

Supplementary material

A Experimental Detail

Scheduling for $\Sigma\Delta$ -TDAM model All the models in Tab.2 are trained for the same number of epochs shown in Tab.1. The $\Sigma\Delta$ -TDAM model was initialized using parameters from *Dense* models trained for (epoch/2) epochs shown in Tab.1, weight and bias are fixed and then trained for another (epoch/2) epochs to optimize the quantization step size using layer-wise optimization described in [49]. For TDAM weight scheduling, we utilize the same scheduling with our TDSS, and the same random sampling technique for TDSS discussed in Sec. 3.2.2 is also used for a fair comparison.

A.1 PilotNet Steering Angle Prediction (frame)

Dataset The PilotNet dataset [51] consists of a 10Hz video sequence captured by a camera mounted on a car and the corresponding steering angles. Following the code accompanied with the data, we randomly split the sequence into a train and test set with a ratio of 8:2. We computed the MAC from the first 2,800 frames following the procedure of [69]. To evaluate the efficiency in the higher frame-rate scenario, we used Super-sloMo [51] with their provided pre-trained model to up-sample the sequence into 120 Hz and 480 Hz. The accuracy metric for this dataset is mean squared error (MSE) in degrees. Steering angle label larger than 2π are clamped to $[-2\pi, +2\pi]$.

Network In this experiment, we used a 10-layer feed-forward CNN proposed in [9]. We adopted the base network from [9] designed for this task. The input size of the network is $3 \times 66 \times 200$. The network configuration is 24C5-36C5-48C5-64C3-64C3-1164FC-100FC-50FC-10FC-1FC. The first two layers use a stride of two without padding and the last two use stride of one without padding. All the conv layer use ReLU as non-linearity. All the FC layers except the last layer also use ReLU, and the final layer uses arctan, and the result is doubled following the procedure of [51].

Dynamic weighting for TDSS In this experiment using the PilotNet dataset, we used the dynamic weighting for TDSS loss to avoid the excessive restriction of TDSS for samples undergo large motion. The difference between consecutive frames varied across samples; therefore, suppressing the TDSS with the same weight for these samples may excessively restrict the change in activation for samples with a significant difference. We applied adaptive scaling to mitigate this issue using the difference of output $|y_t - y_{t-1}|$. Specifically, we used the following scaling:

$$\hat{\eta}_t = \eta \frac{T|y_t - y_{t-1}|_2}{\sum_{t'=1}^T (|y_{t'} - y_{t'-1}|_2)}, \quad (17)$$

where T is the total number of training frames and $y(t)$ is steering angle at time t , and η is TDSS weight scheduled using validation data (Sec. 4.1).

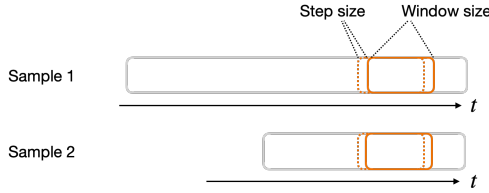


Figure A1: **Sliding window estimation used in N-MNIST dataset.** For the N-MNIST dataset, the output histogram is computed in a sliding window manner. Each event sample contains a different number of events. Therefore, the number of accumulation depends on the length (number of events) of a data sample.

A.2 N-MNIST Digit Recognition (event)

Dataset The N-MNIST dataset [47] consists of MNIST images converted into an event-stream using an event-based camera moving on a pan-tilt unit mimicking saccadic motion. The training and testing separation are the same as the standard MNIST split of 60,000 training samples and 10,000 testing samples. The data was not stabilized for a fair comparison with SLAYERS [59]. This dataset’s accuracy metric is classification accuracy. We used testing samples without sliding window (Fig. A1) as validation data and testing samples with the sliding window as test data.

