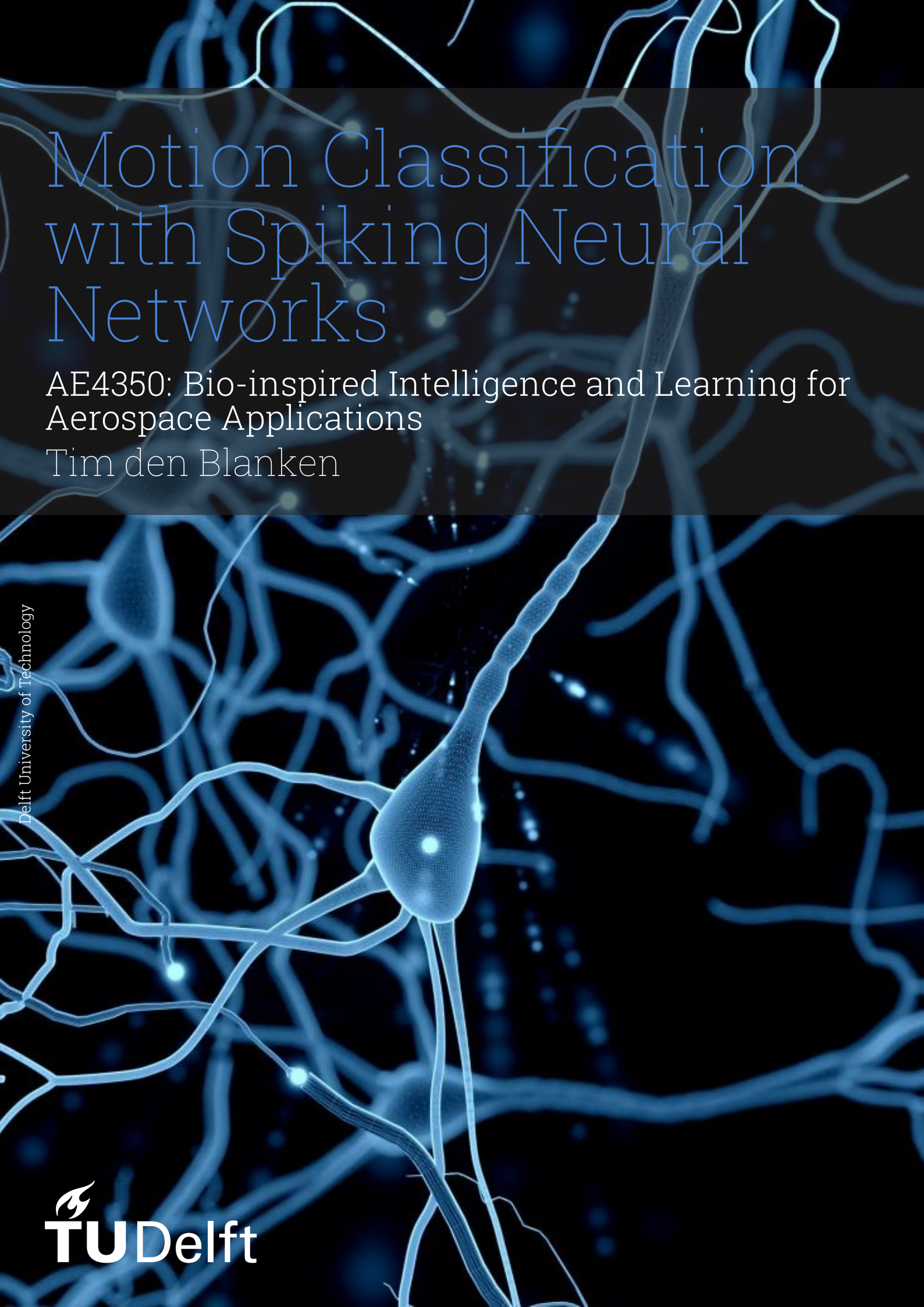

Motion Classification with Spiking Neural Networks

AE4350: Bio-inspired Intelligence and Learning for Aerospace Applications

Tim den Blanken

Delft University of Technology

TU Delft

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
ANN	Artifical Neural Network
SNN	Spiking Neural Network
CSNN	Convolutional Spiking Neural Network

1 Introduction

Spiking Neural Networks (SNNs) are increasingly gaining attention in the realm of artificial intelligence, particularly due to their potential to emulate the brain's natural processing capabilities more closely than traditional Artificial Neural Networks (ANNs) [5]. While ANNs have historically dominated the field (and still do), their reliance on continuous data and extensive computational resources has motivated the exploration of more biologically inspired alternatives like SNNs. Central in SNNs is the fact that information is encoded in the rate and or timing of the spikes, and not the shape of the spikes [4]. These discrete spikes allow for very high efficiency in both energy consumption and real-time processing, particularly when implemented in neuromorphic hardware. The pioneer in neuromorphic hardware is arguably the Intel Loihi chip [2], however in recent years many more neuromorphic (research) processors are entering the market. Unfortunately, this project will not make use of such neuromorphic processor, and instead will solely be based in software.

A significant advancement in the domain of neuromorphic computing is the development of event-based cameras, which operate fundamentally different from conventional frame-based cameras. Instead of capturing continuous streams of images at fixed time intervals, event-based cameras detect changes in the visual scene asynchronously, triggering events at a per-pixel basis only when a brightness change occurs [3]. This results in a sparse, highly efficient representation of motion, which aligns perfectly with the spike-based processing paradigm of SNNs.

The integration of event-based data with SNNs is particularly impactful in scenarios where real-time, low-power processing is critical. For instance, the recent work on fully neuromorphic vision and control for autonomous drone flight [8] has demonstrated the feasibility and advantages of such an approach. Utilizing event-based cameras and SNNs running on Intel's Loihi neuromorphic processor, the system efficiently processes visual data to control drone movements, achieving tasks like hovering and landing with precision and minimal power consumption. This project serves as a strong inspiration for the current project, where a convolutional spiking neural network (CSNN) will be developed to classify

First in chapter 2 the method used is explained, i.e. what data is generated and how the models are created. Then in chapter 3 the results are listed, which are then discussed in chapter 4. Finally, a conclusion is presented in chapter 5. Appendix A contains all graphs supporting the results.

All code, which is referenced throughout these chapters, can be found in GitHub:

- https://github.com/Timdnb/csnn-motion-classification.

2 Method

This chapter covers the method taken to train CSNNs such that they are capable of classifying motions. It starts at the data generation in section 2.1, after which the model setup and training are explained in

section 2.2.

2.1. Data generation

The first choice to be made is what type of data will be used, and what implications that will have on the CSNNs. In this project the choice has been made to assume a setup in which a moving device (e.g. a drone, or RC car) has a camera pointing straight down to the ground. The video of the ground can then be used to classify four planar directions and a rotation. For this, either an existing dataset can be used, or the data can be artificially generated. The choice has been made for the latter, as this allows to adjust the data to the project's needs. Generally, this is bad practice, as ideally one would use real-world data and train a model that can handle that. However, for the scope of this project, which is more the investigation of CSNNs and their implications rather than creating models that can work in reality, it makes more sense to use artificially generated data. An additional benefit is that an unlimited amount of data can be generated, giving the theoretical possibility of training infinitely.

