Radboud University

A Spiking Neural Network for Speech Recognition

Thijs Luttikholt* *s1007894*

Thijme de Valk* *s4798902*

Marit Hagens* *s4808061*

Pascal Schröder* *s1062138*

*All authors contributed equally to the project and this report

March 9, 2021

1 Introduction

Research in the domain of automatic speech recognition (ASR) focuses on translating a speech signal into words. The biggest difficulty in this process is the high variability between the speech signals [7]. Such variation is, among others, caused by gender, age, dialect, and speech tempo [5].

ASR is being developed to improve communication between humans and machines [13]. Effective human-computer communication is important for a large number of applications, such as voice command functionality, speech-to-text conversion, and e-help. For example, with proper speech recognition, call centers can be partially automated to increase the speed of helping customers and decrease the costs for the call center [6].

Over the years, the development in ASR improved. Due to the rise of better GPUs, more computing power has become available to train an Artificial Neural Network (ANN) system on larger datasets. Additionally, the expansion of data collection has made more data available for training. These developments have increased the generalizability of ASR models [13].

State of the art ANNs can outperform human judgement on some tasks in restricted domains. One such example is the TIDIGITS dataset of spoken digits. Deep Neural Networks reach the highest performance on this dataset, with up to 99 % accuracy, but do so at high training costs [4]. Less complex models, that can be used on simple mobile devices, have been shown to reach accuracies of approximately 97 % [8]. Although these models perform well, ANNs and in particular deep ANNs suffer from high energy requirements due to the computational power they require.

1.1 Spiking Networks for ASR

This problem of high energy demand has led to new applications, inspired by the human biological neural systems. Spiking Neural Networks (SNNs) are different from ANNs in that they do not communicate using rate-based coding, but communicate using single bursts of activation (spikes)

when a neuron's input reaches a certain threshold. Computation is thus carried into the time domain, and since not every neuron spikes at each timestep, SNNs are much more energy-efficient in comparison to ANNs. Additionally, the spiking behaviour allows for more biologically inspired learning. Instead of the non-local error propagation present in ANNs, time-based learning rules such as spike-timing-dependent-plasticity (STDP) can be used to train SNNs efficiently [1, 3].

In recent years, SNN approaches to ASR have been proposed. Using an unsupervised self-organising map for feature representation and an SNN for classification, researchers were able to reach 97.6 % accuracy on the TIDIGITS dataset [11, 12]. Another network structure has been proposed by Dong et al. [1], consisting of an SNN that is trained using a simplified STDP learning rule. The network identifies features and a support vector machine (SVM) is used to classify samples based on these features. Using this relatively simple approach, an accuracy of 97.5 % is reached.

1.2 Aim of the Project

The goal of this project is to replicate the network that is described by Dong et al. [1] and to reproduce the performance results that they find for the TIDIGITS dataset. We evaluate the performance of the replicated network in terms of accuracy, and by inspection of the network's learned representations. The learning objective of this approach is two-fold. First, by replicating the network, we gain a better understanding of the type of networks that can be built using spiking neurons. Second, the visualisation of the network representation can inform us about possible inconsistencies with the original network. The paper by Dong et al. [1] also discusses a larger network structure, consisting of 70 feature maps, that was tested on an adapted version of the TIMIT dataset [2]. However, due to the lack of a description of the required changes to the dataset, we found it problematic to try to reproduce these results. Thus, this project will solely focus on the re-implementation of the TIDIGITS network.

2 Methods

We will closely replicate the model as described by Dong et al. [1], shown in figure 1. This network starts with the input layer, in which the Mel-Frequency Spectral Coefficients of the data are translated into spikes using time-to-first-spike coding. These resulting spikes are used as input to the convolutional layer, which is responsible for learning different features. The feature maps are learned with spike-timing-dependent plasticity. The output spikes of the convolutional layer are compressed in the pooling layer. The potentials of the pooling layer are then used in a linear classifier to make the predictions. The details of these layers will be explained in this section. But first, the data acquisition and preprocessing will be explained. All code for the model and project can be found on GitHub[1].

2.1 Data acquisition and preprocessing

The data that were used for this report were provided by Pan et al. [9], and can be found on their GitHub[2]. The data were split into two zipped folders, one containing TIDIGITS data and the other containing TIMIT data. We only use the TIDIGITS data. The TIDIGITS dataset is a speech corpus of spoken digits, voiced by various different speakers. The vocabulary consists of 11 different, isolated digits, ranging from 1 up to and including 11. In the training set, every digit is repeated 224 times and in the test set 226 times. Note that the data that we used are not identical to that of the original paper, which covers the digits 0 up to and including 9. A different dataset is used since the data to which the authors refer requires payment, which was not considered to be within the scope of this project.

