Spike-Based Anytime Perception
Supplementary Material

Matthew Dutson, Yin Li, Mohit Gupta
University of Wisconsin–Madison
`{dutson,yin.li,mgupta37}@wisc.org`

This is the supplement to our main paper. Here we present some additional results (section A). We also provide further details on our methods (section B and section C) and our experiments (section D). For sections, figures, tables, and equations, we use numbers (*e.g.*, Figure 1) to refer to the main paper and capital letters (*e.g.*, Figure A) to refer to this supplement.

A. Additional Results

Simulator Runtimes. Table A compares the runtime performance of SaRNN to two other SNN simulators: SNN-TB [2] and NEURON [1]. Like SaRNN, SNN-TB has a TensorFlow backend and is optimized for the NL-IAF neuron model; however, it repeatedly invokes the Python interpreter during simulation. In contrast, SaRNN compiles the simulation to a static computation graph. NEURON is a general-purpose simulator that supports a broader range of neuron models, but it only runs on the CPU. For this reason, NEURON is much slower than SaRNN or SNN-TB.

We test the three MNIST and CIFAR architectures defined in section D. We use an Intel Core i7-8700K CPU and GeForce GTX 1080 GPU. The PyNN API used to interface with NEURON does not support weight-sharing convolutions, which makes the NEURON convolutional models hugely inefficient. For this reason, we only report NEURON results on the dense MNIST model (we expect the actual runtimes on convolutional models to be at least 1×10^4 s).

Generalization To Threshold Metrics. The main paper shows results on the proposed Pareto metrics, P_l and P_p. Table E shows that our improvements also generalize to threshold metrics (time or synaptic events to an accuracy threshold). We decrease the accuracy thresholds for the "after" models to account for the slight reduction in ANN accuracy caused by sparsity fine-tuning (Table F). Figure A show model accuracy as a function of time, with dotted lines showing the point at which a model crosses its accuracy threshold. Our optimizations reduce both Pareto latency and the amount of time required to cross the threshold.

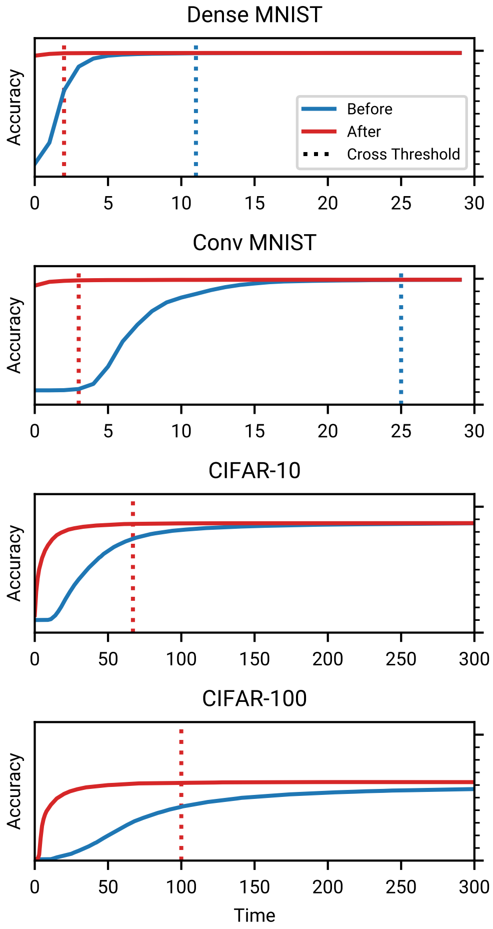

Figure A: **Accuracy Time Series.** Accuracy of the MNIST and CIFAR models as a function of time, before and after our optimizations. Dotted lines show the point at which a model crosses the accuracy threshold in Table E.

Sparsity Improvements. Table F and Table G show the change in ANN sparsity and accuracy after sparsity fine-tuning. The activation sparsity S_a is the measured fraction of zero activations on the test dataset. The weight sparsity

Table A: **Simulator Runtimes.** We simulate 50 time steps over 1000 dataset items.

Simulator	Dense MNIST	Conv MNIST	Conv CIFAR-10
NEURON	261 s	–	–
SNN-TB	23.7 s	266 s	1047 s
SaRNN (ours)	2.93 s	28.5 s	139 s

S_w is the measured fraction of zero synapses. The combined sparsity S_c is defined as $1 - (1 - S_a)(1 - S_w)$. We see that, in general, sparsity fine-tuning sacrifices a small amount of accuracy for a significant increase in sparsity.

MNIST and CIFAR. Table H shows the data corresponding to Figure 5 in the main paper. Figure B shows example inferences on CIFAR-10.

B. Input and Output

This section expands on section 3 (Background) from the main paper.

