Spoken digit classification SNN

Replicate the study by Dong et al within Nengo

Marije Bekema s1106132

Karan Chand s1033357

Freek Hens s1033509

Suzan Lejeune s1015354

January 29, 2023

Faculty of Social Sciences
Radboud University
The Netherlands

Neuromorphic Computing

Abstract

Research by Dong et al., 2018 showed a new model used to achieve spoken digit classification with the help of spiking neural networks. The major feature of the model is the combination of a feedforward spiking neural network in combination with spike-timing-dependent plasticity. In this way, the model is able to perform speech recognition with a high-performance score.-

During this research, the research from Dong et al. will be repeated in Nengo within the Python environment. The dataset that will be used is the same as the dataset in the Dong et al. experiment, namely the TIDGIT dataset. After this, the results will be evaluated based on the same parameters as Dong et al. by a linear classifier.

Due to limitations of Nengo, we were not able to train our model the same amount of data as Dong et al. and therefore our model was underfitted. The results reflect this by showing a low accuracy.

Keywords— speech recognition, spiking neural networks, spoken digit classification, neuromorphic computing, nengo

Contents

1 Specification **1**

2 Design **2**
2.1 Input encoding 2
2.2 Convolutional Layer 2
2.3 Local weight sharing 2
2.4 Learning weights with STDP 2
2.5 Pooling layer 2
2.6 Evaluation 3
2.7 Results 3

3 Implementation **4**
3.1 Pre-processing 4
3.2 Nengo 7
3.3 Classification 9

4 Testing **11**
4.1 Proposed testing 11
4.2 Separate testing 11
4.2.1 Pre-processing 11
4.2.2 Nengo learning & classification 11

5 Conclusion **13**
5.1 Future research 13

1 Specification

Research already showed the low power consumption of spiking neural networks, which is facilitated by the spiking pattern communication (Han and Roy, 2020). This spiking pattern communication is designed on neural mechanisms in the brain. In the brain, a neuron fires when its membrane potential reaches the threshold. This same mechanism is applied in spiking neural networks and provide a unique form of communication which can be combined with machine learning. Spiking neural networks could provide a solution for solving many problems. However, using these spiking patterns for speech recognition is not often used.

Speech recognition aims to provide methods which give computers the possibility to process spoken languages. One of the major examples of speech recognition are voice assistants such as Google Home or Siri. Usually, speech recognition is done automatically, which means that the system recognizes spoken languages and starts to process this spoken language into text. However, there are many factors that could influence the outcome. For example, different speakers leads to different voices and different speeds, but also background noise (Dong et al., 2018).

Different models have been proposed for using spiking neural networks and speech recognition. Some do not perform that well (Wu et al., 2018), where other perform well but are not biologically plausible (Dominguez-Morales et al., 2018; Wu et al., 2020). During this research the model from Dong et al. will be used. Their model consists of a feedforward spiking neural network with spike-timing-dependent plasticity, which is used to adjust the synaptic weights of convolutional neurons to form receptive fields in an unsupervised way. Also, a fast temporal encoding scheme is used. The model consists of a convolutional layer (which extract acoustic features from speech signals), and a pooling layer (performs a pooling operations to reduce the size of feature maps in the convolutional layer). The output of the pooling layer is used to train a linear classifier. Their model is evaluated in two ways. The first was with a support vector machine, which resulted in a performance of 97.5% and the second was a more difficult recognition task, which resulted in a performance of 93.8%.

During our research, the research from Dong et al. will be repeated. This means that we will use the proposed model for speech recognition in spiking neural networks. However, this will be implemented within Nengo. Nengo is a Python package which can be easily used for building, testing, and deploying neural networks. It can be used for spiking or traditional non-spiking models and it is fullyscriptable or one can use the GUI-based development. Lastly, one can tacle dynamic information processing (Nengo, w.d.).

2 Design

2.1 Input encoding

First of all, the data provided by Dong et al., 2018 needs to be pre-processed. In this step all data samples with different temporal lengths are converted into Mel-Frequency Spectral Coefficients (MFSCs) with the same lengths. The MFSC is different compared to MFCC, since MFSC preserves locality. The Mel-scaled filter is calculated by applying triangular filters on a Mel-scale to the power spectrum, and take the logarithm of the result. To get an input of fixed length a different window length is used within the Fourier transform step during the MFSC feature extraction.