[1] https://github.com/verrannt/neuromorphic_project
[2] https://github.com/pandarialTJU/Biologically-plausible-Auditory-Encoding

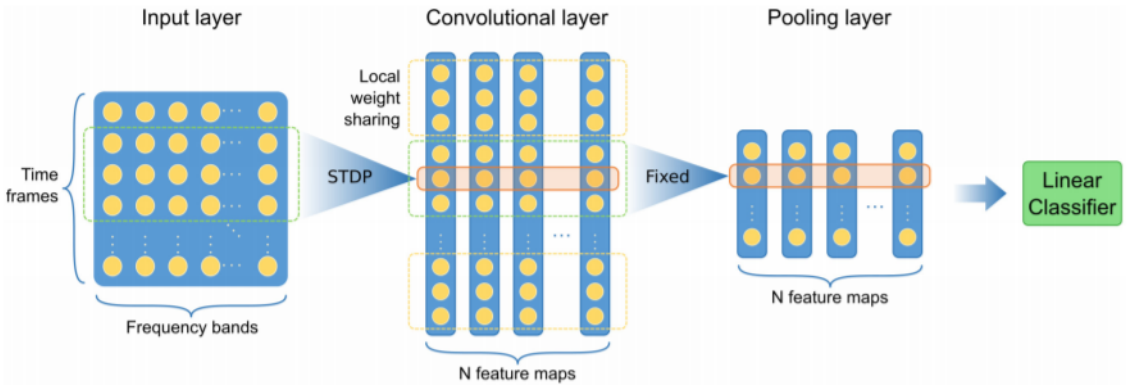


Figure 1: *The network architecture as proposed by Dong et al. [1]. The network starts with an input layer, followed by a convolutional layer, and ending with a pooling layer. The potentials of the pooling layer are used in a linear classifier to make the predictions.*

For the TIDIGITS set, the preprocessing comprised the translation of data from Matlab files to numpy arrays to be used in Python. After this preprocessing, numpy arrays for the sampling rates and the sounds were left for the TIDIGITS dataset. In order to make this data usable for the input layer as described in the paper, these data were converted to Mel-Frequency Spectral Coefficients (MFSC) features. These features make up the Mel-frequency spectrum of a sound, which in turn represents a power spectrum of the sound. This process will be described next.

2.1.1 MFSC features

In order to convert sounds to their corresponding MFSC features in the correct shape, a new program was created. Due to the authors mentioning that the window length would have to be varied in order to obtain fixed size outputs, we chose to create a new program from scratch instead of relying on an external library. This allowed us to have full control over the MFSC conversion step. We will now describe the program generally, for further details, we refer to the code itself.

The function that was created to convert one sound into its MFSC features required 4 parameters for controlling the shape and output. First, the sound and its corresponding sample rate are required. Second, the amounts of time frames and frequency bins are expected to be given. The former are the input values based on which the MFSC conversion was done. The latter are the parameters that determine the output size of the function.

The function starts by normalizing the audio input, after which this normalized audio is framed. This framing process calculates the Fourier transform frame length and the Fourier transform size. This calculation makes use of the length of the audio input and the specified amount of time frames of which the output should consist. After the framing process, a Fourier transform is applied to convert the sound to the frequency domain instead of the time domain. Subsequently, the signal power is calculated and Mel-spaced filter banks are determined. This last process involves computing filter points based on various variables. These include a lower bound frequency of 0, the Nyquist rate, the amount of frequency bins we want the output to have, the length of the Fourier transform frames, and the sample rate.

After calculation of the filter points, the Mel-spaced filter banks are created and the signal is filtered using the filter banks. As a last step, the log of the signal is taken to finish the MFSC process. Usually a final step involving cepstral coefficients would have been necessary, then it would be so called Mel-Frequency Cepstral Coefficients (MFCC) features. However, the paper which we aim to replicate chose not to use the MFCC features, although these features are used more commonly. This was due to MFCC features losing the locality of the features in the original sound, whereas the network's convolutional layer requires this locality [1].

For the computation of MFSC features, it should be noted that our approach of calculating the frame length in the Fourier transform step may not be necessarily similar to that which the

original authors have used. This is due to a lack of detail regarding the calculation of this length in the original paper. Which mentions "we use different window length in the Fourier transform step during the MFSC feature extraction to get an input of fixed length" [1]. This in itself shows that the window length was varied based on the samples. However, it does not clearly state in what manner this length variation was done. Therefore, our implementation can possibly follow a different principle.