The next question is what a single data sample (i.e. a short video of multiple frames) should look like. Known is that the classes to be created should have the labels ”up”, ”down”, ”left”, ”right” indicating the planar directions and ”rotation” for, well, the rotation. To keep it simple, this will be achieved by drawing a specific shape that undergoes this motion. The shapes used for this are circles and squares. To get an idea of what this looks like, a single frame of these motions can be seen in Figure 2.1, however for the full video please head over to the ‘dataset_showcase.ipynb‘ notebook in GitHub, run it and check out Figure 2.1 there.

Figure 2.1: Single frame of squares with different motions

After creating these short videos, they need to be converted to event-based data, since this is what will be used as input to the CSNN. This is a simple process where continuously two consecutive frames are used to calculate the events. Again, a single frame of these event-based videos can be found in Figure 2.2. A red pixel indicates a decrease in brightness while a green pixel indicates an increase in brightness. For the video, again head over to the ‘dataset_showcase.ipynb‘ notebook in GitHub, run it and check out Figure 2.2 there.

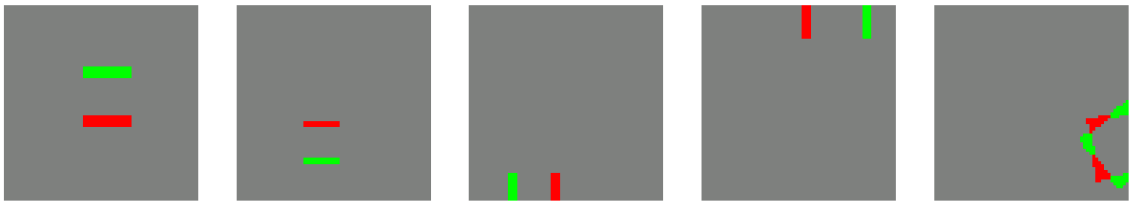

Figure 2.2: Single event-based frame of squares with different motions

Apart from the shapes (circles and squares), also a third category has been added: noise. Noise is basically a frame where every pixel is randomly 0 or 255. This frame then undergoes a specific motion. Once again, to get a better understanding of all data samples, have a look at the ’dataset_showcase.ipynb‘ notebook in GitHub. This notebook calls functions from the file ‘utils.py‘. This file contains all the specifics of how the data is created and under what circumstances. With the data creation process in place, it is time to move over to the models that will be trained.

2.2. Models

As mentioned before, in this project convolutional spiking neural networks (CSNNs) will used to classify five different motions. This choice has been made as traditionally convolutional neural networks are great at processing images. But now instead of a single image, a motion, i.e. consecutive frames, needs to be classified. This calls for a model with temporal dependence, which is what a spiking neural network has. Combining the two leads to the CSNN as in this project. Of course other architectures (both non-spiking and spiking) could be used and compared, however the choice has been made to stick to a single architecture and investigate the effect of changes within the architecture. The coming sections explain the considerations and general steps taken. For the exact implementation in code please look at the GitHub repository (section 6 in 'CSNN_training.ipynb').

2.2.1. Setup

The snnTorch framework made by Jason K. Eshraghian in combination with PyTorch Lightning [1] makes for a fairly simple implementation of CSNNs. To begin with. there is no need to use any spike encoding (e.g. rate or latency coding), as the generated data is already a spike encoded, i.e. the events. The next step is the architecture of the model. This is where a lot of testing comes into play, as there is an enormous amount of parameters that defines an architecture. Finding a balance, and with that a well-performing model can be rather complicated.

First the number and type of layers need to be chosen. Generally, more layers lead to better performing models, but they can also be more prone to overfitting. Another downside of having many layers is that inference becomes more expensive, which defeats one of the main purposes of using neuromorphic algorithms in the first place: their efficiency. So with this in mind and after testing to get an initial idea of performance, the following architecture has been chosen:

1. The first layer is a 2D convolutional layer with 1 input and 16 output channels, a kernel size of 3 and 'same' padding
2. Then a leaky layer follows, which uses a first-order leaky integrate-and-fire neuron model. Beta, the membrane potential decay rate, is set to 0.95
3. Next a max pooling layer with kernel size and stride 2 is added
4. Again a 2D convolutional layer is added with 16 input and 32 output layers
5. Followed by another leaky layer with beta equal to 0.95
6. And again a max pooling layer with kernel size and stride 2
7. Next is a fully connected layer with five output channels
8. Followed by the final layer, which is a leaky layer with beta 0.95 again

This architecture boasts a relatively small number of layers, while still maintaining performance, as can be seen in the results. From now on, the number and type of layers will stay fixed, to limit the number of possibilities to investigate. The parameters within the layers will be investigated further in a sensitivity analysis later in the report. The architecture is also pictured in Figure 2.3.

The square at the top of the figure is the input frame, which by default is 64 by 64 pixels. A black pixel has a value of -1 (negative spike) and a white pixel has a value of +1 (positive spike). The other squares indicate the leaky layers, where a white pixel indicates a neuron that fired. Please note that the image depicts a single frame of video. These videos are by default 16 frames in length.

A forward pass in this model comes down to feeding the frames of the video one by one, until the video is over. The outputs of the final leaky layer are added over time, and the neuron that fired the most is then the final output of the model. This output is compared to the actual motion to evaluate the performance.

Another interesting point that has not been discussed yet is population coding. In the brain, single neuron messages are ambiguous, so to fight this the brain tries to resolve the ambiguity by using multiple neurons [7]. This has also been implemented in the snnTorch framework, and as such it is easy to add to the current architecture. In the architecture above, setting a population code other than 1 will increase the number of output channels of the fully connected layer to the population code times

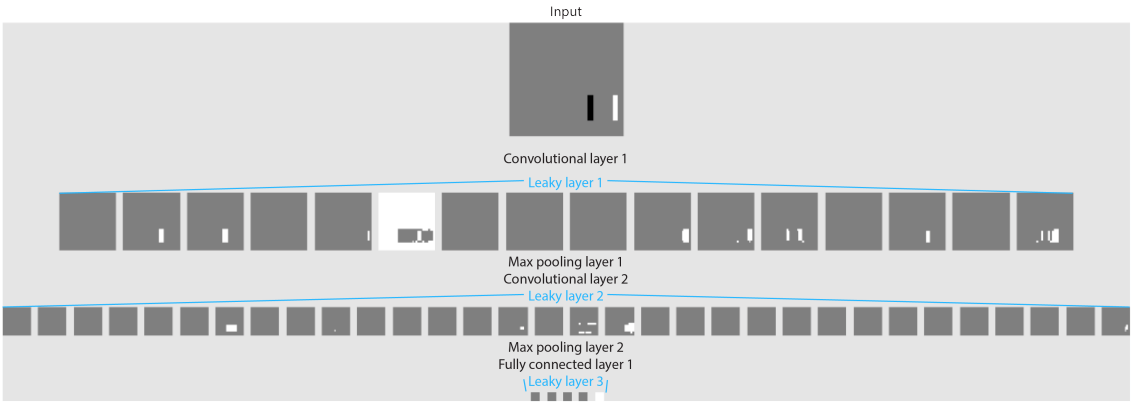

Figure 2.3: Convolutional spiking neural network architecture

five. These outputs are then sent to the final leaky layer, which now also consists of more neurons. Finally, the output for a single class is calculated by adding the outputs of all neurons for that class. The prediction then corresponds to the class with the highest output, i.e. with this most neurons fired. By default, population code is set to 1.