Input Layer. As part of conversion, we add a spiking input layer to the SNN. During inference, each neuron in this layer receives a real-valued input equal to the intensity of one image pixel (in the range 0 to 1). This value is added to the neuron's membrane potential on each time step. Input layer neurons spike according to the standard NL-IAF rules. As a result, these neurons have an average firing rate equal to the their pixel intensity.

Output Layer. Most classification ANNs have a final softmax layer which is paired with a cross-entropy loss during training. Because our converted SNNs are used only for inference, we can discard the softmax and determine the class prediction simply by finding the maximum value of the raw output logits. Logits, which may take negative values, would be truncated to zero by NL-IAF. Therefore, we disable spiking in the final layer and read the membrane potentials directly.

C. Operation Counting

This section expands on section 6 (SNN Simulation) from the main paper.

Computing the P_p term in L_{SNN} requires counting the number of synaptic operations. Counting synaptic operations is not as straightforward as it may seem. For example, counting operations in a convolutional layer requires accounting for strides, dilation, and padding. A naive implementation of operation-counting logic can easily be more expensive than the inference itself.

We use the following method. Let $\boldsymbol{S}_{i,t}$ be a tensor of incoming spikes to layer i at time t, let $\boldsymbol{W}_i$ be the weight kernel for layer i, and let $f_i(\boldsymbol{S}_{i,t}; \boldsymbol{W}_i)$ be a function which computes the linear component of the layer (*i.e.*, multiplication by synaptic weights before updating the membrane potential). The number of synaptic operations is

$$\sum f_i(\boldsymbol{S}_{i,t}; \mathbf{1}), \tag{A}$$

where $\mathbf{1}$ is a tensor of ones with the same shape as $\boldsymbol{W}_i$. Equation A can be computed within the RNN's update loop, preventing inefficient calls to the Python interpreter. To account for weight sparsity, we can replace $\mathbf{1}$ with a tensor containing a zero wherever $\boldsymbol{W}_i$ is zero and a one wherever $\boldsymbol{W}_i$ is nonzero.

D. Experiment Details

MNIST and CIFAR. Table B, Table C, and Table D show the architectures of our MNIST and CIFAR models. Each weighted layer except the last is followed by batch normalization and ReLU. The final layer uses a softmax activation and has no batch normalization. All weighted layers have biases, and we fix the γ of all batch normalization to 1.

We train these ANNs using stochastic gradient descent (SGD) for 50 epochs with a learning rate (LR) of 5×10^{-2}, followed by 50 epochs with a LR of 5×10^{-3}. We apply an L2 penalty of 10^{-3} to kernel weights for regularization. We hold aside 10 000 items from each dataset for validation and save the model at the epoch with the lowest validation loss. All models take less than an hour to train on a single GTX 1080 GPU.

For sparsity fine-tuning, we use activation and weight sparsity penalties $\lambda_a = 0.5$ and $\lambda_w = 20$ (based on a grid search over $\lambda_a \in \{0.2, 0.5, 1, 2\}$ and $\lambda_s \in \{5, 10, 20, 50\}$). We fine-tune using SGD for 100 epochs with a LR of 5×10^{-3}, followed by 100 epochs with a LR of 1×10^{-3}. As before, we save at the epoch with the lowest validation loss. All models take less than two hours to fine-tune on a single GTX 1080 GPU.

For optimization over L_{SNN} we set $\lambda_l = 1/P_l(H_0)$ and $\lambda_p = 1/P_p(H_0)$, where we define H_0 as the scaling set generated by a heuristic method (*e.g.*, percentile-based normalization). For MNIST models we set λ_e to 200, and for CIFAR models we set λ_e to 50. Our goal is for an $x\%$ increase in latency or power to balance against a $y\%$ reduction in accuracy. For example, for the dense MNIST model a $0.5\,\%$ reduction in accuracy would be balanced by a $50\,\%$ reduction in P_l and P_p.

Table B: **Dense MNIST Architecture**

Layer	Type	Output Shape
1	Fully-connected	128
2	Fully-connected	128
3	Fully-connected	10

Table C: **Convolutional MNIST Architecture**

Layer	Type	Output Shape
1	Convolution 3×3	$28 \times 28 \times 32$
2	Convolution 3×3	$28 \times 28 \times 32$
3	Average pooling 2×2	$14 \times 14 \times 32$
4	Convolution 3×3	$14 \times 14 \times 64$
5	Convolution 3×3	$14 \times 14 \times 64$
6	Average pooling 2×2	$7 \times 7 \times 64$
7	Fully-connected	128
8	Fully-connected	10