Network In this experiment, we used a 3-layer feed-forward CNN. Input size of the network is $2 \times 34 \times 34$. The network configuration is 32C5-P2-64C5-P2-16C3-1024FC-10FC. All the layers use a stride of one with padding so that the spatial dimension does not change between input and output. All the conv layer use ReLU as non-linearity. All the FC layer except the last layer also use ReLU and the last layer use softmax.

Estimation by sliding window The output from the network is accumulated in a sliding window manner (in step size of 100) within each dataset sample before computing class prediction. This is illustrated in Fig. A1. This strategy is also applied for *Dense*, *Asyc-SSC* [42], and $\Sigma\Delta$ -TDAM [46]. The results without sliding windows are 0.7-1.0% worse than those with sliding windows in all configurations.

More results In addition to the method shown in the main paper, we also report an additional comparison with computationally light models DART [54] and the SOTA binary SNN model SNN-BP [57] and SLAYERS [59]. We did not have a chance to measure MAC for these methods; ours showed superior or comparable accuracy to these methods, DART (accuracy: 98.0%) and SNN-BP (accuracy: 98.7%) and SLAYERS (accuracy: 99.2%).

A.3 N-Caltech101 Object Recognition (event)

Dataset The N-Caltech101 dataset [47] is a converted event-stream from Caltech101 [59] using a method similar to N-MNIST. It contains 8,246 event samples in total, which are labeled 101 classes. The accuracy metric for this dataset is classification accuracy.

Network The base network uses VGG13 [60] as the backbone, followed by an FC layer that outputs a 101-dimensional vector. Input size of the network is $2 \times 191 \times 255$. The network comprises three VGG conv blocks followed by two conv with a kernel size of 3 outputting 128-dimensional feature, followed by FC to output 101 classes. In this experiment, we followed the testing method of [42] and used the first 25,000 events from each data sample, without accumulation by sliding windows, as in the case of N-MNIST.

For Asyc-SSC [42] we utilized their published code⁴, only modifications from original code is optimizer (we used AdamW [41] instead of Adam [63]), batch size and a number of training epochs which is summarized in Tab.1. The results, including the dense VGG model, shown in Tab.2 were slightly lower than the reported accuracy in [42].

More comparison In addition to the method shown in the main paper, we made an additional comparison with computationally light models HOTS [65], HATS [60] and DART [64], and CNN-based method called YOLE [6], which also utilized recursive update. Ours achieve considerably better MAC/accuracy trade off than HOTS (accuracy: 0.210%, MAC: 27MOPS for 1 event), HATS (accuracy: 0.642%, MAC: 2.2MOPS for 1 event), DART (accuracy: 0.664%), and YOLE (accuracy: 0.702% , MAC: 1,830 MOPS for 1 event)

A.4 Gen1 Autom. Object Detection (event)

Dataset Gen1 automotive dataset [63] contains 228,123 bounding boxes for cars and 27,658 pedestrians collected from an event-based camera mounted on a car. The accuracy metric is mean average precision (mAP) [13]. The base network uses VGG13 [60] as the backbone, and the last layer is followed by the YOLO header [65] to generate bounding boxes and class predictions.

Network Input size of the network is $2 \times 223 \times 287$. The network comprised of three VGG Conv block followed by a conv with a kernel size of 3 which output 256-dimensional feature; that is processed by two FC output $6 \times 8 \times 2 \times 2$ dimensional vector; that is passed to the YOLO [65] header to output bounding-boxes and class prediction.

For Asyc-SSC [42] we utilized their published code⁵, modifications from original code is optimizer (we used AdamW [41] instead of Adam [63]), batch size and number of training epochs which is summarized in Tab.1. The results, including the dense VGG model, shown in Tab.2 were slightly lower than the reported accuracy in [42].

⁴https://github.com/uzh-rpg/rpg_asynet

⁵https://github.com/uzh-rpg/rpg_asynet

B Learning Dynamics

Fig. A2 shows the transition of parameters during training on the PilotNet and N-MNIST experiment conducted in the main experiment. As expected, we observed that as the TDSS loss decreased, the MAC also decreased, and the quantization step size s and sparsity of synaptic connection increase correspondingly.

