FPR are higher than both the other models, while recall is slightly lower than the CNNs. However, as outlined in section 5.3, the RNN can achieve a higher true positive rate at the expense of false positive rate. For data privacy reasons we consider the latter to be of greater importance, which is reflected in the overall evaluation of the three models.

Out of the three models, the CNN has the lowest computational footprint of 11.4 ms on processing one second of audio. The Naïve RNN uses significantly more time, while the other RNN uses approximately double the time of the CNN. This is a relatively large difference, but evaluating the computational footprints individually yields another perspective. A quarter of a second latency for the RNN is in our opinion sufficient for the task of trigger word detection and would be perceived as almost immediate feedback if implemented in an embedded system.

We sought to acquire further insight into the models' robustness to false triggers. As presented, all models seemed to have good precision, but this is not reflected in the additional analysis of robustness. The Naïve RNN wrongly detected "Marvin" three times from white noise and 81 times from 10 minutes of audio in the LibriSpeech dataset. This was worse than both the CNN and RNN, which had 56 and 33 wrong triggers respectively. The high amount of false triggers on the LibriSpeech dataset, relative to the Speech Commands Dataset used for implementation, might have a simple explanation. The original dataset only contained 30 words, intended for the application of speech commands. The new dataset has a wide range of extra words and phonemes that the models have not yet encountered, which leads to a higher chance of detecting false positives. Thus, the conclusion of the RNN being better at avoiding false triggers can still be drawn.

The theoretical results are further substantiated by our empirical findings. All models were able to detect the word "Marvin", and the RNN and CNN seemed to perform better than the Naïve RNN. This cannot be used as confirmation, but it certainly supports the theory of both the models taking preprocessed data as input being superior to the baseline model using raw audio.

6 Conclusion

In order to solve the project task regarding trigger word detection, three deep learning methods for binary classification were implemented. Two of the models are RNNs, differing in the input format, while the last model has a CNN architecture. The Naïve RNN takes raw audio data as input and was used as a baseline to evaluate the effects of preprocessing on classification performance. Preprocessing of audio data in the other models aimed to extract as much information as possible from the raw audio, while at the same time lowering input size. Resulting from the preprocessing pipeline is a Log-Mel spectrogram, an image capturing time and frequency, which was fed as input to both the more sophisticated RNN and CNN. All three models were carefully tuned in order to optimize performance prior to evaluation.

The project has been interesting and we have learned a lot about complex Machine Learning methods. It is fascinating to see how a stream of audio can be processed, so that image classification models can be applied in order to detect a trigger word. After the detectors had been fully implemented, it was also exciting to be able to test them ourselves and get immediate feedback on performance in the real world.

6.1 Findings

The three methods have all been evaluated against each other in light of the task of trigger word detection, and the results from the two models using Log-Mel spectrograms as input was evaluated against the Naïve model to examine the hypothesis that measures can be made to extract relevant information from raw audio data.

Regarding classification performance, the results were quite clear. The RNN taking Log-Mel spectrograms as input had substantially better accuracy and precision than both the other models. Regarding computational footprint, the Naïve RNN was clearly inferior. The two other models were considerably faster, and despite the CNN being better than the RNN, they are both sufficiently quick for real-time trigger word detection. The RNNs classification performance weighs in our opinion up for the slight difference in computational footprint. Avoidance of false triggers is an important aspect. This was evaluated against the LibriSpeech dataset, and once again the RNN had the best performance and comes as no surprise as the RNN had highest the precision among the models.

After a thorough assessment of the different evaluations, we have concluded that the RNN taking Log-Mel spectrograms as input is the best model with regards to trigger word detection. This is justified by its superior performance to the two other models in every aspect other than computational footprint, where the CNN was slightly better.

We had an initial hypothesis that measures could be made to extract relevant features from the audio data before classification, and that this would lead to a better mapping for the complex models. As highlighted, the Naïve model that takes raw audio as input performed poorer than both the other models in every evaluation. The performance of this baseline was significantly worse, while also having a higher computational footprint, which confirms that our preprocessing pipeline resulting in a Log-Mel spectrogram had a positive effect on overall performance.

6.2 Future work

If there were more resources available, we would have expanded the scope of the project to include additional aspects needed in a fully integrated voice assistant. This involves both improving the trigger word detector and implementing explicit interaction with the users.