2.2.2. Training

Training of a CSNN follows the same philosophy as for a standard ANN: do a forward pass, calculate the loss and change the parameters to minimize the loss. Yet, spiking neural networks are fundamentally different and do require different algorithms to be trained. How the training is implemented under the hood in PyTorch Lightning and snnTorch can be found in their excellent documentation. Important to know is that the leaky layers in the CSNN are optimized using surrogate gradients [6].

2.2.3. Parameters and logging

This section serves as an overview of all parameters that can be changed within the model or the dataset, together with the values that will be investigated as part of a parameter analysis. The parameter name corresponds with the actual parameter name in the code. The default value for each parameter is the value that is used in and for the baseline model. This will serve as a comparison ground when changing parameters. The overview for model parameters that can be changed can be found in Table 2.1 and for dataset parameters that can be changed in Table 2.2

Table 2.1: Changeable parameters for CSNN

Parameter	**What is it?**	**Default Value**	**Possible values**
conv_layers.output_channels	Number of output channels for each convolutional layer	(16, 32)	(8-128, 8-128)
conv_layers.kernel_sizes	Size of kernel for each convolutional layer	(3, 3)	(3-7, 3-7)
max_pool_layers.kernel_sizes	Size of kernel for each max pooling layer	(2, 2)	(1-4, 1-4)
leaky_layers.betas	Beta value for each leaky layer	(0.95, 0.95, 0.95)	(0.8-1, 0.8-1, 0.8-1)
leaky_layers.learn_betas	Whether to make beta a learnable parameter in each leaky layer	(False, False, False)	(True/False, True/False, True/False)
population	Population code in final leaky layer	1	1-100

Table 2.2: Changeable parameters for dataset used to train CSNN

Parameter	What is it?	Default Value	Possible values
n_samples	Number of samples in train set	32000	500-100000
frame_size	Size of the frame	64	16-128
n_frames	Number of frames in single sample	16	8-128
shapes	Type of shapes in dataset	circles and squares	circles, squares and noise

To keep track of all parameters and especially see the effect of the change of parameters, a logging dashboard has been setup. This will track all performance statistics of the models and allows for an easy comparison. Weights and Biases [9] has been used for this, and the dashboard can be found at the following link:

- https://wandb.ai/timdb/CSNN-motion-classification.

3 Results

Section 3.1 contains all results obtained with the CSNN architecture as presented in Figure 2.3. Then in section 3.2 the performance of the CSNN with changing parameters is listed.

3.1. Baseline CSNN

Figure 3.1 and Figure 3.2 show the accuracy and loss respectively during training. A single training step (x-axis) is a batch of 32 samples. Note that there are no epochs, this is because the dataset can be infinitely generated, and thus an epoch does not exist. When more data is needed, more can be generated.

Figure 3.1: Training accuracy of baseline model

Figure 3.2: Training loss of baseline model

For validation and test performance see Table 3.1.

Table 3.1: Validation and test performance of baseline model

	Accuracy	Loss
Validation	0.9744	0.0842
Test	0.9784	0.0819

3.2. Parameter analysis

The coming subsections contain tables that show the test and training accuracies and losses for different parameter changes. In subsection 3.2.1 this is done for model parameters, while subsection 3.2.2 shows the effects of changes in dataset parameters. All plots accompanying this data (including training progression over time) can be found in Appendix A.

3.2.1. Model parameters

Convolutional layers output channels

Table 3.2: Accuracy and losses for varying number of output channels per convolutional layer. Run name should be interpreted as follows: (x,y)-output-channels means that the first convolutional layers had x output channels and the second convolutional layer had y output channels.

	Test		Train	
Run name	**Accuracy**	**Loss**	**Accuracy**	**Loss**
(8,8)-output-channels	0.9822	0.0487	0.9375	0.1150
(16,16)-output-channels	0.9813	0.0519	0.9688	0.0509
baseline	0.9784	0.0819	0.9784	0.0932
(32,64)-output-channels	0.9581	0.1998	0.9375	0.1982
(32,32)-output-channels	0.9434	0.1625	0.9063	0.3175
(8,16)-output-channels	0.8628	0.4628	0.9063	0.3227
(64,128)-output-channels	0.6363	0.9083	0.5625	0.8748

Convolutional layers kernel sizes

Table 3.3: Accuracy and losses for varying kernel sizes for each convolutional layer. Run name should be interpreted as follows: (x,y)-kernels means that the first convolutional layer had a kernel size of x and the second convolutional layer had a kernel size of y.

	Test		Train	
Run name	**Accuracy**	**Loss**	**Accuracy**	**Loss**
(7,3)-kernels	0.9863	0.0560	1.0000	0.0160
(7,5)-kernels	0.9828	0.0406	0.9828	0.0837
baseline	0.9784	0.0819	0.9784	0.0932
(5,3)-kernels	0.9775	0.0601	0.9775	0.0571
(7,7)-kernels	0.9772	0.0568	0.9772	0.1216
(5,5)-kernels	0.9628	0.1138	0.9628	0.1831

Max pooling layers kernel sizes

Table 3.4: Accuracy and losses for varying kernel sizes per max pooling layer. Run name should be interpreted as follows: (x,y)-maxp-kernels means that the first max pooling layer had a kernel size and stride of x and the second max pooling layer had a kernel size and stride of y.

	Test		Train	
Run name	**Accuracy**	**Loss**	**Accuracy**	**Loss**
(4,4)-maxp-kernels	0.9919	0.0307	1.0000	0.0200
baseline	0.9784	0.0819	0.9784	0.0932
(4,2)-maxp-kernels	0.9778	0.0544	0.9688	0.1007
(1,1)-maxp-kernels	0.7438	0.8525	0.8438	0.7779

Leaky layers beta values

Table 3.5: Accuracy and losses for varying beta values for leaky layers. Run name should be interpreted as follows: beta-x means that the value for beta for all leaky layers is equal to x.

	Test		Train	
Run name	**Accuracy**	**Loss**	**Accuracy**	**Loss**
beta-0	0.9934	0.0241	1.0000	0.0223
beta-0.25	0.9925	0.0190	1.0000	0.0110
beta-0.5	0.9916	0.0220	0.9688	0.0684
beta-0.75	0.9841	0.0394	0.9688	0.0286
beta-0.9	0.9834	0.0484	1.0000	0.0191
baseline	0.9784	0.0819	0.9784	0.0932
beta-1	0.9691	0.1912	1.0000	0.1620

Leaky layers learnable betas

Table 3.6: Accuracy and losses when using learnable betas versus the baseline, where betas are fixed at 0.95.

	Test		Train	
Run name	**Accuracy**	**Loss**	**Accuracy**	**Loss**
learnable-betas	0.9928	0.0203	1.0000	0.0022
baseline	0.9784	0.0819	0.9784	0.0932

Population codes

Table 3.7: Accuracy and losses for different population codes. Run name should be interpreted as follows: population-x means that a population code of x has been used in the model.