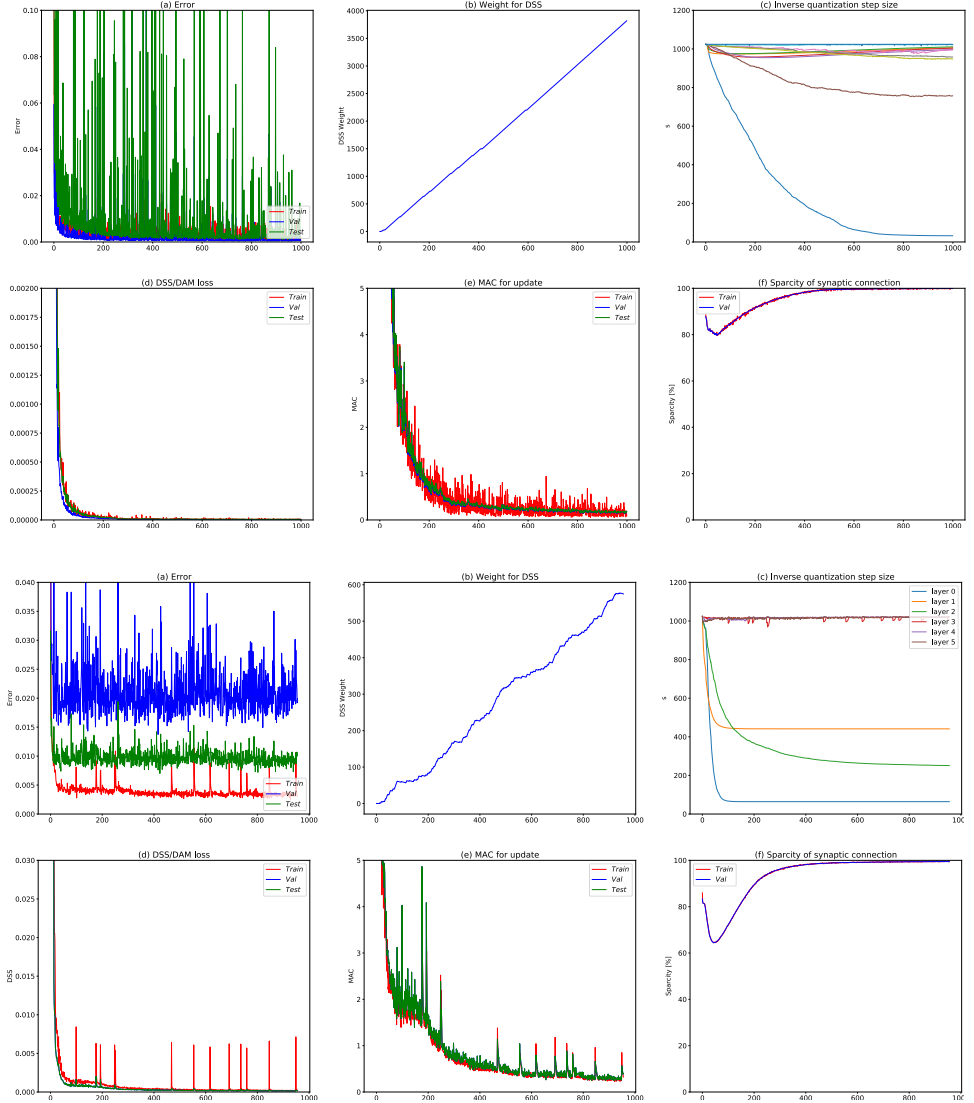


Figure A2: The evolution of parameters during training. PilotNet (Top), N-MNIST (Bottom). The horizontal axis is the number of learning epochs. $\kappa = 8$ for both experiments.