The implementation as was described above was compared to a different implementation which made use of the Python library called 'python_speech_features'. This library contained a function called 'logfbank' which performs an MFSC conversion as well. However, the use of this library required us to have a fixed size for the FFT size, which contradicts a statement made in the original paper about using a differing size. A comparison between the two implementations revealed that the one using the Python library yielded higher accuracies for the resulting network performance. Thus, this implementation was used to obtain the results that will be discussed later in this report.

2.2 Input layer

The purpose of the input layer is to encode a given MFSC feature image as a time-to-first-spike representation, i.e. the time it takes for a neuron representing a single pixel to fire initially. This is done in the following way: first, the minimum and maximum values of the full 41×40 MFSC score frame are calculated. Subsequently, a $30 \times 41 \times 40$ Boolean array is created, where the first dimension corresponds to 30 available discrete timesteps. Each of the 41×40 values in every timestep corresponds to a neuron encoding for the corresponding pixel in the MFSC image, with a single value which is set to true if the neuron fires in that timestep. Next, the MFSC image is mapped onto the Boolean array such that the highest pixel value will be found in the first timestep and the lowest in the last, with intermediate values spread linearly over the intermediate timesteps. To give an example, let us assume that a score at index (1,1) in the MFSC image is computed to spike at timestep 13. This means that the Boolean array will be set to true at index (13,1,1), whereas all other indices of the form $(x, 1, 1)$ with $x \in [1, 30]$, which correspond to the same neuron, are set to false. The input layer then returns the Boolean array as its output.

It must be noted that the conversion to time-to-first-spike may not be identical to the approach in the original paper. After having read the paper carefully, it was determined that a clear conversion method was not specified. Therefore, the approach using discrete timesteps was chosen since it seemed to be the most logical choice. The decision for 30 timesteps was also not clearly stated in the original paper. However, the paper contained an image of an example time-to-first spike encoding, based on which we determined that 30 timesteps was most similar to what the original authors seemed to have used.

2.3 Convolutional layer

The convolutional layer of the model was implemented using integrate-and-fire (IF) neurons, which are simplifications of the more commonly used leaky-integrate-and-fire neuron models for which no leakage is assumed. An IF neuron in our network fires when its membrane potential reaches a threshold of 23. The convolution process is performed over the time domain of the time-to-first-spike representation. All frequency band values within a time window of 6 time frames are convolved with a stride of 1, resulting in 36 overlapping periods of time (rows). The convolutional layer consists of multiple sub-layers that each correspond to their own feature map, and have their own weights. In total, 50 feature maps resulting from 50 sub-layers are represented in the network. To detect the same linguistic features occurring at slightly different timings per sample, local weights sharing is used. The size of the shared weights window is 4 time points in our implementation, the same as in the original paper. These windows do not overlap, which leads to 9 weight sharing windows. The weights are initiated randomly by sampling from a Gaussian distribution with a mean of 0.8 and a standard deviation of 0.05, which is identical to the original implementation. To prevent that different feature maps are activated for the same periods of time, lateral inhibition was implemented. If a IF neuron fires, all neurons in other feature maps at the

same position become inhibited, and can no longer fire for the current input sample. Unfortunately, the description of this process in the original paper [1] did not state whether lateral inhibition continues after training. We made the choice to implement lateral inhibition for both training and testing.

2.4 STDP learning of the convolutional layer weights

The learning of the convolutional layer weights is inspired by the audio processing systems in the human ear and brain. A simplified version of STDP is used, where a connection is either strengthened or weakened. The connection is strengthened if the presynaptic neuron fires before the postsynaptic neuron. The connection is weakened if the presynaptic neuron fires after the postsynaptic neuron. The relative time difference between the spikes is irrelevant for our simplified STDP. This simplified STDP rule is defined as follows:

$$\Delta w_{ij} = \begin{cases} a^+ w_{ij}(1 - w_{ij}), & \text{if } t_j < t_i \\ -a^- w_{ij}(1 - w_{ij}), & \text{otherwise} \end{cases} \tag{1}$$

where w_{ij} is the weight between the ith neuron in convolutional layer and the jth neuron in input layer, t_i and t_j are the corresponding firing time of the two neurons, and a^+ and a^- are the learning rates. We used the learning rates used by the original paper, $a^+ = 0.004$ and $a^- = 0.003$. Learning is stopped completely for all samples once the absolute change in weights for one particular sample summed together is smaller than 0.01. At that moment, convergence of the model is assumed.