The sensitivity of a single auditory neuron depends not only on frequency but also on time. This means that the neurons sensitive to the same frequency band may have different response latencies. To facilitate this feature, the neurons in the input layer are organized into a two-dimensional $M \times N$ array. Each row is a time frame and each column is a frequency band. In this way, the dynamics of a neuron depends on both the time frame and the frequency band.

The time-to-first spike coding scheme is used. Within this scheme, each neuron only needs to emit a single spike to transmit information during a presentation of an input sample. After this, a neuron will be shut off as soon as it fires a spike.

2.2 Convolutional Layer

The integrate-and-fire (IF) model needs to be implemented. For this model, the following rule is used:

$$\mathbf{V}(t) = \mathbf{V}(t-1) + \mathbf{W}^T * \mathbf{S}(t-1) \tag{1}$$

Where $\mathbf{V}$ is a vector of all neurons' membrane potential, $\mathbf{W}$ is the input synaptic weight matrix, and $\mathbf{S}$ is a vector which represents the spikes in the last time step. After each sample is processed, the membrane potential is reset to $\mathbf{V(t)} = 0$.

The convolutional layer consists of several sub-layers which can be seen as feature maps. The computation of neuronal potentials can be viewed as a convolution of the input signal with the shared weights. Weight sharing means that the same weights are used by many neurons in the same feature map.

2.3 Local weight sharing

Weights will be shared only among nearby convolutional neurons but will use separate sets of weights for different time periods of a spoken word, respectively. The local shared weights in the temporal domain are used for calculating the inputs of the corresponding convolutional neuron, and this strategy is called local weight sharing.

2.4 Learning weights with STDP

The weights of convolutional layer are initialized by drawing from a Gaussian distribution. When the network is in the training process, the weights are updated by the STDP rule. We use a simplified STDP within the model.

$$\Delta w_{ij} = \begin{cases} a^{+} w_{ij}(1-w_{ij}), & \text{if } t_j < t_i \\ -a^{-} w_{ij}(1-w_{ij}), & \text{elsewise.} \end{cases} \tag{2}$$

Where w is the weight of the synapse from the j^{th} neuron in the input layer to the i^{th} neuron in the convolutional layer, t_i and t_j are the corresponding firing time of two neurons, and a^{+} and a^{-} are the learning rates of STDP. The learning process will be ceased when the change of weight values is so small enough that they have no effect on the final network performance on the test dataset. After STDP is triggered on a neuron, the neuron in its neighborhood and all neurons at the same position in other feature maps are not allowed to perform STDP until the next sample appears.

2.5 Pooling layer

The pooling layer performs a pooling operation on the convolutional layer to reduce the dimension of the representation. Each feature map in the convolutional layer is processed independently, so the number of feature maps in the pooling layer is the same as that in the convolutional layer.

Each neuron in the pooling layer integrates inputs from one section in the corresponding feature map in the convolutional layer. The pooling neurons are not allowed to fire and their final membrane potentials are used as training data for a linear classifier. It should be noted that the pooling layer is not trained, and the value of its weights are fixed to 1. So the final potential of a pooling neuron can be seen as the spike number in its corresponding section and feature map in the convolutional layer. After the processing of each sample, the membrane potentials of pooling neurons are also reset.

2.6 Evaluation

The evaluation will be done with a linear classifier. First the SNN model is trained on the training set with STDP adjusting its synaptic weights. After the training is complete, we fix the weights by turning off the plasticity, run the network on the training set and train the linear classifier on the membrane potentials of the pooling layer and the corresponding labels.

Finally, we run the fixed network on the test set, and use the trained classifier to predict the labels to get the classification accuracy

2.7 Results

The results from Dong et al., 2018 used the utterances of female and male adults from the TIDIGITS dataset from Leonard et al., 1993, which includes more than 4000 samples from 200 speakers. The dataset was randomly ordered and split into training and test sets with the ratio 7:3. We use the same train and test split, because they were provided by the dataset.

- In the experiment, there were 40 neurons in the input layer.
- The convolution layer consisted of 50 feature maps, and the IF neurons in which had thresholds of 23.
- The convolutional window size was 6×40, which made it span all frequency bands, and its stride was 1.
- We divided the convolutional layer into 9 non-overlapping sections to share weights locally, and each section had a length of 4.
- The weights of convolutional layer were initialized with random values sampled from a Gaussian distribution with mean of 0.8 and standard deviation of 0.05.
- The learning rates of the STDP rule were a = 0.004 and a = 0.003.
- After the SNN was trained by STDP, the output of pooling layer was classified using a linear SVM.

3 Implementation

In this section, steps taken are explained in more detail. Code snippets accompany the text for clarity. The full code is available on Github (Hens et al., 2023).