	Test		Train	
Run name	**Accuracy**	**Loss**	**Accuracy**	**Loss**
population-5	0.9822	0.0568	1.0000	0.0041
population-20	0.9806	0.0805	1.0000	0.0323
population-10	0.9803	0.0523	1.0000	0.0001
baseline	0.9784	0.0819	0.9784	0.0932
population-50	0.9625	0.1106	1.0000	0.0218
population-100	0.8672	0.4085	0.90625	0.3674

3.2.2. Dataset parameters

Number of samples

Table 3.8: Accuracy and losses for number of samples in dataset. Run name should be interpreted as follows: xk-samples means that x times 1000 number of samples has been used to train the model.

	Test		Train	
Run name	**Accuracy**	**Loss**	**Accuracy**	**Loss**
100k-samples	0.9935	0.0154	1.0000	0.0517
50k-samples	0.9904	0.02493	1.0000	0.0120
10k-samples	0.9860	0.0567	0.9688	0.0448
baseline	0.9784	0.0819	0.9784	0.0932
5k-samples	0.9620	0.1469	0.9375	0.1443
2k-samples	0.7000	0.7114	0.5000	1.1365
0.5k-samples	0.3200	1.5329	0.2188	1.6220
1k-samples	0.3100	1.4971	0.2500	1.3597

Frame sizes

Table 3.9: Accuracy and losses for different frame sizes. Run name should be interpreted as follows: frame-size-x means that the dataset contained samples with frame size x times x.

	Test		Train	
Run name	**Accuracy**	**Loss**	**Accuracy**	**Loss**
frame-size-16	0.9838	0.0322	1.0000	0.0001
frame-size-32	0.9838	0.0359	1.0000	0.0035
baseline	0.9784	0.0819	0.9784	0.0932
frame-size-128	0.7169	0.7447	0.7813	0.7568

Number of frames

Table 3.10: Accuracy and losses for different number of frames per samples. Run name should be interpreted as follows: x-frames means that a single sample in the dataset consisted of x frames.

	Test		Train	
Run name	**Accuracy**	**Loss**	**Accuracy**	**Loss**
8-frames	0.9894	0.0492	0.9688	0.0592
baseline	0.9784	0.0819	0.9784	0.0932
32-frames	0.9125	0.5567	0.8125	0.6638
128-frames	0.4922	0.9089	0.5000	0.8328
64-frames	0.4447	1.3268	0.3438	1.5565

Shapes in dataset

Table 3.11: Accuracy and losses for different shapes during training and testing. Run name should be interpreted as follows: train-x-test-y means that the model has been trained on x (where x can be s for square, c for circle or n for noise) and tested on y (where y can be s for square, c for circle or n for noise).

	Test		Train	
Run name	**Accuracy**	**Loss**	**Accuracy**	**Loss**
train-s-test-s	0.9888	0.0289	1.0000	0.0128
train-c-test-c	0.9881	0.0388	1.0000	0.0274
baseline	0.9784	0.0819	0.9784	0.0932
train-c-test-s	0.5347	3.1265	1.0000	0.0144
train-s-test-c	0.3438	3.4689	0.9375	0.1020
train-n-test-n	0.2013	1.5024	0.1875	1.5038
train-sc-test-n	0.1994	12.0094	0.9688	0.1248
train-n-test-sc	0.1941	1.6747	0.1875	1.6094

4

Discussion

The results obtained about the performance of CSNNs in classifying motions together with the results obtained by varying numerous parameters allows for an insightful discussion. Section 4.1 covers the

performance of the baseline model, while section 4.2 covers the effects of changing parameters and discusses possible reasons for them.

4.1. Baseline CSNN

The baseline convolutional spiking neural network demonstrated strong capabilities in classifying short event-based videos of planar motions and rotations. This follows from the fact that the model has a test accuracy of 97.84% with a test loss of 0.0842 (Table 3.1). This performance is confirmed when investigating the spiking behaviour (last cell in the CSNN_training.ipynb notebook in GitHub). Here one can see that for nearly every frame of each sample, the correct neuron spikes.

Since motions can only be identified if the model contains a temporal dependence, and since the leaky layers are the only parts of the model with temporal processing capabilities, it is clear that the model's effectiveness can be attributed to the leaky layers with spiking neurons.

4.2. Parameter analysis

Arguably more interesting insight will follow from the results of varying parameters. Each of the parameters will be covered below:

1. **Convolutional layers output channels:** the models with fewer output channels performed better, where the one with the least (i.e. (8,8) channels) performed best with an accuracy of 98.22%. This could be explained by the simplicity of the data (just squares or circles, no complex shapes), where more channels only increase computational complexity and or increase overfitting.
2. **Convolutional layers kernel sizes:** having a larger kernel size in the first convolutional layer seems to improve performance a little, since the top two models both have a kernel size of 7 in the first layer. This suggests that the model benefited from capturing larger spatial features first. Still performance across all runs is quite good, so its seems that the kernel size is not a too important parameter in this case.
3. **Max pooling kernel sizes:** here it is apparent that larger kernel sizes lead to better performance, with the best-performing model having a kernel size of 4 for both layers. This makes sense as the data contains simple and fairly big shapes, so reducing this quickly without losing much information is beneficial to the model. However when investigating training progression (Figure A.9) it appears that with more training steps, the model with (1,1) kernels could catch up. This shows that having larger kernels leads to faster learning.
4. **Leaky layers beta values:** a clear trend is visible where lower beta values perform better. A lower beta value corresponds to a smaller membrane potential decay rate. This suggests that having no decay in the neurons is better for this model. This could be explained by the fact that the motion only spikes neurons a couple of times, and once the shape has passed there will not be any other spikes. This means that the model needs to work with little spikes, and thus it appears to be more beneficial to not decay the membrane potential. One interesting point when looking at the training performance with beta equal to 0 (Figure A.13): there is a clear sudden drop in accuracy, which might indicate instability when using this beta value. This is something to watch out for.
5. **Leaky layers learnable betas:** it is clear that having learnable betas improves the model performance. This also makes sense considering the previous point, as it is now clear that a beta value of 0.95 (which is what the baseline has) is not very good. When turning on learnable betas the model can improve on this value, and indeed when checking the values after training they drop, although interestingly not to 0.
6. **Population coding:** when using a small population code the model improves in performance, but when using a large population code it decreases again. This seems to indicate that having a couple of neurons per class indeed removes some ambiguity in the neuron spiking, but having too many complicates it too much.
7. **Dataset parameters:** the first three dataset parameters, i.e. "n_samples", "frame_size" and "n_frames", basically serve as a sanity check. One would expect the model performance to increase when trained on more samples; this is indeed what happens. Similarly smaller frame

sizes and less frames (both equate to less data) should be easier to classify. This is also the case when checking the accuracies and losses. The final dataset parameter ”shapes” is more about the generalizability of the model, and is covered in the next part.