Additionally, we use a process of lateral inhibition to ensure that different feature maps learn different features. Neurons in the same row or in the same shared weight block of a firing postsynaptic neuron will not update their weights until the next sample.

2.5 Pooling layer

The pooling layer computes the summed activation for each shared weight block of neurons. Therefore, it reduces the dimension in the time domain, but keeps the different feature maps separated. The neurons in the pooling layer are not allowed to fire. Rather, their membrane potentials represent the embedding space that is used during classification. The membrane potentials are computed as the summation of convolutional layer spikes for each block of shared weights, and for all feature maps.

2.6 SVM classification

The speech model itself does not provide a functionality to classify its input. Rather, it acts as a feature extractor, where the membrane potentials of the pooling layer represent a lower dimensional embedding of the inputs. To measure the effectiveness as such, a Support Vector Machine (SVM) is used to measure train and test classification performance on the embeddings. This is implemented in the following manner: the data is split into a train and validation set, with 20% of data reserved for validation. During training, the potentials computed by the pooling layer for every MFSC input frame are collected in each episode. When all training inputs are iterated through, a linear SVM is fitted on the obtained potentials and its accuracy score on predicting the training labels is recorded . The SVM is implemented using SciKit Learn's LinearSVC [10]. Then, all samples in the validation set are iterated through to obtain the corresponding potentials. Subsequently, the SVM is scored on the validation potentials to record the classification accuracy for inputs it has not seen before. This process is repeated for every training episode and allows us to observe the model's performance during training and monitor its tendency to overfit on the data. Consequently, a testing functionality is implemented that trains another SVM on the training potentials from the last training episode and scores it on the entire corpus of the testing set.

3 Results

We tested our model on the TIDIGITS dataset [9]. The data were preprocessed with the Python library 'python_speech_features' to create the MFSC features. The model was trained for 12 episode to learn the STDP weights. We fit the SVM on the training potentials as computed by the pooling layer at the end of each episode, and record its classification accuracy on the training as well as validation set. This resulted in the training curve of figure 2. The accuracy on the training set converges rather quickly to 100%. It reaches 100% after 4 episodes and never falls below this score after that. The performance on the validation set does not improve as fast and is around 90% after 12 episodes. After 12 episodes, the model weights were frozen and we recorded the embeddings for the test set. The SVM trained on the SSN potentials of the last training episode achieved an accuracy of 91% on the testing set. See figure 3 for the confusion matrix on the test set. This shows that digit 6 has the best recognition accuracy of 98%. The worst recognized digit is 3, with an accuracy of 85%.

Just like in the original paper, we also tried to classify the digits based on the MFSC features alone. We fitted the SVM on the MFSC features of the training set. The fitted SVM reached a 95% accuracy on the test set. This means that the SNN cannot exceed the performance of the MFSC features.

Figure 2: *Classification accuracy of the linear SVM on the pooling layer's embeddings over 12 episodes of training. Distinguished between accuracy on the training set and validation set. Note how training accuracy quickly reaches* 100% *after 4 episodes, while validation accuracy continuously increases. Training was stopped after 12 episodes because no significant gains could be observed anymore.*

3.1 Running time

The training, including fitting and validating of the SVM, took 13 minutes on average per episode, with an overall 2.6 hours of wall clock time. The training was done on an Intel i7-6700K desktop CPU. We investigated possibilities to speed up execution time, and managed to reduce the time for a single episode from an additional 30 minutes down to the mentioned 13 minutes. We note that significant gains could be achieved by implementing the model in a different programming language, as Python is known for its low speed. Unfortunately, the paradigm of the STDP updates did not allow us to leverage the benefits of parallelization, since every simulation step requires the weight updates of the previous step.

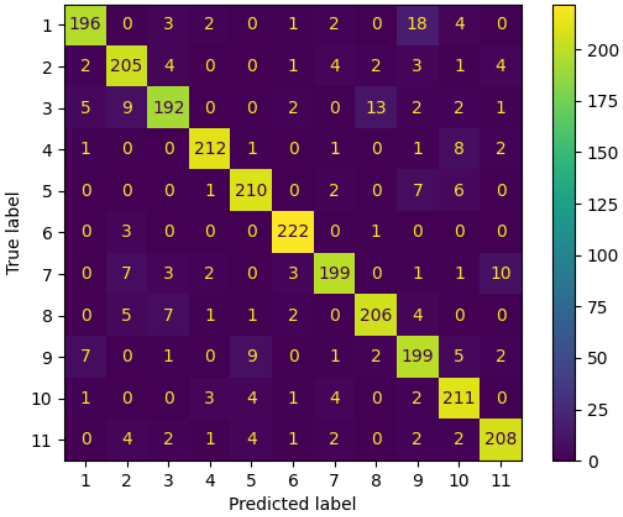


Figure 3: *Confusion matrix for the SVM fitted on the last training potentials after 12 episodes, tested on the test potentials. The test set contains each digit 226 times.*