3.1 Pre-processing

The audio files from the TIDGIT dataset (Leonard et al., 1993) are pre-processed by passing them through a MFSC filter. This step creates 41 frames with 40 frequency bands per sample. We used a public library to filter the data using MFSC (Hagens et al., 2021). We recreated a figure by Dong et al., 2018 in Figure 1. The figure shows in A the spectrogram of a raw input sample, where the label is "One". Because of the MFSC, all audio fragments are split into an equal amount of frames: 41, on the y-axis. The x-axis depicts the frequency bands (40) which also result from the MFSC filtering. Figure 1B shows the time-to-first-spike encoding for frame 17 from A. The y-axis shows the frequency bands (from A), and the x-axis shows the discrete time steps, in a range between 0 and 29, based on the paper by Dong et al., 2018. The time-to-first-spike encoding is computed by finding the the largest value in a frequency band per frame, and giving that frequency band the lowest time step. The second highest activation in figure A, spikes at a later time step, et cetera. The timesteps that correspond to the spikes need to be rounded, as the spikes are put into a new range. Rounding was performed using flooring. This design choice was made, as this was not specified by Dong et al., 2018.

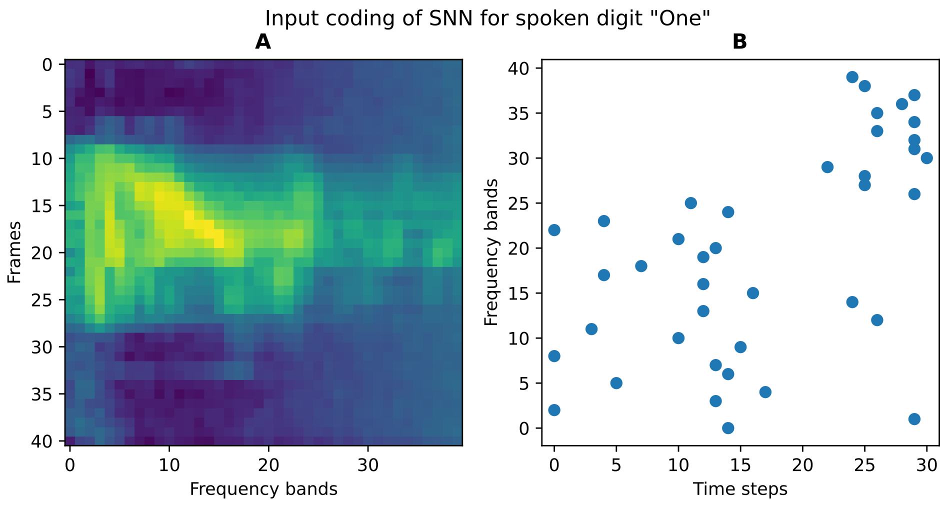

Figure 1: Defined frequency bands and time-to-first-spike encoding for the spoken digit "one", frame 17. *In this figure, part A shows the spectrogram of the frequencies for the input for the digit "One", with its frequency bands. In figure B, frame 17 of figure A is shown. The time-to-first-spike encoding is applied to the frequency bands. This is done for every frame of every sample.*

The code snippet below shows the way the train digits are filtered using MFSC using the library from Hagens et al., 2021. The integers for the `convert_tidigit_lib` are `samplerate, timeframes`, and `frequency bins`. These values are derived from the paper by Dong et al., 2018, as described in section 2.

```python
from mfsc import *
''' The data are loaded and passed through an MFSC filter: '''

[...]

digit_converter = TIDIGIT_Converter()

# Converting the train set:
results_lib_train_digit = digit_converter.convert_tidigit_lib(
    'data/TIDIGIT_train.mat',
    'train_samples', 20000, 41, 40)

[...]

# Saving files
handler = result_handler()
handler.save_file(
    'data/results_lib_train_digit.npy',
    results_lib_train_digit)

[...]
```