8. **Shapes in dataset:** training and testing on different shapes shows the generalization capabilities of the model. However when investigating the numbers, it is clear that it has very poor generalizability. Accuracies between 97% and 99% are only achieved when training and testing on the same shapes. When training on circles and testing on squares, an accuracy of 53.47% is achieved, which is definitely better than chance (i.e. 20%), but it is still pretty bad. When using noise samples, performance is quickly degraded to chance levels. This seems to indicate that the model is not complex enough to capture motion from more complicated frames of data.

5 Conclusion

In conclusion, CSNNs have proven to be effective in classifying short event-based videos of motion when leveraging their temporal processing capabilities. The parameter analysis underscores the importance of carefully tuning model and dataset parameters to optimize performance. The baseline model performed well, but many different parameters changes lead to a model that was better than the baseline model. The model did show very poor generalization properties, and was unable to classify any motion when using more complex videos than a simple shape moving across the screen. Increasing the number of layers could resolve this issue. With this change or other improvements, a final conclusion suggests that CSNNs are well-suited for tasks involving event-based data and motion classification, with potential applications in areas requiring real-time, low-power processing.

References

[1] PyTorch Lightning creators. *PyTorch Lightning*. URL: https://lightning.ai/docs/pytorch/stable/.

[2] Mike Davies et al. “Loihi: A Neuromorphic Manycore Processor with On-Chip Learning”. In: *IEEE Micro* 38.1 (2018), pp. 82–99. DOI: 10.1109/MM.2018.112130359.

[3] Guillermo Gallego et al. “Event-Based Vision: A Survey”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.1 (Jan. 2022), pp. 154–180. ISSN: 1939-3539. DOI: 10.1109/tpami.2020.3008413. URL: http://dx.doi.org/10.1109/TPAMI.2020.3008413.

[4] Wulfram Gerstner and Werner M. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.

[5] SAMANWOY GHOSH-DASTIDAR and HOJJAT ADELI. “SPIKING NEURAL NETWORKS”. In: *International Journal of Neural Systems* 19.04 (2009). PMID: 19731402, pp. 295–308. DOI: 10.1142/S0129065709002002. eprint: https://doi.org/10.1142/S0129065709002002. URL: https://doi.org/10.1142/S0129065709002002.

[6] Emre O. Neftci, Hesham Mostafa, and Friedemann Zenke. “Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-Based Optimization to Spiking Neural Networks”. In: *IEEE Signal Processing Magazine* 36.6 (2019), pp. 51–63. DOI: 10.1109/MSP.2019.2931595.

[7] Stefano Panzeri et al. “Population Coding”. In: *Analysis of Parallel Spike Trains*. Ed. by Sonja Grün and Stefan Rotter. Boston, MA: Springer US, 2010, pp. 303–319. ISBN: 978-1-4419-5675-0. DOI: 10.1007/978-1-4419-5675-0_14. URL: https://doi.org/10.1007/978-1-4419-5675-0_14.

[8] Federico Paredes-Vallés et al. *Fully neuromorphic vision and control for autonomous drone flight*. 2023. arXiv: 2303.08778 [cs.RO]. URL: https://arxiv.org/abs/2303.08778.

[9] Weights and Biases creators. *Weights and Biases*. URL: https://wandb.ai/site.

A

Parameter change results

This appendix contains all plots that accompany the data as presented in chapter 3.

A.1. Model parameters

A.1.1. Convolutional layers output channels

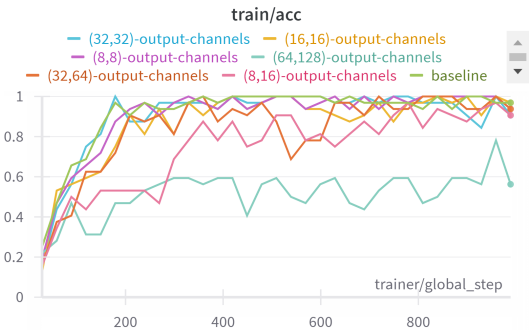

Figure A.1: Training accuracy of models with varying number of output channels per convolutional layers

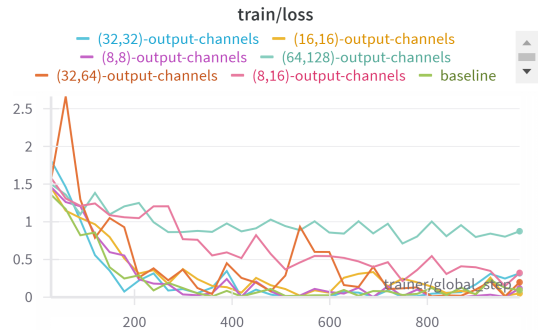

Figure A.2: Training loss of models with varying number of output channels per convolutional layers

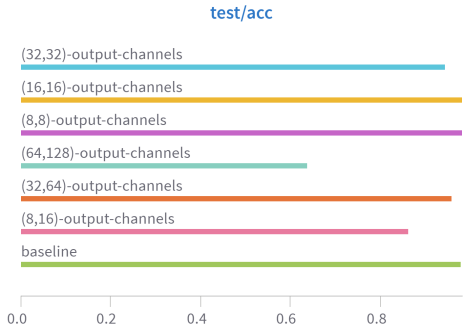

Figure A.3: Test accuracy of models with varying number of output channels per convolutional layers

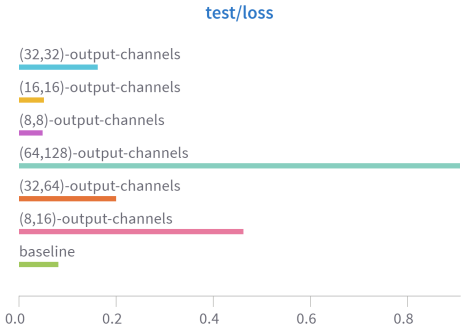

Figure A.4: Test loss of models with varying number of output channels per convolutional layers

A.1.2. Convolutional layers kernel sizes

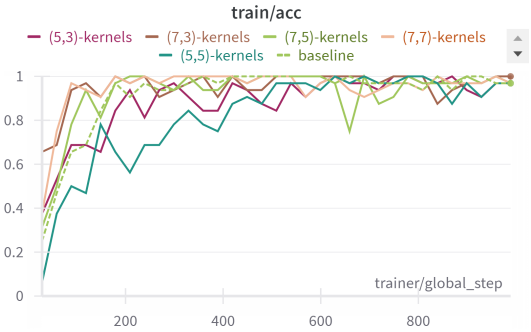

Figure A.5: Training accuracy for varying kernel sizes for each convolutional layer