C Sparsity of Synaptic Connection

Tab.A1 shows the number of synaptic connections of the network used in the main and ablation experiments in Sec. 4. As shown in Tab.2, ours achieve lower MAC when the network has more learnable parameters. We hypothesized this is because our $m\Sigma\Delta$ network trained with TDSS learns to select the computational path by activation that triggers neurons in the following layers sparsely. And when the network has more parameters, it could represent a sparser path that results in a lower MAC. There are other possibilities based on the lottery thicket hypothesis [14]. Given this hypothesis, the networks having more trainable parameters won the lottery to realize sparser synaptic connections achieving the same accuracy.

Results in Tab.A1 summarize synaptic connection after training (sparse SC) for each network. We observed that networks that achieved lower MAC (larger learnable parameters) still had more synaptic connections after training. This fact partially supports the hypothesis that the $m\Sigma\Delta$ network trained with TDSS selects computational paths that require less MAC by activation. More in-depth analysis will be future work.

Table A1: **Sparsity of synaptic connection in different network size.** Each row corresponds to our $m\Sigma\Delta$ -TDSS model having κ times larger input/output channel dimension in mconv layer (except FC layer). Columns are defined as follows: weight is number of trainable weight for mconv, dense SC is number of synaptic connection when all the masks m are 1, and sparse SC is number of synaptic connection after training.

	PilotNet			N-MNIST		
	weight	dense SC	sparse SC	weight	dense SC	sparse SC
(x1)	1.59E+06	2.82E+07	1.44E+06	7.24E+04	1.72E+07	3.44E+06
(x2)	1.91E+06	9.62E+07	1.85E+06	2.37E+05	6.41E+07	4.72E+06
(x4)	3.10E+06	3.56E+08	2.27E+06	8.73E+05	2.47E+08	4.75E+06
(x8)	7.69E+06	1.37E+09	3.18E+06	3.37E+06	9.67E+08	5.07E+06
	N-Caltech 101			Gen1 Automotive		
	weight	dense SC	sparse SC	weight	dense SC	sparse SC
(x1)	3.69E+06	8.09E+08	5.60E+08	2.81E+07	1.09E+09	7.23E+08
(x2)	4.87E+06	2.67E+09	1.29E+09	2.93E+07	3.55E+09	1.48E+09

D Locally Connected Network

Locally Connected Network (LCN) was invented before CNN [16] has arrived on the scene. Inspired by the organization of the visual cortex [28, 29], several early neural networks consisted of locally connected neurons [15, 66] without spatial parameter sharing. After the successful combination of CNN and backpropagation [56], CNN was considered to generalize better than LCN [2]. Low-Rank Locally Connected (LRLC) layer [10], a low-rank version of LCN, was proposed recently, and they demonstrated that it performed better than CNN. From now on, both will be referred to as LC together; in particular, the former is called Full-Rank LC (FRLC), and the latter is called LRLC.

LC has more flexibility than conv therefore, we considered using the extra flexibility of these models could further reduce MAC. To this end, we considered a masked version of LC, which we call mLC similar to mconv discussed in the main paper. More specifically, we considered two variants of mLC, one is based on FRLC (mFRLC), and the other is based on LRLC (mLRLC). The number of learnable parameters is $\text{mFRLC} \gg \text{mLRLC} > \text{mconv}$; however, all the variants requires the same number of neurons and synapses when mapped onto graph-based processors (given the same shape of weight and weight sparsity). The MAC and number of trainable parameters of mconv and mLC are summarized in Tab.A2.

D.1 Preliminary Results using mLC

We trained mFRLC and mLRLC networks using the same framework discussed in Sec. 3.2. In the preliminary experiments, mFRLC performed exceptionally well on N-MNIST and PilotNet experiments, achieving more than 3x better MAC/accuracy trade-off than the mconv(1x) model shown in Tab.2, both mFRLC and mLRLC network are trained in the same way using TDSS loss and quantizer with *macro-grad*. The mLRLC model also performed better than the mconv(1x) model, which showed a 1.5x better MAC/accuracy trade-off.