The sorting of frequency bands per filtered sample can be found below. `filtered_sample` contains the MFSC features from a particular sample. This sorting is done for every sample in the data set, before concatenating the sorted data. Below the code snippet, a visualisation of the concatenated data is shown in Figure 2. It provides an intuition for the encoding of the audio files.

```python
for nr_sample, filtered_sample in enumerate(tqdm(train_samples,
        desc='Computing sample')):

    [...]

    # computing spikes for each time frame
    idx = np.argsort(filtered_sample)
    x = np.atleast_2d(filtered_sample)
    transformed = np.zeros(x.shape)
    for i, row in enumerate(x):
        transformed[i] = (((row - np.min(row)) * (29 - 0))
        / (np.max(row) - np.min(row))) + 0

    # concatenating time frames into one spiketrain
    amt_timesteps = int(np.max(transformed[0])) #30
    amt_frames = transformed.shape[0] #41
    amt_fqbands = transformed.shape[1] #40

    c = 0 # for indexing spike_train
    t = 0 # for indexing x axis
    spike_train = np.zeros((2, amt_frames*amt_frames))

    for i, frequency_bands in enumerate(idx):
        spike_train[:, c:c+amt_fqbands] =\
            [transformed[i, :]+t, frequency_bands]
        c += amt_fqbands + 1
        t += amt_timesteps + 1

    # spike_train is the variable which stores a
    # spike train for a sample
    # spike_trains stores all spike_train samples
    spike_trains[nr_sample] = spike_train

spike_trains = np.floor(spike_trains)
```

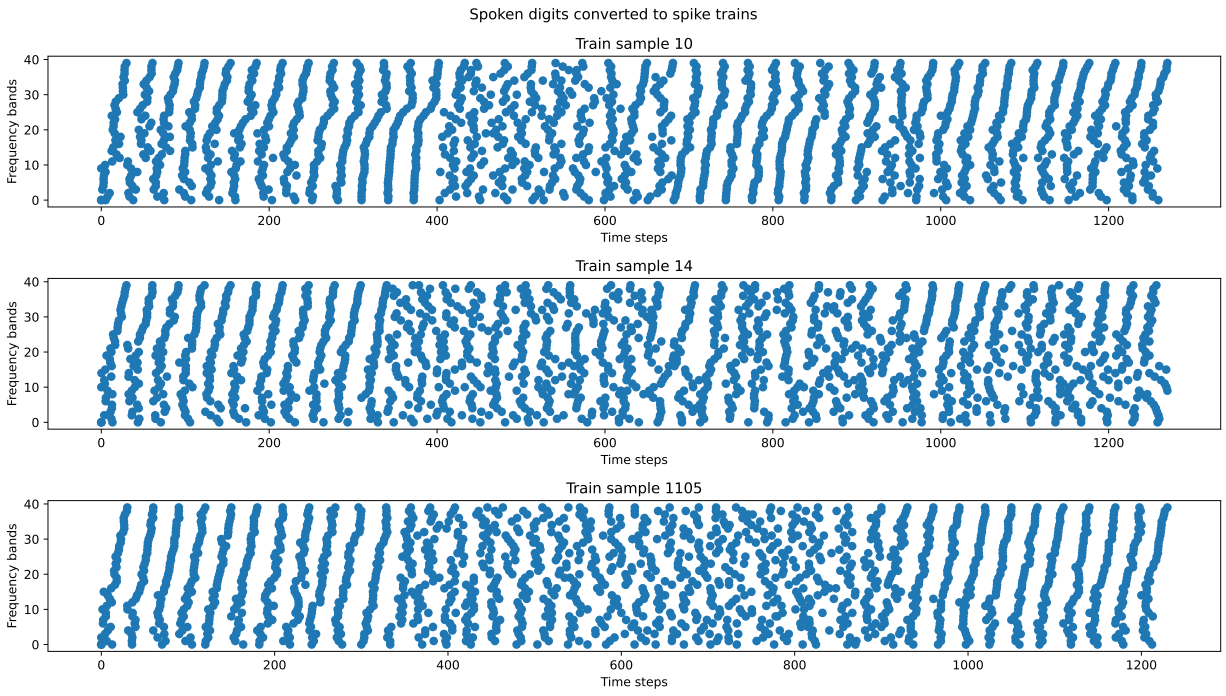

Figure 2: Spike trains of three training samples. *The spike trains for training samples 10, 14, and 1105 are shown. It is visible that in the middle of the spoken digit (the middle time steps), the time-to-first-spike for the frequency bands differs from the outer time steps. This corresponds to there being a silence in the audio file at the beginning and end of the spoken digit audio file.*