We have some thoughts on how the detector could be improved. It should be capable of recognizing both ”Hey” and ”Marvin” in succession, as the combination of these words would lower the probability of false triggers. The classification could be performed by feeding probabilities from the neural network to an HMM, much like the method of *Siri* outlined in section 2.2.1. This would enable the model to detect the two words in correct order based on the predictions from a continuous stream of probabilities, thus solving the current problem of triggering from ”Marvin” as a standalone word.

If there was more time, the trigger word detector could have been combined with explicit user interaction, which would require the implementation of a system for *Natural Language Processing*. We have played with the thought of applying Google's *speech-to-text* API for transcription of audio data after a trigger word has been detected. The transcribed audio could then be processed by an *intent classification model*, and in this manner, we could map commands to certain actions, like for example telling the time or extraction of weather information from an external API.

Bibliography

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

Xavier Amatriain, Jordi Bonada, Alex Loscos, and Xavier Serra. Spectral processing. *Zölzer U, editor. DAFX-Digital Audio Effects. Chichester: John Wiley & Sons; 2002.*, 2002.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

Guoguo Chen, Carolina Parada, and Georg Heigold. Small-footprint keyword spotting using deep neural networks. *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2014.

Israel Cohen and Sharon Gannot. *Springer Handbook of Speech Processing*. 01 2008. ISBN 978-3-540-49125-5. doi: 10.1007/978-3-540-49127-9_44.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, pages 326–366, 367–412. MIT Press, 2016.

Matthew Hoy. Alexa, siri, cortana, and more: An introduction to voice assistants. *Medical Reference Services Quarterly*, 37:81–88, 01 2018. doi: 10.1080/02763869.2018.1404391.

H. Ide and T. Kurita. Improvement of learning for cnn with relu activation by sparse regularization. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2684–2691. IEEE, 2017.

Woojay Jeon, Leo Liu, and Henry Mason. Voice trigger detection from lvcsr hypothesis lattices using bidirectional lattice recurrent neural networks. 2020.

Diederik P Kingma and J Adam Ba. A method for stochastic optimization. arxiv 2014. *arXiv preprint arXiv:1412.6980*, 434, 2019.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

Rajath Kumar, Vaishnavi Yeruva, and Sriram Ganapathy. On convolutional lstm modeling for joint wake-word detection and text dependent speaker verification. pages 1121–1125, 09 2018. doi: 10.21437/Interspeech.2018-1759.

S. R. Madikeri. Mel filter bank energy-based slope feature and its application to speaker recognition. In *2011 National Conference on Communications*, pages 1–4, 2011. doi: 10.1109/NCC.2011.5734713.

Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: an asr corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210. IEEE, 2015.

D. S. Park, W. Chan, Y. Zhang, C. Chiu, B. Zoph, E. D Cubuk, and Q. V. Le. Specaugment: A simple data augmentation method for automatic speech recognition. *arXiv preprint arXiv:1904.08779*, 2019.

Haşim Sak, Andrew Senior, Kanishka Rao, and Françoise Beaufays. Fast and accurate recurrent neural network acoustic models for speech recognition. *arXiv preprint arXiv:1507.06947*, 2015.

Chris Seiffert, Taghi M Khoshgoftaar, Jason Van Hulse, and Amri Napolitano. Resampling or reweighting: A comparison of boosting implementations. In *2008 20th IEEE International Conference on Tools with Artificial Intelligence*, volume 1, pages 445–451. IEEE, 2008.

S. Sigtia, R. Haynes, H. Richards, E. Marchi, and J. Bridle. Efficient voice trigger detection for low resource hardware. In *Interspeech 2018*, pages 2092–2096, 2018. doi: 10.21437/Interspeech.2018-2204.

Shrutika Singh, Harshita Arya, and P. Arun Kumar. Voice assistant for ubuntu implementation using deep neural network. In *Advanced Computing Technologies and Applications*, pages 11–20, Singapore, 2020. Springer Singapore. ISBN 978-981-15-3242-9.

Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018.

Lonce Wyse. Audio spectrogram representations for processing with convolutional neural networks. *arXiv preprint arXiv:1706.09559*, 2017.

Appendix

A Model configurations

Table 6: All parameters and hyperparameters, metadata for the three models.

Attribute	Naïve RNN	CNN	RNN	Description
`train_split`	0.600	0.600	0.600	Training dataset split ratio.
`val_split`	0.200	0.200	0.200	Validation dataset split ratio.
`test_split`	0.200	0.200	0.200	Test dataset split ratio.
`sample_rate`	16000	16000	16000	Audio sample rate.
`min_freq`	—	0	0	Lowest mel-bin frequency.
`max_freq`	—	8000	8000	Highest mel-bin frequency.