However, in our preliminary experiments on N-Caltech101 and Gen1 automotive, we saw the opposite result; mFRLC performed poorly than the model using mconv the mLRLC only slightly outperformed the model using mconv. We suspect this is due to overfilling due to the excessive flexibility of mLC since the VGG13-mFRLC model has a too large degree of freedom for the amount of given training data, making it over-fit to the training data.

Thought, we believe mLC trained with the proposed *DSS aware training* framework will further reduce MAC on these datasets if we incorporate an appropriate regularizer or utilize training techniques such as self-supervised training. The computation of DSS of (12) does not require labels. Therefore, we can train the network in a self-supervised way utilizing pseudo-labels estimated by another pre-trained network for unlabeled data. We will leave the investigation as the future research topic. For the benefit of interested readers, each network is briefly described below.

D.2 Masked Full-Rank Locally Connected Layer (mFRLC)

This variant has maximum flexibility, and it is expected to achieve a better MAC/accuracy trade-off given sufficient training data. The parameters for mFRLC are more than two orders of magnitude larger than those of mconv. One could realize the mFRLC model simply by replacing mconv layer with FRLC layer⁶. In the case of mFRLC, the TDSS loss in (12) can

⁶This is implemented as default layer in some frameworks, e.g., in Keras, it is implemented as LocallyConnected2D layer. Since this operation does not exist in Pytorch [49] (version 1.8), we implemented it

be evaluated in the same way as described for Sec. 3.2.2. The possible major drawback of this network is over-fitting.

D.3 Masked Low-Rank Locally Connected Layer (mLRLC)

To mitigate the overfitting of mFRLC, we considered mLRLC, which is the low-rank version of mFRLC. This variant is based on LRLC [10] and extended by introducing a masking mechanism. Once the position dependant low-rank LC weight is generated by a linear combination of the filter bank, it is also used as a mask in the same way we discussed in Sec. 3.2.2 using the masking procedure of (14).

We have reimplemented LRLC in Pytorch, referring to their paper [10] and open-sourced code⁷. For combining weights, we utilized the technique described in their paper, which learn combining weights per-row and per-column of location (r, c) to reduces the number of combining weights parameters as follows:

$$w_{rc}^{(l)} = \alpha_r^{(l)} + \beta_c^{(l)}. \quad (18)$$

Spatially varying bias is also parameterized in a similar manner, which is also described in their paper as follows:

$$B_{r,c,f} = b_r^{\text{row}} + b_c^{\text{column}} + b_f^{\text{channel}}, \quad (19)$$

where $b^{\text{row}} \in \mathbb{R}^H$, $b^{\text{column}} \in \mathbb{R}^W$, and $b^{\text{channel}} \in \mathbb{R}^{C_{\text{out}}}$.

Improving training efficiency on GPUs To increase the computational efficiency on the GPU, they first perform the convolution with the filter banks and then linearly combine the results instead of synthesizing equivalent LC weight and then perform the convolution with the synthesized weight. Although this technique increases the required MAC, the actual computational speed is greatly improved on GPU. This seemingly contradictory result is due to the GPU’s limited memory bandwidth.

We cannot use the technique because we need synthesized weight to compute mask m using the binarizer of (14) to evaluate the TDSS of (12). Therefore, we first synthesize the weight and perform the convolution with the synthesized weight. On GPUs, this method is several times slower than using their technique.

Note that the opposite is true when running the LC (FRLC or LRLC) network on a graph-based processor, where directly applying the synthesized LC filter requires less MAC, and it is actually faster.

Table A2: **Ablation analysis of computational complexity.** $\rho_i^{(l)}$ and $\rho_m^{(l)}$ represent ratios of non-zero elements in $\Delta I^{(l)}$ and $m^{(l)}$, respectively. $\mathcal{M}^{(l)} := C_{\text{out}}^{(l+1)} k^{(l+1)} k^{(l+1)} C_{\text{in}}^{(l+1)}$. The table shows the case where stride=1. The superscript in l represents layer index.