Now we have the MFSC filtered audio data stored in a time-to-first-spike encoding, but they are stored as $x, y =$ (timestep, frequency band) coordinates. But since Nengo can only process spike data when they are presented in binary, we converted the spike trains in the following snippet, as well as saving the file.

```python
for i, spike_train in enumerate(tqdm(spike_trains,
desc='Spike trains train')):
    spike_trains01 = np.zeros((amt_fqbands,
    amt_frames*amt_frames))
    for f_band in range(amt_fqbands):
        arr = np.array([i for i, x in
        enumerate(spike_train[1]) if x == f_band])
        idx = spike_train[0][arr]
        idx = idx[idx != 0]
        st = spike_trains01[f_band,:]
        idx = idx.astype(int)
        np.put(st, idx, 1)
        spike_trains01[f_band,:] = st
    all_spike_trains[i] = spike_trains01[:,0:30*41]

all_spike_trains = np.reshape(all_spike_trains,
(2464, 40, 41, 30))

# Saving files
handler = result_handler()
handler.save_file('data/spike_trains_train.npy', all_spike_trains)
```

Now the training set is converted to binary spike trains of shape (2464, 40, 41, 30), being the number of samples, frequency bands, timeframes and timesteps, respectively.

The same preprocessing is used for the (2486) test samples.

3.2 Nengo

Before setting up the Nengo environment, the data sets need to be created. For the dataset, the code from the pre-processing is used. After that code is executed, the following code will be executed.

```python
import nengo
import numpy as np
from sklearn.svm import LinearSVC
from skimage.measure import block_reduce
from sklearn.model_selection import train_test_split
from STDP_learning import STDP
from convolution import conv3D

# Parameters from the paper
n_freqbands = 40
n_timeframes = 41
n_timesteps = 30

n_neurons = n_freqbands * n_timeframes * n_timesteps

n_samples = 2464

f_maps = 50

window_size = [6,n_freqbands]
conv_inp_shape = (n_freqbands, n_timeframes, n_timesteps)
stride = [1,1]

mean = 0.8
std = 0.05

LIMIT = 500

presentation_time = 0.001

threshold = 23
thresh_config = nengo.presets.ThresholdingEnsembles(threshold) # Set the threshold
```

First, we create the parameters that were discussed in the paper by Dong et al., 2018 that will be used in the Nengo model. The values for the parameters in this implementation are equal to the values used by Dong et al., 2018.

```python
train_conv = np.reshape(train, (n_freqbands,n_timeframes,n_timesteps,n_samples))
train_new = []

for n in range(n_samples):
    train_new.append(conv3D(train_conv[:,:,:,n], np.array([[40,6]])))

train_new = np.array(train_new)
```

We attempted to do the convolution step within Nengo, but unfortunately there were some issues with incompatibility in shapes. Nengo is quite rigorous and does not allow much flexibility, so we were unable to resolve this. Instead, the convolution is taken using a helper function outside of Nengo. A convolution over the timeframes is taken with a width of six.

```python
transform = nengo.Convolution(
     n_filters = f_maps,
     input_shape = conv_inp_shape,
     kernel_size = window_size,
     strides = stride,
     padding="valid",
     channels_last = True,
     init = nengo.dists.Gaussian(mean, std)
 )
conv_conn = nengo.Connection(input_layer, pre, transform = transform)
```

Since the convolution is covered here, we will also show the proposed code for the convolution step as it would be implemented in Nengo here. Convolution in Nengo can be seen as a transformation over the data, therefore we can create a transformation between the input layer and the pre-learning node, where the convolution takes place in the same way as in the paper by Dong et al., 2018.

```python
train_new = np.reshape(train_new.T, (1600,n_samples))[:,0:LIMIT]
```

There were quite a lot of issues with Out-Of-Memory errors, most likely caused by the large, completely connected network that is created, since every sample is a neuron. This causes Nengo to create a matrix that is (1, n_samples * n_samples) large, since Nengo wants to have the input in a 1D array. This is problematic with memory, even when running it on an impressive online cluster, mlp10, which has 251 gigabytes of working memory.