3.2 Convolutional layer receptive fields

To gain additional insights into what the SNN has learned, Dong et al. [1] plotted the weights between the input layer and the convolutional layer over time during training. We do the same, to easily interpret the receptive fields of the different feature maps. See figure 4 for a depiction of feature maps 5, 10, and 15 of the fifth weight sharing section plotted every 2,000 steps during training. We chose those specific feature maps as the difference between them is sufficient to show that earlier feature maps converge faster than later feature maps to a distinct receptive field. The figure clearly shows that each feature map learns to recognize a different pattern, this is due to the lateral inhibition.

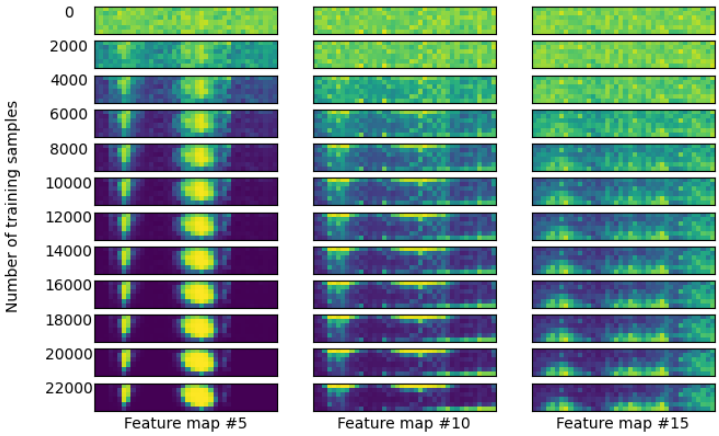


Figure 4: *Feature maps 5, 10, and 15 of the fifth weight sharing section of the convolutional layer plotted every 2,000 steps during training. All show an emergence of weights focused on a specific region, with weights in other regions moving towards zero. This process is more pronounced in early feature maps.*

To gain even more insights into what the network learns by means of STDP, we visualized the receptive fields for all neurons after training completed. Figure 5 shows all features after 12 episodes of training. This allows us to make two observations. To start with, the first feature maps again learn first, as we see that the earlier feature maps have progressed further in learning a pattern. The unnecessary weights for those feature maps are almost zero (dark blue), while the unnecessary weights for the later feature maps have not progressed that far yet. Approximately half of all the feature maps are almost fully learned. Second, the digit samples are more pronounced in the middle of the sample. The figure shows that many sharing windows in the middle were able to get a distinct patterns for most feature maps. However, the first and last sharing windows were only able to get a few feature maps with distinct patterns. This shows that most of the differences in the signal of the digits are in the middle part of the sample.