	MAC		# param.	# synapse
	Dense	$\Sigma \Delta$		
mconv	$H^{(l)} W^{(l)} \mathcal{M}^{(l)}$	$H^{(l)} W^{(l)} \rho_m^{(l)} \rho_i^{(l)} \mathcal{M}^{(l)}$	$\mathcal{M}^{(l)}$	$H^{(l)} W^{(l)} \mathcal{M}^{(l)}$
mLC	$H^{(l)} W^{(l)} \mathcal{M}^{(l)}$	$H^{(l)} W^{(l)} \rho_m^{(l)} \rho_i^{(l)} \mathcal{M}^{(l)}$	$H^{(l)} W^{(l)} \mathcal{M}^{(l)}$	$H^{(l)} W^{(l)} \mathcal{M}^{(l)}$

using unfold and einsum operation.

⁷https://github.com/google-research/google-research/tree/master/low_rank_local_connectivity

E Neuron model

SNNs [53] have recently gained much attention, especially for the low-power processing of sparse data from event-based cameras on edge devices. The key to their efficiency lies in the sparse event-based processing, and they show promising energy efficiency [54, 56, 55]. The leaky and fire (LIF) neuron (Fig. A3-a) is one of the popular neuron models in SNN. In the LIF model, neurons communicate using binary spike $\delta(t)$; This binary spike generates a time-decaying current $I(t)$. The current is integrated according to the weights w to update the time-decaying membrane potential. When this membrane potential reaches a threshold, it fires a binary spike. However, the LIF model involves complex temporal dynamics that prevent scaling to larger neuron sizes. This also poses difficulties concerning training due to the non-differentiability.

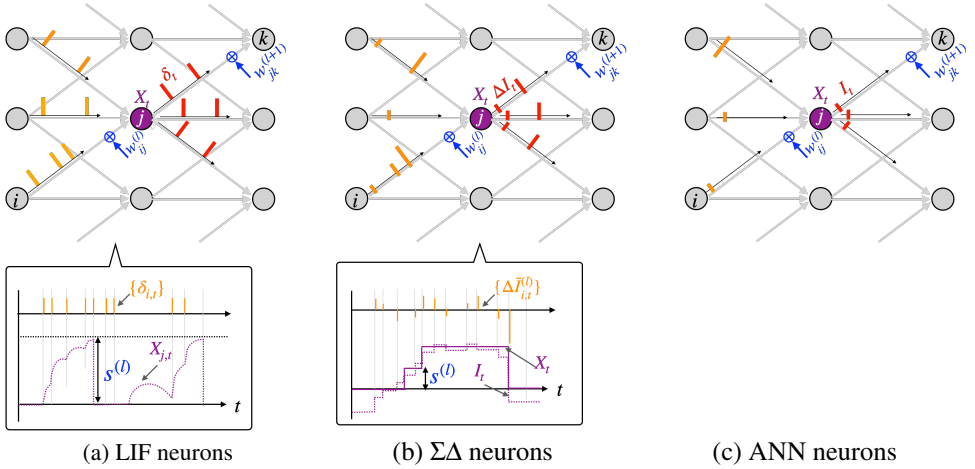


Figure A3: $\Sigma\Delta$ type neuron compared with SNN neuron and ANN neuron. The $\Sigma\Delta$ neurons discussed in the paper are shown in (b). It is equivalent to Temporal-Difference (TD) neuron when the quantization Q in (3) is identity. In the case of $\Sigma\Delta$ neuron, the last fired activation I_t (solid purple line) behaves with a slight delay relative to the membrane potential X_t (dashed purple line). The $\Sigma\Delta$ or TD neurons are somewhat intermediate between LIF neuron (a) and ANN neuron (c).