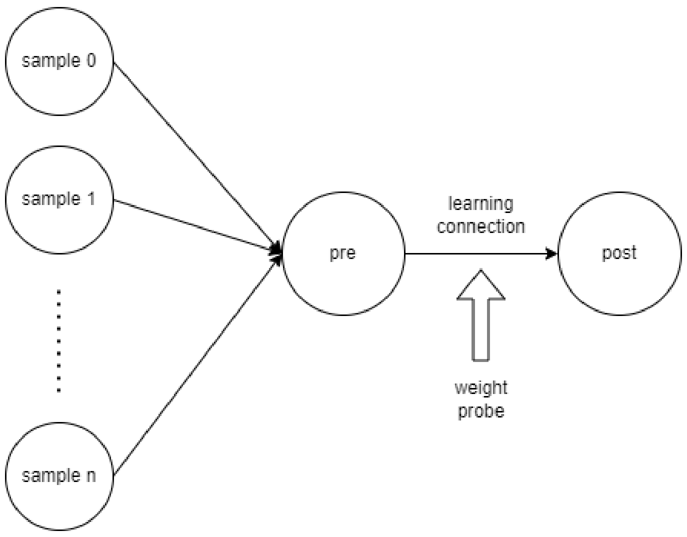

Figure 3: The architecture of the Nengo model.

```python
 model = nengo.Network(label="Audio STDP learning")

with model:

    # Layers ----------------------------------------------------------------------
    input_layer = nengo.Node(nengo.processes.PresentInput(train_new,
    ↪  presentation_time))

    pre = nengo.Ensemble(LIMIT, dimensions=LIMIT)

    post = nengo.Ensemble(LIMIT, dimensions=LIMIT)

    # Connections -------------------------------------------------------------

    learn_conn = nengo.Connection(
        pre, post,
        learning_rule_type = STDP(learning_rate=0.04),
        solver = nengo.solvers.LstsqL2(weights=True)
    )

    # Probes -----------------------------------------------------------------
    synapses_probe = nengo.Probe(learn_conn,"weights",label="synapses")

print("Running for {} seconds".format(len(train)*presentation_time))
with nengo.Simulator(model) as sim:

    sim.run(len(train)*presentation_time)
```

Now the Nengo model can be created using the architecture seen in Figure 3. Each sample is considered to be a neuron, as to get the original samples back individually. Once again, Nengo does not allow much freedom when defining a network forcing us to define the network like this. The samples are presented for a specific amount of time, depending on their length. With a presentation time of 0.001, the current code will present the samples for 2.5 seconds. The learning connection uses the STDP learning rule to update the weights. These weights are then probed and used as final results, rather than the output of the model, since this approach lets us retrieve the original samples.

```python
sample_output = sim.data[synapses_probe]
avg = sum(sample_output)/sample_output.shape[1]
pooled = block_reduce(avg, (4,1), np.max)
```

Finally, the probed information is pooled using the code above. There may also be a way to do this within Nengo, but such a way was not feasible in our case, since we take the data from the weights. Therefore, this approach to pool the data outside of Nengo was chosen.

3.3 Classification

To receive similar results as Dong et al., 2018 two kinds of classifiers can be used: linear SVM or spike-based tempotron. Both classifiers ensure to classify of the trained data with high accuracy and in a biologically realistic way. For simplicity, we chose a linear SVM classifier.

```python
X = pooled.T

y = np.load("train_labels.npy",allow_pickle=True)
y = y[0:LIMIT]
y_new = []
for i in range(len(y)):
    y_new.append(y[i][0][0][0])
y_new = np.array(y_new)

X_train, X_test, y_train, y_test = train_test_split(X, y_new, test_size=0.3)

X_train=X_train.astype('int')
X_test=X_test.astype('int')
y_train=y_train.astype('int')
y_test=y_test.astype('int')

clf = LinearSVC()
clf.fit(X_train, y_train)
print("Accuracy with sample size" + str(LIMIT) ": " + str(clf.score(X_test,
↪  y_test)))
```

The data is transformed into a format the classifier accepts and then the classifier is trained on a cut of the data. Finally, the accuracy of the model is retrieved by calculating the score. With the parameters shown above, the model has an accuracy of 0.067, which is very far off from the results from the paper of Dong et al., 2018.

4 Testing

The implementation of our model showed that the model cannot be tested as a whole. The reasons for these limitations will be discussed in this Section and will be summarized in Section 5. Although the implementation is restricted by some limitations, we will discuss the way in which the model can be tested. Besides that we will also show the results of testing our implementation on a smaller set of samples.

4.1 Proposed testing

Although our implementation cannot be tested on the whole data set, it is still relevant to discuss the way how the implementation can be tested. The TIDGIT data set consists of 4000 samples of speakers pronouncing digits in English. To replicate the research by Dong et al. Dong et al., 2018, our proposed testing consists of the same test and training size with a ratio 7:3. This means that the training set will consist of 2800 samples and the test set will consist of 1200 samples. First, the TIDGIT data set will be randomized, after this, the data set will be split into the training and test sets.

After defining the training and test sets, the model will be executed with the training set. This will define the optimal parameters for the model, which are used during the test set. During the test set the accuracy of the model will be evaluated with the help of a linear classifier. The expectations for an implementation that can be tested as a whole, are similar to the results shown by Dong et al., 2018. This means that the expected accuracy will be around 90%.