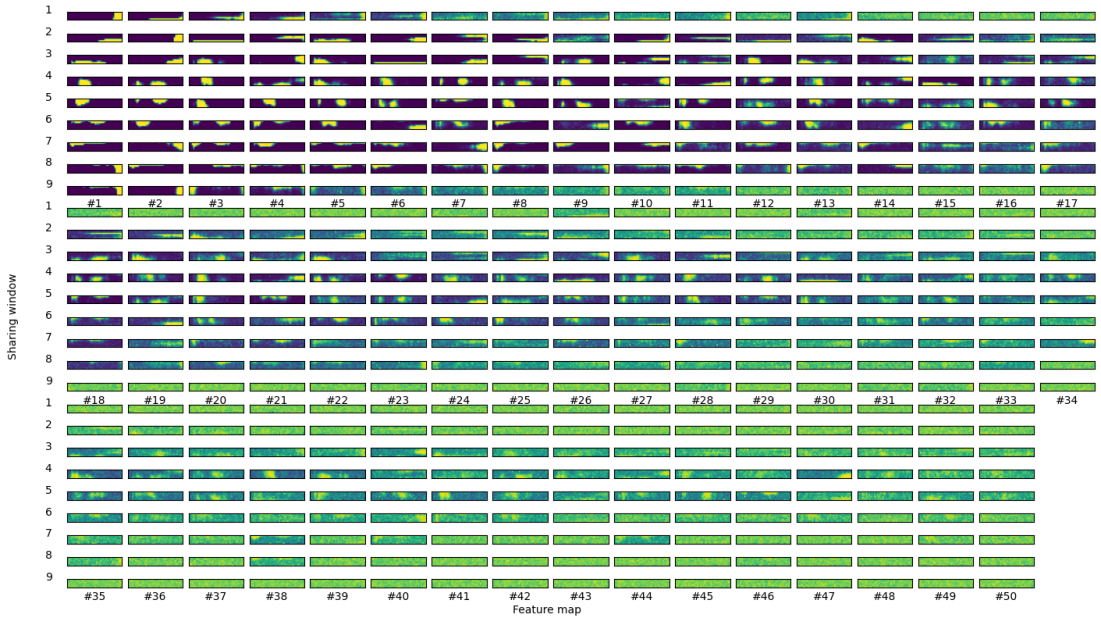


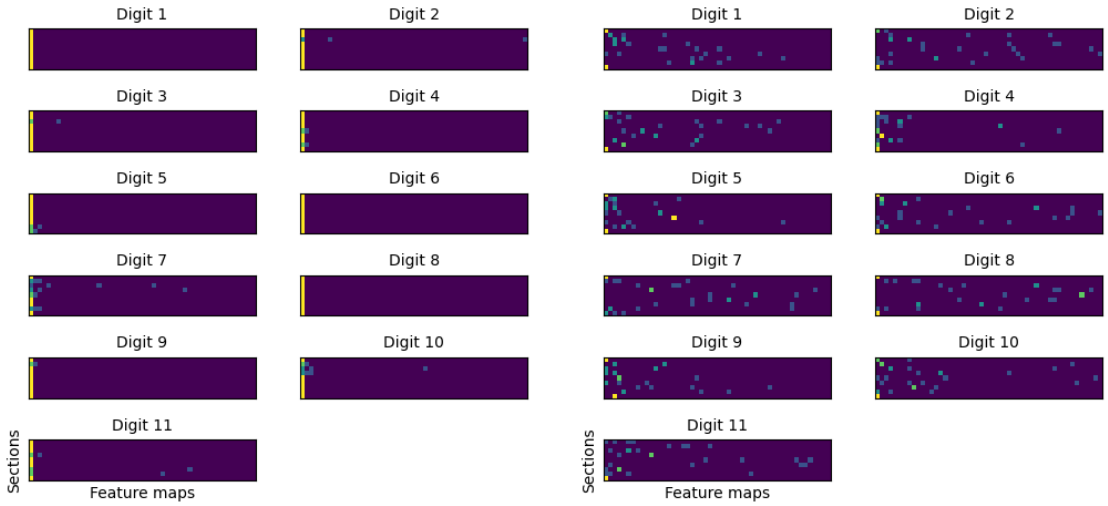
Figure 5: *All feature maps for all weight sharing windows after 12 episodes of training. This shows that it takes longer for later feature maps to learn.*

3.3 Pooling layer feature map activations

The goal of the lateral inhibition of different feature maps is to let the feature maps learn distinct patterns. To see whether different classes of digits are represented by different feature map activations, we plot the activation of the pooling layer neurons for randomly chosen instances of each digit class. The results can be seen in figure 6. Figure 6a shows the outputs for a randomly initialized model, while figure 6b shows the outputs for the model trained for 12 episodes. Note how the random model mostly has high activation values in the leftmost column, the first feature map. The trained model clearly shows different activation patterns for each digit class. Additionally, it is worth noticing that the output for most neurons, even in the trained model, is zero (deep purple). This shows that the model representations are sparsely encoded.

3.4 Continuing training

Since validation and test accuracy was still lacking after 12 episodes, we trained the model for an additional 36 episodes (see figure 7). We noted that although validation accuracy improved to around 95%, the improvement in testing accuracy was only minor, with a final accuracy score



(a) *Random Initialization* **(b)** *After 12 Episodes*

Figure 6: *Outputs of the pooling layer for a randomly chosen uttering of each digit. (a) depicts the outputs of a newly initialized model, (b) the outputs of the model after 12 episodes of training. Dark pixels represent zero potentials, while brighter ones represent high potentials.*

of 92%. We chose the 12 episode model as our main model to report since 48 episodes took an additional 8 hours of training, but did not yield great benefits in terms of accuracy.

Even though testing accuracy did not improve considerably, we observed that many more feature maps converged onto a distinct receptive field. In contrast, approximately half of the feature maps were still highly noisy after 12 episodes.