The temporal-difference (TD) neuron [46, 48] (Fig. A3-b) is equivalent to ANN neuron (Fig. A3-c). Therefore it can be easily trained using error back-propagation on the equivalent ANN network. Furthermore, TD neurons do not have complex temporal dynamics as LIF; therefore, they are highly compatible with digital circuits.

To further reduce MAC for computing $w^{(l)} * \Delta I_t^{(l)}$ one needs to increase the sparsity of $\Delta I_t^{(l)}$. The RRM [48] utilizes thresholding, while the $\Sigma\Delta$ [46] network utilizes quantization for the difference of activation. Since the RRM thresholding simply discards values whose absolute value is less than the threshold, errors accumulate over time; thus, a resetting mechanism is required. Our framework is applicable for both TD and $\Sigma\Delta$ neurons. We choose $\Sigma\Delta$ neuron as our base model because it does not require resetting (easier to operate). It is also hardware feasible, and SoC designed for this type of neuron is emerging on the market [43]. As discussed in Sec. 3.1, $\Sigma\Delta$ network is equivalent to quantized ANNs. Therefore, in

some respects, the $\Sigma\Delta$ neuron can be considered somewhere between the LIF and (quantized) ANN neuron models. The $\Sigma\Delta$ network could be operated very efficiently when the network is mapped onto processors that can exploit dynamic sparsity (ignore zeros in activation map or weight), such as EIE [23], IPU [24], TrueNorth [5], or neuron-flow [43]. These devices use an in-memory processor architecture where memory and ALU (arithmetic logic unit) are co-located, which is quite different from Neumann-type architectures such as GPUs.

F Soft-shrink and Binarizer

The soft-shrink function \mathcal{S} of (13) and the binarizer function \mathcal{B} of (14) discussed in Sec. 3.2 is defined as follows:

$$\mathcal{S}(x, \gamma) = \begin{cases} x - \gamma, & \text{if } x > +\gamma \\ x + \gamma, & \text{if } x < -\gamma \\ 0, & \text{otherwise} \end{cases} \quad (20)$$

$$\mathcal{B}(x, \gamma) = \begin{cases} 1, & \text{if } |x| > \gamma \\ 0, & \text{otherwise} \end{cases} \quad (21)$$

We use the following gradient for the operations,

$$\frac{\partial \mathcal{S}}{\partial x}(x, \gamma) = \begin{cases} 1, & \text{if } |x| > \gamma \\ x, & \text{otherwise} \end{cases} \quad (22)$$

$$\frac{\partial \mathcal{B}}{\partial x}(x, \gamma) = \begin{cases} \text{sgn}(x) & \text{if } |x| > \gamma \\ 0, & \text{otherwise} \end{cases} \quad (23)$$

The gradient for \mathcal{S} has a non-zero value even when its output is zero; this enable inactive neuron (masked neuron) could be alive again to improve accuracy. The gradient for \mathcal{B} guides the mask to be sparser (disconnect synaptic connection) by decreasing the TDSS loss.

The threshold γ is initialized as follows:

$$\gamma = 0.1 \sqrt{\frac{6}{C_{out}}}. \quad (24)$$

We used initialization based on the Kaiming uniform method [24]. To compensated the effect of the softshrink we modified it as follows:

$$w \sim \mathcal{U}(-\beta, +\beta) \quad (25)$$

$$\beta = \sqrt{\frac{6}{C_{out}}} + \gamma \quad (26)$$

G MAC computation

In the main experiment, we used multiply-add accumulation (MAC) as the evaluation metric of computational complexity. We report MAC of conv or mconv for all networks (FC was treated as 1x1 conv or mconv). Following operations are omitted from the MAC calculation: i) bias of *Dense* network, ii) pooling (maxpool/avgpool) of all network type.