4.2 Separate testing

4.2.1 Pre-processing

When implementing the pre-processing code for creating spike trains from the MFSC filtered data, we recreated Figure 2 from Dong et al., 2018, as Figure 1.

We expected our Figure 1B of sorted spikes to look like the paper, but it does not. This is because the exact method of sorting was not specified in the paper, though the expected results were clear.

We think our results, shown in Figure 1B, represent the idea of a time-to-first-spike encoding like Dong et al. intended, even though the figures do not exactly match (see Dong et al., 2018). Figure 2 also makes intuitive sense, since the audio files are silent in the beginning and end, while the digit is spoken in the middle. This is visible by the time-to-first-spike encodings being different in the middle of the time steps compared to the start and end.

4.2.2 Nengo learning & classification

The final accuracy is a far cry from what was achieved by Dong et al., 2018, which confirms that there is some kind of error present, either conceptually or in the implementation. A large factor that may contribute to this is the memory error caused by Nengo. The way Nengo does its internal calculations makes it run out of memory when dealing with large networks. Switching to the deep learning variant of Nengo did not solve this. The only solution we were able to find was creating a smaller network, but this meant that the amount of samples is also substantially lower. Where the original paper used 4000 samples, we had 500 samples for the train and test set together. This makes it very likely that the final classifier is underfitted, on top of any other errors hiding in the current implementation.
This theory is supported by the following testing results:

n_samples	accuracy
200	0.033
300	0.044
400	0.059
500	0.067

Where we can see that as the amount of samples increases, the accuracy also increased. It is thus likely that at least some of the low accuracy can be explained as underfitting.
The convolution is taken outside of Nengo, over the entire data, which has a shape of (40, 41, 30, 2464), or (n_freqbands, n_timeframes, n_timesteps, n_samples). A frame of (n_freqbands, 6) is moved over the timeframe axis, as to take the convolution over that axis. After taking the convolution we end up with an array of shape (2464, 40, 40). We expected to have the n_freqbands dimension stay the same, but the n_timeframes dimension also changed. This may be a side effect of the convolution, as well as the

complete flattening of the third dimension, the n_timesteps. The effect of the convolution currently is not well known. Other ways of taking the convolution may be better, such as taking the convolution using Nengo.

The convolution proposed using Nengo in Section 3 was tested with a different network architecture. The relevant code is listed below.

```python
input_layer = nengo.Node(nengo.processes.PresentInput(train_new, presentation_time))

pre = nengo.Ensemble(250, dimensions=250)

post = nengo.Ensemble(250, dimensions=250)

[...]

transform = nengo.Convolution(
            n_filters = f_maps,
            input_shape = (10,10),
            kernel_size = [6],
            strides = [1],
            padding="valid",
            channels_last = True,
            init = nengo.dists.Gaussian(mean, std)
        )

conv_conn = nengo.Connection(input_layer, pre, transform = transform)
```

In this way, the convolution is correctly taken as a Nengo layer over 100 samples, which creates an output shape of (2464, 250, 250).

To test whether the current STDP learning rule is realisticly accurate, we will compare it with the BCM rule from Nengo. The BCM is a good comparison to test our STDP rule against since it can be an equivalent in specific situations, as shown by Pfister and Gerstner, 2006. When we compare the two learning rules for 500 samples, we get an accuracy of 0.067 for the BCM learning rule, and an accuracy of 0.053 for the STDP learning rule. While these accuracies are similar, and we can thus assume that no large differences are present between these learning rules, it should be noted that while the BCM learning rule only takes 17.2 seconds to simulate the model, the STDP learning rule takes 1 minute and 52 seconds.

Pooling is, like the convolution, done outside of Nengo, since Nengo does not allow much flexibility. The shape of the output before pooling is (500, 500), and after pooling the shape is (125, 500), thus successfully pooling the data along only one axis just as required.

5 Conclusion

The research performed by Dong et al., 2018 showed promising results to implement the design in Nengo as well. However, multiple factors contributed to the unfeasibility of implementing this design within the Nengo environment. These factors will be discussed in the conclusion. Besides that, room for future research will be discussed as well.

The pre-processing was easily doable. The digits are filtered using MFSC and a library and values are derived from the research performed by Dong et al. Within this part of the research, no limitations were found. This part of the project also did not need to be implemented within the Nengo environment and could be used as a separate Python script. The results of the pre-processing were promising and worked as expected, which is shown in Sections 3 and 4.