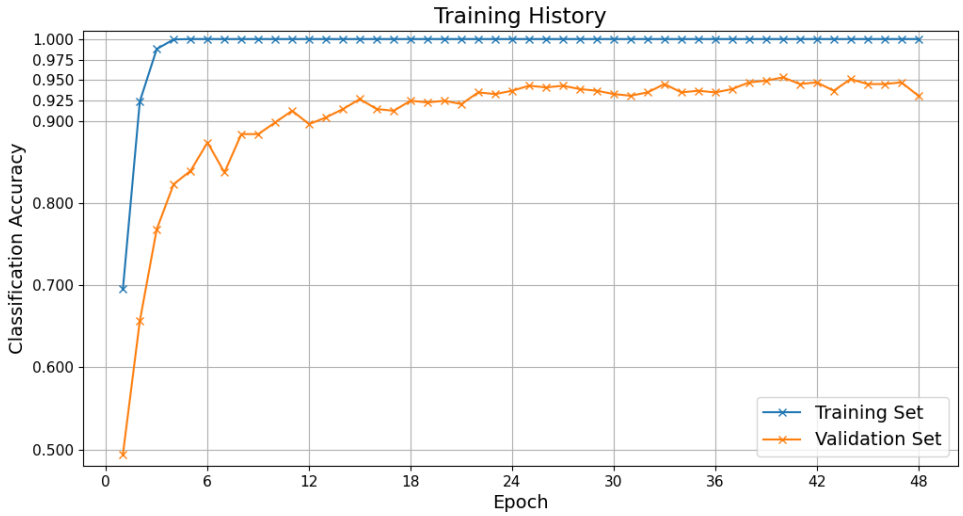


Figure 7: *Classification accuracy of the linear SVM over a total of 48 episodes of training. Note how validation accuracy still improves after episode 12, while we found that testing accuracy only improved to* 92% *in episode 48, compared to* 91% *already achieved in episode 12.*

4 Discussion

We have found that our model performs quite well, we reach a test accuracy of 91% after 12 episodes, which is approximately 23,500 training samples. However, compared to the original paper, this is unexpected. They already reach 95% after 900 samples and even 97.5% after 6,000 samples. We get quite close to those accuracies, but our convergence speed is a lot slower. However, we should take into account that we used a slightly different dataset as compared to the original paper. Our TIDIGITS dataset has 11 different digits, ranging from 1 up to and including 11, while the original paper has 10 different digits, ranging from 0 up to and including 9. This leads to differences in speech signals, which might lead to a more difficult classification task. However, our MFSC features alone do reach a 95% accuracy with the SVM. This shows that, at least, the MFSC features of our TIDIGITS data do have the same capacity of classification as the original paper, which means that the bad performance of our model is most likely caused by our SNN and not by the TIDIGITS dataset. The difference in performance can be caused by several areas of the project. These areas will be discussed in this section.

Firstly, the update order for the weights was not made clear in the original paper. Moreover, the paper failed to specify whether the lateral inhibition would occur at the timestep of the first firing neuron, thus inhibiting other neurons from firing during that same timestep, or whether inhibition would only occur for neurons firing in the next timesteps. The feature maps in figure 5 show that mainly the first feature maps learn well. It takes quite a while for the later feature maps to learn to recognize a pattern. The lateral inhibition should cause competition between the feature maps, which it does, since all feature maps have a different pattern. However, it also causes the feature maps to be slow in learning, which should not happen. Therefore, it can be assumed that the update order of the weights or the lateral inhibition thus does not function as it should. We have chosen to employ a sequential order updating the weights, going from the first to the last feature map and from the bottom to the top of each individual map, that works instantly. However, a random ordering might also have been used, which would likely change the weights that result from the STDP updating. Another possible strategy would be to use a more competitive mechanism, in which the neurons that went the highest above their threshold would be considered first. This strategy was eventually not employed in this study, as there was no clear evidence of such a strategy having been used in the original paper.

A second possible cause of the differences in results may be found in the stopping criterion for the STDP weight updating. Figure 4 shows the learning of feature map 5 of the fifth weight sharing section, which barely learns anything after 16,000 samples. The feature map is still allowed to learn and due to the lateral inhibition, other feature maps in that row are not allowed to learn anymore when this feature map learns. However, this feature map does not need to learn anymore. The original paper indicated that the stopping criterion would be when $|\Delta w|$ would be lower than 0.01 [1]. This was interpreted as indicating that the total weight difference over all of the calculations within one sample should be below the threshold. Conversely, a possible interpretation might have been that each individual set of weights has its own threshold value, and it would be possible to stop training a specific set of weights at a time. Instead of stopping training for all weights once the threshold value has been reached. Due to the uncertainty regarding which approach was chosen in the original paper, we have chosen for our own interpretation, which might not be identical to the original one.