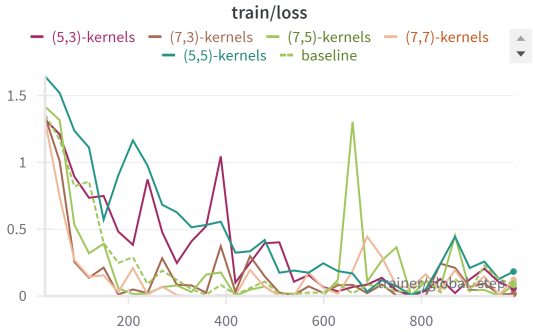

Figure A.6: Training loss for varying kernel sizes for each convolutional layer

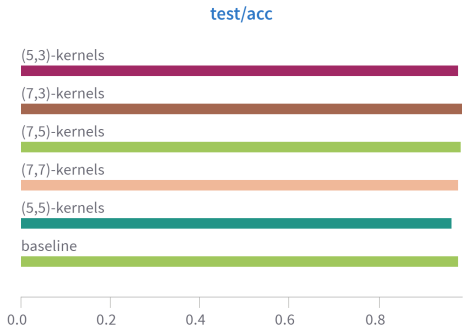

Figure A.7: Test accuracy for varying kernel sizes for each convolutional layer

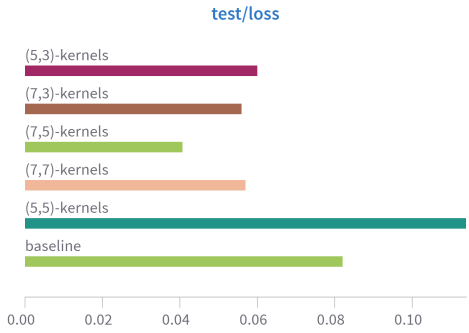

Figure A.8: Test loss for varying kernel sizes for each convolutional layer

A.1.3. Max pooling layers kernel sizes

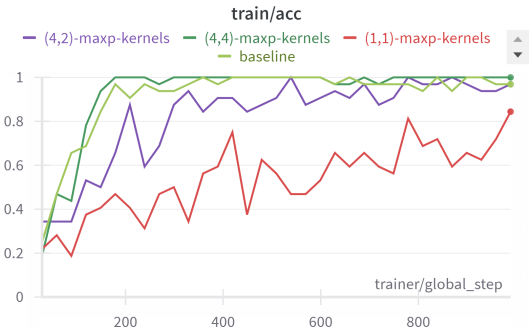

Figure A.9: Training accuracy for varying kernel sizes per max pooling layer

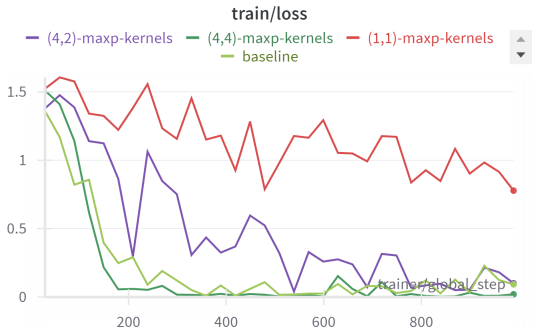

Figure A.10: Training loss for varying kernel sizes per max pooling layer

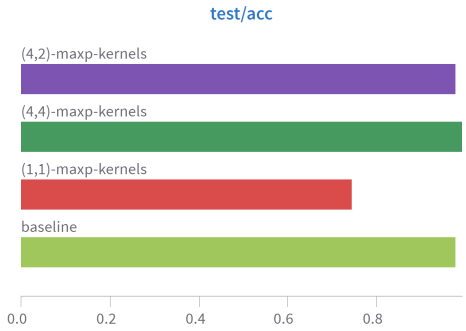

Figure A.11: Test accuracy for varying kernel sizes per max pooling layer

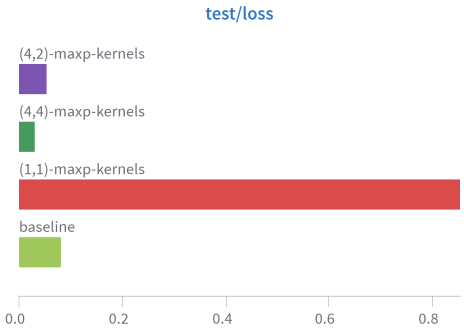

Figure A.12: Test loss for varying kernel sizes per max pooling layer

A.1.4. Leaky layers beta values

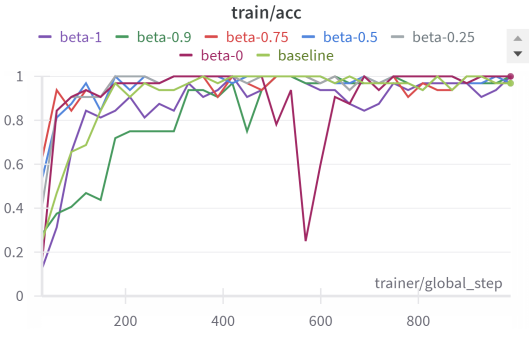

Figure A.13: Training accuracy for varying betas for leaky layers

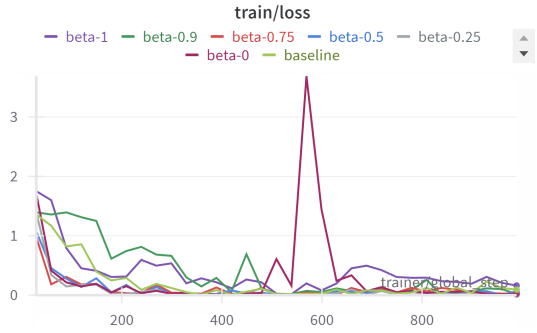

Figure A.14: Training loss for varying betas for leaky layers

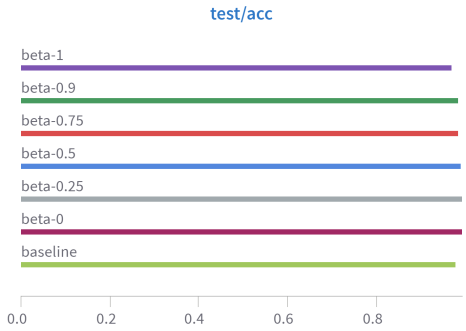

Figure A.15: Test accuracy for varying betas for leaky layers

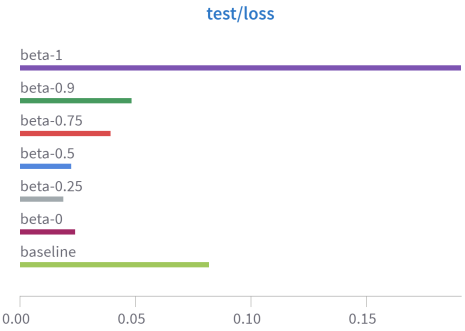

Figure A.16: Test loss for varying betas for leaky layers