The unfeasibility of the project is caused within the Nengo environment. First of all the size of input neurons of the Nengo model is made of the shape 40×41 (each input neuron receives a frequency band and a frame). Besides that, each neuron also receives a timestep. Besides the fact that this shape is incredibly large, it is also 3D due to its three inputs. The result of this large model is that the simulation time is quite large. After this, the convolutional layer and the learning rule needs to be applied. This step is done in the simulation part of the Nengo environment.

Although the size of the model resulted in large building times, the actual learning of the model was the most unfeasible and not possible to implement within the Nengo environment. First of all, due to the size of the model, it was difficult to apply the convolutional layer. The Nengo environment required a specific size of the following layer, which was not specified according to the research performed by Dong et al., 2018. However, the code for the convolutional layer could be applied within Python, but not within the Nengo environment.

While slower than the BCM rule implemented within the Nengo environment, the STDP learning rule works properly within Nengo. The rule on its own works perfectly fine, however, again with the convolutional layer there are a lot of problems with the size of the model. Besides that, applying both the convolutional layer and the learning rule for a single sample (using a different structure than the one shown in this paper) creates really time-consuming simulations. On different computers, simulating the model results in simulations that take almost two hours. This approach is therefore not feasible for multiple samples, which is why the approach of this project is the way it currently is. However, besides time being an issue, there are also difficulties with memory. The model combined with the convolutional layer and the learning rule takes multiple gigabytes of memory.

Lastly, an important feature of the model proposed by Dong et al. (2018) could not be implemented. This is the local weight sharing between neurons. Nengo does not have a function or a possibility to implement a function that aims to reproduce this feature. Since this is an important part of the learning process, the model implemented in Nengo will not produce the same results as Dong et al. (2018).

Due to the time-consuming, memory-intensive characteristics, and features of this model within Nengo, the model is not feasible to implement. However, during our research, we found some interesting findings which could lead to better results in future research. These findings are explained in Section 5.1.

5.1 Future research

Due to the limitations of Nengo, we discovered the possibilities of NengoDL. NengoDL uses two different frameworks, namely: Nengo and Tensorflow. Tensorflow is used to perform different machine learning functions, whereas Nengo performs the neuromorphic modeling of the models. In this project the Nengo framework of NengoDL was explored as a way of overcoming Nengo's limitations, but unfortunately this did not resolve the issues present. However, the Tensorflow framework of NengoDL may have more success overcoming these issues. For example, multiple experiments were already performed in recognizing digits in images. However, it is important to note that the research performed by Dong et al. (2018) used spoken digits as their input values. More additional information about NengoDL can be found here Also, running the entire network on neuromorphic hardware is feasible due to the integration of Intel's Loihi with Nengo. This could further improve the energy efficiency of the model.

References

Dominguez-Morales, J. P., Liu, Q., James, R., Gutierrez-Galan, D., Jimenez-Fernandez, A., Davidson, S., & Furber, S. (2018). Deep spiking neural network model for time-variant signals classification: A real-time speech recognition approach. *2018 International Joint Conference on Neural Networks (IJCNN)*, 1–8.

Dong, M., Huang, X., & Xu, B. (2018). Unsupervised speech recognition through spike-timing-dependent plasticity in a convolutional spiking neural network [Retrieved 6-12-2022]. *PLOS One*. https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0204596

Hagens, M., Luttikholt, T., de Valk, T., & Schröder, P. (2021). Speech recognition with spiking nets. https://github.com/verrannt/snn_speechrec

Han, B., & Roy, K. (2020). Deep spiking neural network: Energy efficiency through time based coding. *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part X*, 388–404.

Hens, F., Chand, K., Lejeune, S., & Bekema, M. (2023). Spoken digit classificaion snn. https://github.com/freek1/neuromorphic-computing-project

Leonard, R. G., Doddington, & R., G. (1993). Tidigits. https://doi.org/10.35111/72XZ-6X59

Nengo. (w.d.). Nengo build brains [Retrieved 6-12-2022]. *Nengo Documentation*. https://www.nengo.ai/

Pfister, J.-P., & Gerstner, W. (2006). Triplets of spikes in a model of spike timing-dependent plasticity. *Journal of Neuroscience*, *26*(38), 9673–9682.

Wu, J., Chua, Y., & Li, H. (2018). A biologically plausible speech recognition framework based on spiking neural networks. *2018 International Joint Conference on Neural Networks (IJCNN)*, 1–8.

Wu, J., Yılmaz, E., Zhang, M., Li, H., & Tan, K. C. (2020). Deep spiking neural networks for large vocabulary automatic speech recognition. *Frontiers in neuroscience*, *14*, 199.