Lastly, a possible cause of difference in results might be found in the input layer. The original paper mentions that this layer converts an MFSC feature to a time-to-first-spike representation [1]. However, it does not mention in what way this conversion is performed, nor does it clearly state the number of time steps used. Therefore, it was chosen in this study to base the number of time steps on Fig.2 in the original paper, resulting in setting it to 30. The conversion method itself was chosen to be based on the minimum and maximum value of the MFSC features. Figure 5 shows that the feature maps in the middle sharing window are able to learn the most different feature maps. This means that the first and last sharing window, thus the start and end of the signal, have less different features to learn. It is possible that the time-to-first-spike convergence is not linear, such that each sharing window has an equal amount of features to learn. This indicates

that our time-to-first-spike approach might not be identical to the approach used in the original paper, therefore resulting in differences in results.

5 Conclusion

To conclude, our model performs reasonable well, reaching a test accuracy of 91% after 12 episodes (approximately 23,500 samples). However, this is worse as the 97.5% accuracy reached after 6,000 samples in the original paper. We observe that our model learns much slower than the author's one, and that even after long training the accuracy is inferior.

The difference in performance can be due to numerous factors. First, the update order of the network might be incorrect. Second, the stopping criteria could be implemented wrongly. Third, the time-to-first-spike translation might have been different. And lastly, we use a slightly different dataset, which might be harder to classify. However, since the MFSC features of that dataset do reach a 95% accuracy, it is unlikely that the different dataset causes a problem.

In terms of our learning goals, the project was a partial success. Re-implementation of the network definitely increased our understanding of the type of spiking network that can be used in ASR. In terms of our second learning goal, our visualisations provided us with a number of likely explanations for the differences in performance. However, based on our data, we can not identify a definite cause of why our network implementation does not perform as well as the original model.

References

[1] Meng Dong, Xuhui Huang, and Bo Xu. Unsupervised speech recognition through spike-timing-dependent plasticity in a convolutional spiking neural network. *PloS one*, 13(11):e0204596, 2018.

[2] John S Garofolo. Timit acoustic phonetic continuous speech corpus. *Linguistic Data Consortium, 1993*, 1993.

[3] Samanwoy Ghosh-Dastidar and Hojjat Adeli. Spiking neural networks. *International journal of neural systems*, 19(04):295–308, 2009.

[4] Md Amaan Haque, John Sahaya Rani Alex, and Nithya Venkatesan. Evaluation of modified deep neural network architecture performance for speech recognition. In *2018 International Conference on Intelligent and Advanced System (ICIAS)*, pages 1–5. IEEE, 2018.

[5] Ewa Jacewicz, Robert Allen Fox, and Lai Wei. Between-speaker and within-speaker variation in speech tempo of american english. *The Journal of the Acoustical Society of America*, 128(2):839–850, 2010.

[6] Biing-Hwang Juang and Lawrence R Rabiner. Automatic speech recognition–a brief history of the technology development. *Georgia Institute of Technology. Atlanta Rutgers University and the University of California. Santa Barbara*, 1:67, 2005.

[7] Chee Peng Lim, Siew Chan Woo, Aun Sim Loh, and Rohaizan Osman. Speech recognition using artificial neural networks. In *Proceedings of the First International Conference on Web Information Systems Engineering*, volume 1, pages 419–423. IEEE, 2000.

[8] Mohammed Kyari Mustafa, Tony Allen, and Kofi Appiah. A comparative review of dynamic neural networks and hidden markov model methods for mobile on-device speech recognition. *Neural Computing and Applications*, 31(2):891–899, 2019.

[9] Zihan Pan, Yansong Chua, Jibin Wu, Malu Zhang, Haizhou Li, and Eliathamby Ambikairajah. An efficient and perceptually motivated auditory neural encoding and decoding algorithm for spiking neural networks. *Frontiers in neuroscience*, 13:1420, 2020.

[10] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[11] Jibin Wu, Yansong Chua, and Haizhou Li. A biologically plausible speech recognition framework based on spiking neural networks. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2018.

[12] Jibin Wu, Yansong Chua, Malu Zhang, Haizhou Li, and Kay Chen Tan. A spiking neural network framework for robust sound classification. *Frontiers in neuroscience*, 12:836, 2018.

[13] Dong Yu and Li Deng. *Automatic Speech Recognition*. Springer, 2016.