A.1.5. Leaky layers learnable betas

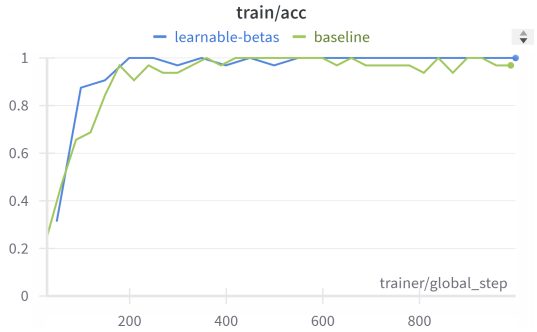

Figure A.17: Training accuracy when using learnable betas

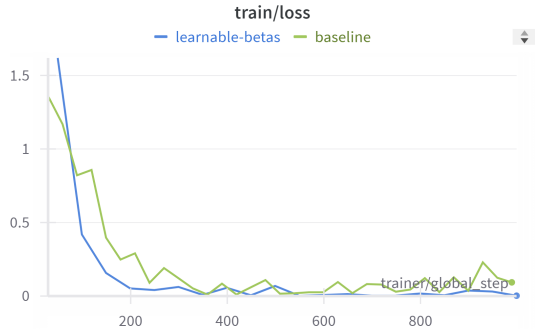

Figure A.18: Training loss when using learnable betas

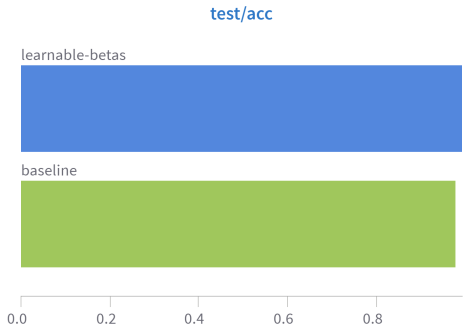

Figure A.19: Test accuracy when using learnable betas

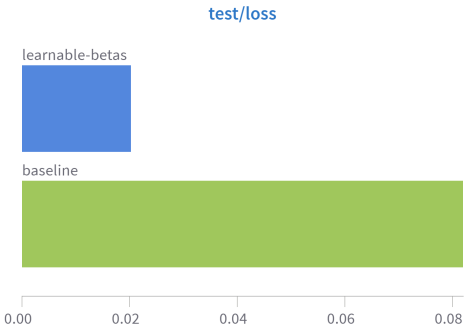

Figure A.20: Test loss when using learnable betas

A.1.6. Population codes

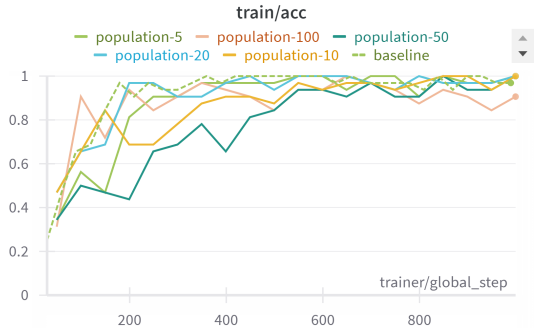

Figure A.21: Training accuracy for different population codes

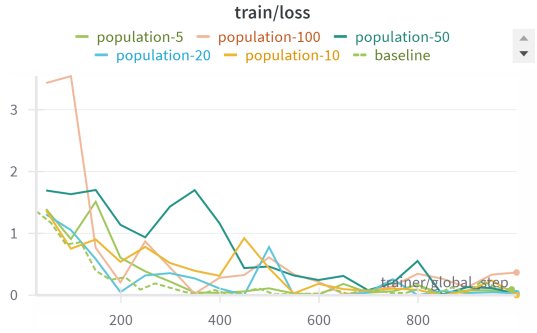

Figure A.22: Training loss for different population codes

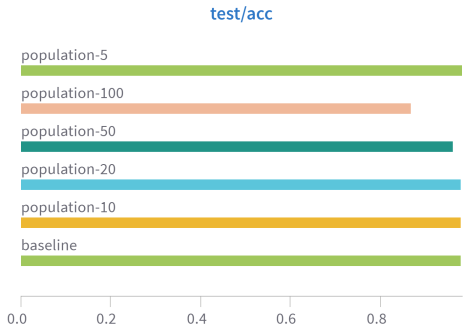

Figure A.23: Test accuracy for different population codes

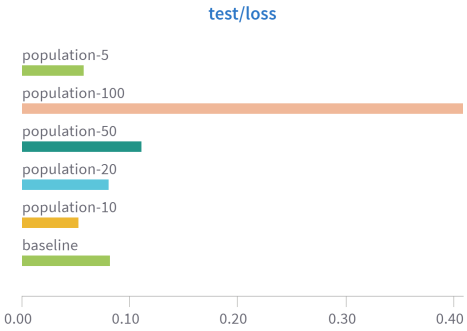

Figure A.24: Test loss for different population codes

A.2. Dataset parameters

A.2.1. Number of samples

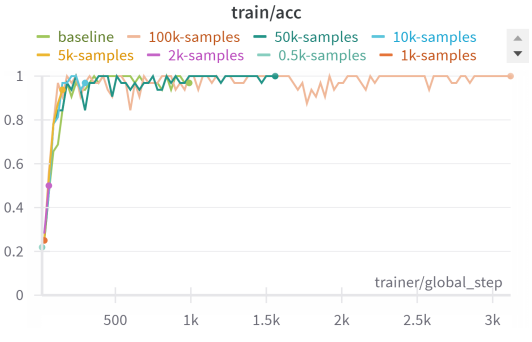

Figure A.25: Training accuracy for number of samples in dataset

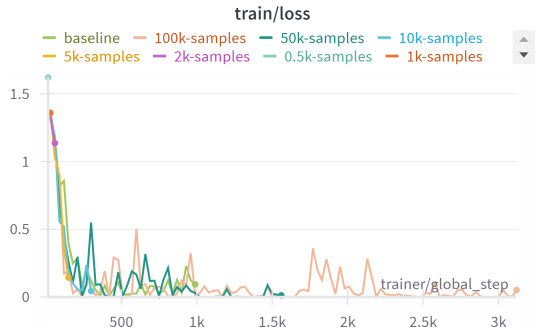

Figure A.26: Training loss for number of samples in dataset

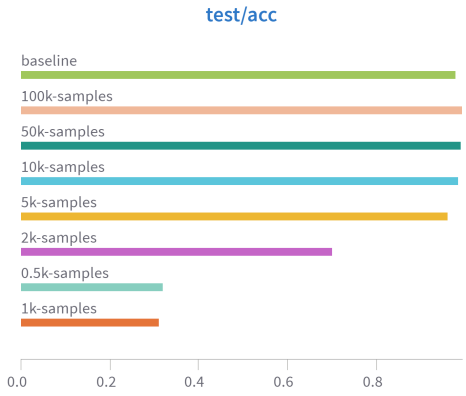

Figure A.27: Test accuracy for number of samples in dataset

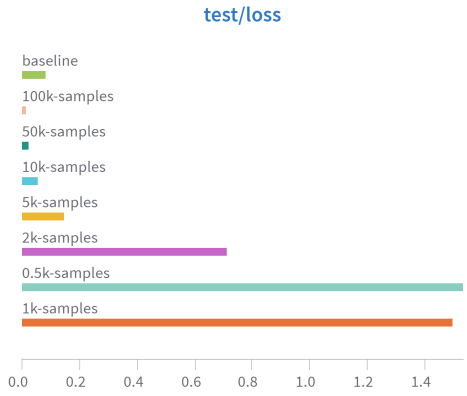

Figure A.28: Test loss for number of samples in dataset

A.2.2. Frame sizes