(a) Overview of parameters for the three models.

Attribute	Naïve RNN	CNN	RNN	Description
`frame_size`	—	512	512	Fourier Transform window size.
`frame_step`	—	256	256	Fourier Transform hop size.
`fft_size`	—	512	512	Fourier Transform discretization bins.
`mel_bins`	—	64	64	Number of mel filterbank bins.
`max_time_mask`	—	10	10	Max. width of temporal mask.
`max_freq_mask`	—	10	10	Max. width of frequency mask.

(b) Overview of preprocessing hyperparameters for the three models.

Attribute	Naïve RNN	CNN	RNN	Description
`n_timesteps`	16	—	—	Num. of chunks per audio sample.
`conv_filters`	32	64	32	Num. of filters in first Conv1D layer.
`conv_kernel_size`	24	5	1	Kernel size for Conv1D layers.
`conv_stride`	4	1	2	Stride for Conv1D layers.
`recurrent_cell`	gru	—	gru	Recurrent cell type.
`recurrent_layers`	4	—	4	Num. of recurrent layers.
`recurrent_units`	128	—	128	Num. of units per recurrent cell.
`batch_size`	256	256	64	Training batch size.
`steps_per_epoch`	77	77	311	Number of batches per epoch.
`learning_rate`	0.000316	0.001	0.001	Learning rate.
`optimizer`	adam[a]	adam[a]	adam[a]	Optimizer.
`conv_layers`	—	2	—	Num. of Conv1D layers.
`maxpool_size`	—	3	—	Pool size for MaxPooling1D layer.
`dense_layers`	—	1	—	Num. of dense layers.
`dense_units`	—	64	—	Num. of units in first dense layer.
`dropout`	—	0.192	—	Probability of neuron deactivation.

(c) Overview of hyperparameters for the three models.

[a]First order decay rate: 0.9, Second order decay rate: 0.999

Attribute	Naïve RNN	CNN	RNN	Description
`cpu_1`	0.288	0.080	0.270	Avg. inference time on batches of 1.
`cpu_10`	0.109	0.065	0.072	Avg. inference time on batches of 10.
`cpu_100`	0.631	0.353	0.395	Avg. inference time on batches of 100.
`gpu_1`	0.086	0.049	0.064	Avg. inference time on batches of 1.
`gpu_10`	0.053	0.036	0.034	Avg. inference time on batches of 10.
`gpu_100`	0.126	0.053	0.056	Avg. inference time on batches of 100.
`trainable_params`	371873	103105	361633	Num. of trainable parameters.
`non_trainable_params`	0	384	0	Num. of non-trainable parameters.
`total_params`	371873	103489	361633	Total num. of parameters.
`num_epochs`[a]	95 (75)	66 (46)	73 (53)	Number of epochs.

(d) Other details for the three models.

[a]Early stopping in parenthesis.

B Explanation of output bias initialization

For all three models, we have been using Xavier initialization of weights in the output layer. As Xavier initialization draws initial weights from a truncated normal distribution centered around zero, the expected value of the weights are zero at initialization. For the model to initially predict *positive* with a probability of 10%, we can set the bias term such that the expected value of the output activation in the last layer equals 0.1.

By doing this, we prevent the model from spending several epochs on learning that guesses heavily biased towards zero will lower its loss. Using a sigmoid activation function in the output layer, we obtain the following equations:

$$Sigmoid(z_L) := \frac{1}{1+e^{-z_L}}, \qquad z_L = W_L a_{L-1} + b_L$$

$$\frac{1}{1+e^{-z_L}} = 0.1 \quad \Leftrightarrow \quad \frac{1}{1+e^{-(W_L a_{L-1}+b_L)}} = 0.1$$

where W_L and b_L denote the weight matrix and bias term in the output layer, and a_{L-1} equals the output activations from the second to last layer. If assuming that a_{L-1} is random at the very first epoch, since no patterns has been learnt, the expected value $E[W_L a_{L-1}]$ is zero. Thus, we obtain the following equation:

$$\frac{1}{1+e^{-b_L}} = 0.1 \quad \Rightarrow \quad b_L = ln(\frac{1}{9}) \approx -2.197$$

In conclusion, setting the initial bias of the output layer to -2.197 leads to the expected activation value of the last layer being close to 0.1, thus on average predicting *positive* in 10% of cases for the first epoch. This speeds up convergence as the model does not have to spend the first few epochs learning the class distribution bias.