We report MAC instead of floating-point operations per second (FLOPS). This evaluation metric is adopted from [2, 62, 69], which uses the same neuron model. This is why the value listed in Tab.2 is about half of the value reported in [42]. FLOPS count the product and sum operations separately, while MAC counts the product and sum operations as a single operation. Since the results in Tab.2 ignore the MAC for pooling operations; therefore, the MAC of *Asyc-SSC* in Tab.2 is not exactly half, but slightly less than half of the FLOPs reported in [42].

H Pytorch code for *macro-grad*

The Pytorch [49] implementation of the proposed *macro-grad* is shown in listing 1. It is implemented using *auto-grad*; therefore, the framework automatically computed backward computation of (9).

Listing 1: Pytorch implementation of the quantization using *macro-grad* (*mg_round*) and *LSQ* (*lsq_round*).

```

1 def ste_round(x):
2     return x + (x.round() - x).detach()
3
4 def mg_round(x, s):
5     return ste_round(x.div(s)).mul(s.data)
6
7 def lsq_round(x, s):
8     return ste_round(x.div(s)).mul(s)

```

I Pytorch code for Conv2d with TDSS loss

The Pytorch [49] implementation of the Conv2d for training with TDSS loss is show in listing 2.

Listing 2: Pytorch implementation of Conv2d with TDSS. Input x for each convolution layer is assumed to be a concatenation of temporally consecutive frame in batch dimension. The TDSS loss is computed by `compute_tdss` (We call this using `forward_hook`).

```

1  class TDSS_Conv2d(nn.Module):
2      def __init__(self, in_channels, out_channels, kernel_size, stride=1, padding=(0,0)):
3          super(TDSSConv2d, self).__init__()
4          self.kernel_size = kernel_size
5          self.padding = padding
6          self.in_channels = in_channels
7          self.out_channels = out_channels
8          self.stride = stride
9          self.quantizer = mg_round
10
11         self.register_parameter("weight", nn.Parameter(torch.zeros(out_channels, in_channels, *kernel_size)))
12         self.register_parameter("bias", nn.Parameter(torch.zeros(out_channels)))
13
14         self.masker = Masker.apply(x)
15         self.binarize = Binarizer.apply(x)
16
17     def compute_tdss(self, x):
18         x = self.quantizer(x)
19         x0, x1 = torch.chunk(x, 2, dim=0)
20         d = F.unfold(x0.sub(x1).abs(), kernel_size=self.kernel_size, padding=self.padding, stride=self.stride)
21
22         mask = self.binarize(self.weight).flatten(1)
23
24         mac = torch.einsum('oi_bis->b', mask, (d!=0).to(x))
25         tdss = torch.einsum('oi_bis->b', mask, d)
26         return tdss, mac
27
28     # x = torch.cat([x_t0, x_t1], dim=0)
29     def forward(self, x):
30         x = self.quantizer(x)
31         x = F.conv2d(x, self.masker(self.weight), bias=self.bias, stride=self.stride, padding=self.padding)
32         return x
33
34     DEFAULT_LAMBD = 5e-3
35     class Binarizer(torch.autograd.Function):
36         @staticmethod
37         def forward(ctx, inputs, lambd=DEFAULT_LAMBD):
38             ctx.save_for_backward(inputs)
39             return inputs.abs().gt(lambd).to(inputs)
40         @staticmethod
41         def backward(ctx, grad_output):
42             inputs = ctx.saved_tensors
43             grad_output = (inputs[0].sgn())*grad_output
44             return grad_output, None
45
46     class Masker(torch.autograd.Function):
47         @staticmethod
48         def forward(ctx, inputs, lambd=DEFAULT_LAMBD):
49             ctx.save_for_backward(inputs)
50             ctx.lambd = lambd
51             return F.softshrink(inputs, lambd=lambd)
52         @staticmethod
53         def backward(ctx, grad_output):
54             inputs = ctx.saved_tensors[0]
55             lambd = ctx.lambd
56             idx_masked = inputs.abs().le(lambd)
57             grad_output[idx_masked]*=(1/lambd*inputs[idx_masked].abs())
58             return grad_output, None

```