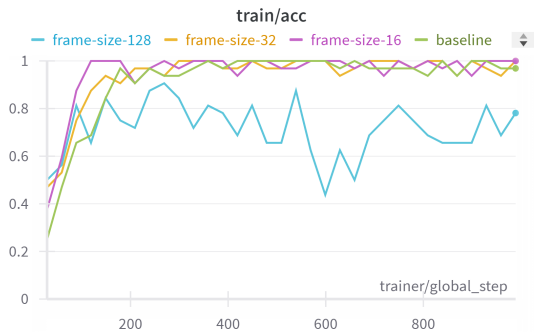

Figure A.29: Training accuracy for different frame sizes

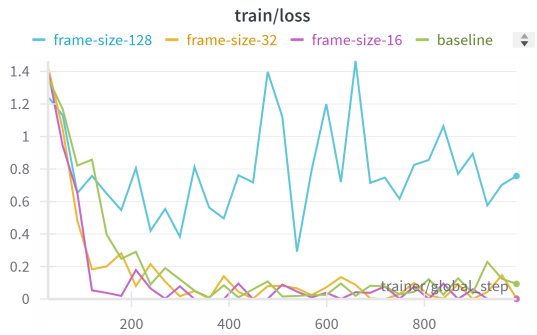

Figure A.30: Training loss for different frame sizes

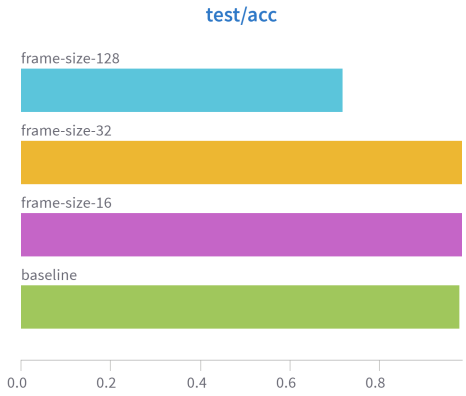

Figure A.31: Test accuracy for different frame sizes

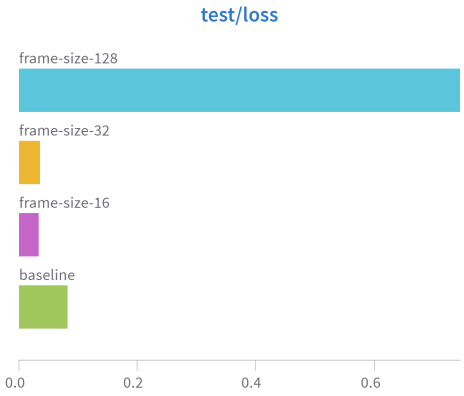

Figure A.32: Test loss for different frame sizes

A.2.3. Number of frames

Figure A.33: Training accuracy for different number of frames per samples

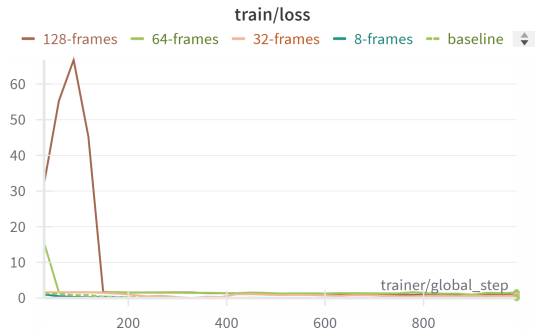

Figure A.34: Training loss for different number of frames per samples

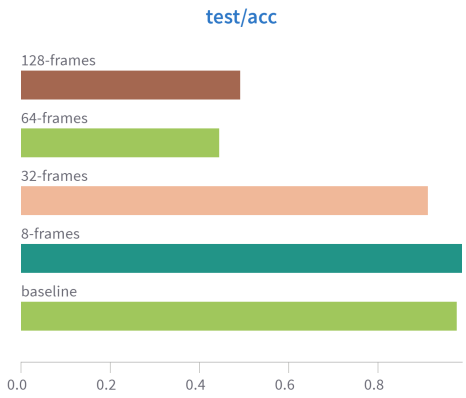

Figure A.35: Test accuracy for different number of frames per samples

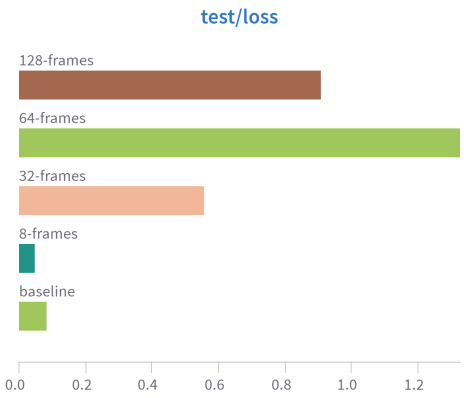

Figure A.36: Test loss for different number of frames per samples

A.2.4. Shapes in datasets

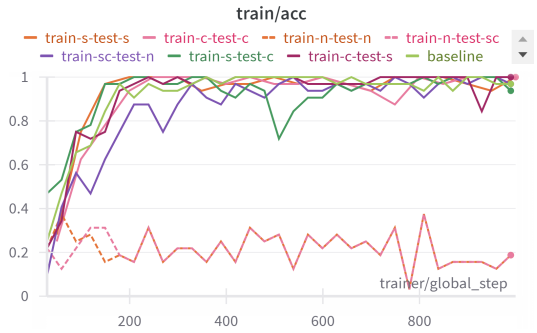

Figure A.37: Training accuracy for different shapes during training and testing

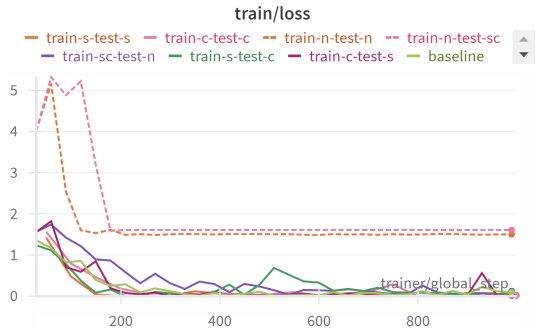

Figure A.38: Training loss for different shapes during training and testing

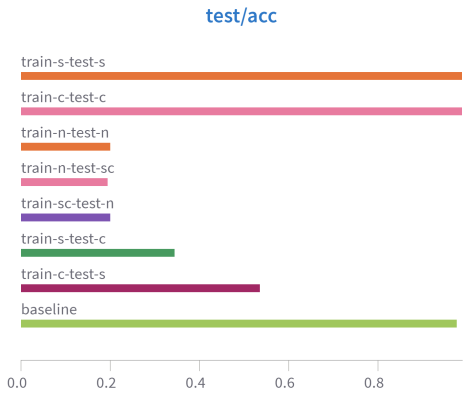

Figure A.39: Test accuracy for different shapes during training and testing

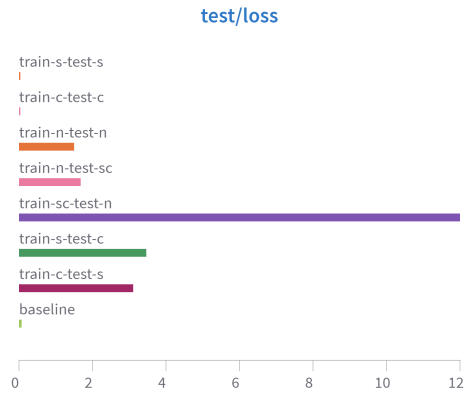

Figure A.40: Test loss for different shapes during training and testing