Table D: **Convolutional CIFAR Architecture**

Layer	Type	Output Shape
1	Convolution 3×3	$32 \times 32 \times 64$
2	Convolution 3×3	$32 \times 32 \times 64$
3	Convolution 3×3	$32 \times 32 \times 64$
4	Average pooling 2×2	$16 \times 16 \times 64$
5	Convolution 3×3	$16 \times 16 \times 128$
6	Convolution 3×3	$16 \times 16 \times 128$
7	Convolution 3×3	$16 \times 16 \times 128$
8	Average pooling 2×2	$8 \times 8 \times 128$
9	Convolution 3×3	$8 \times 8 \times 256$
10	Convolution 3×3	$8 \times 8 \times 256$
11	Convolution 3×3	$8 \times 8 \times 256$
12	Average pooling 2×2	$4 \times 4 \times 256$
13	Fully-connected	512
14	Fully-connected	10/100

We optimize the MNIST models on a cluster node with two GTX 1080 GPUs and the CIFAR models on a node with four GTX 1080 GPUs. The MNIST optimizations take approximately one week each, and the CIFAR optimizations take approximately three weeks each.

ImageNet. We train the MobileNet ANN using SGD for 25 epochs with a LR of 1×10^{-2}, followed by 25 epochs with a LR of 1×10^{-3}. Otherwise, the training hyperparameters are identical to those used in the basic experiments.

During sparsity fine-tuning, we set λ_a to 2 and λ_w to 100 (based on a search over the values $(\lambda_a, \lambda_s) \in \{(0.1, 5), (0.2, 10), (0.5, 20), (1, 50), (2, 100), (5, 200)\}$). We fine-tune for 25 epochs with a LR of 1×10^{-3}, followed by 25 epochs with a LR of 1×10^{-4}. For optimization over L_{SNN}, we set λ_e to 50. Otherwise, both sparsity fine-tuning and optimization over L_{SNN} are identical to the basic experiments.

SpiNNaker. Our SpiNNaker model uses the dense MNIST architecture (Table B), with one modification. At the first layer, the unmodified dense model has a fan-in of $784 : 1$. This fan-in appears to overwhelm the SpiNNaker communication fabric, so we add a 2×2 average pooling before the first layer. Training, sparsity fine-tuning, and optimization over L_{SNN} are otherwise identical to the dense MNIST model in the basic experiments.

Because of limits on our access to SpiNNaker, the before/after evaluations (Table 2) only cover 500 of the $10\,000$ MNIST test-set items. However, we are optimistic that the improvements we see would generalize to the rest of the dataset.

References

[1] Nicholas T. Carnevale and Michael L. Hines. *The NEURON Book*. Cambridge University Press, Jan. 2006.

[2] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in Neuroscience*, 11, 2017.

Table E: **Generalization to Threshold Metrics.** Red values indicate that the threshold was not crossed during the simulation, so the value shown is an underestimate.

	Threshold		Time		Synaptic Events	
Model	Before	After	Before	After	Before	After
Dense MNIST	98 %	97.97 %	12	3	1.75×10^7	4.19×10^6
Conv MNIST	99 %	98.70 %	26	4	1.77×10^8	5.15×10^6
CIFAR-10	88 %	86.29 %	1001	68	6.80×10^9	1.12×10^8
CIFAR-100	62 %	61.71 %	1001	101	5.89×10^9	1.84×10^8

Table F: **Sparsity-Induced Accuracy Reduction.** Model accuracy before and after sparsity fine-tuning.

Model	Before	After
Dense MNIST	98.36 %	98.29 %
Conv MNIST	99.55 %	99.25 %
CIFAR-10	89.39 %	87.68 %
CIFAR-100	63.68 %/85.15 %	63.39 %/85.32 %

Table G: **Sparsity Improvements.** Model sparsity before and after sparsity fine-tuning.

	Activation (S_a)		Weight (S_w)		Combined (S_c)	
Model	Before	After	Before	After	Before	After
Dense MNIST	0.618	0.812	0.152	0.431	0.676	0.893
Conv MNIST	0.535	0.959	0.013	0.839	0.541	0.993
CIFAR-10	0.545	0.781	0.012	0.615	0.550	0.916
CIFAR-100	0.537	0.723	0.012	0.498	0.543	0.861

Table H: **MNIST and CIFAR.** The effect of our optimizations on accuracy, latency, and power. This data corresponds to Figure 5 in the main paper.

	Peak Accuracy		Latency (P_l)		Power (P_p)	
Model	Before	After	Before	After	Before	After
Dense MNIST	98.32 %	98.29 %	2.19	0.005 27	3.14×10^6	7.34×10^4
Conv MNIST	99.43 %	99.22 %	7.02	0.120	4.31×10^7	1.51×10^5
CIFAR-10	87.47 %	87.04 %	65.0	14.6	4.11×10^8	1.34×10^7
CIFAR-100	59.88 %	62.45 %	154	28.5	8.68×10^8	2.72×10^7

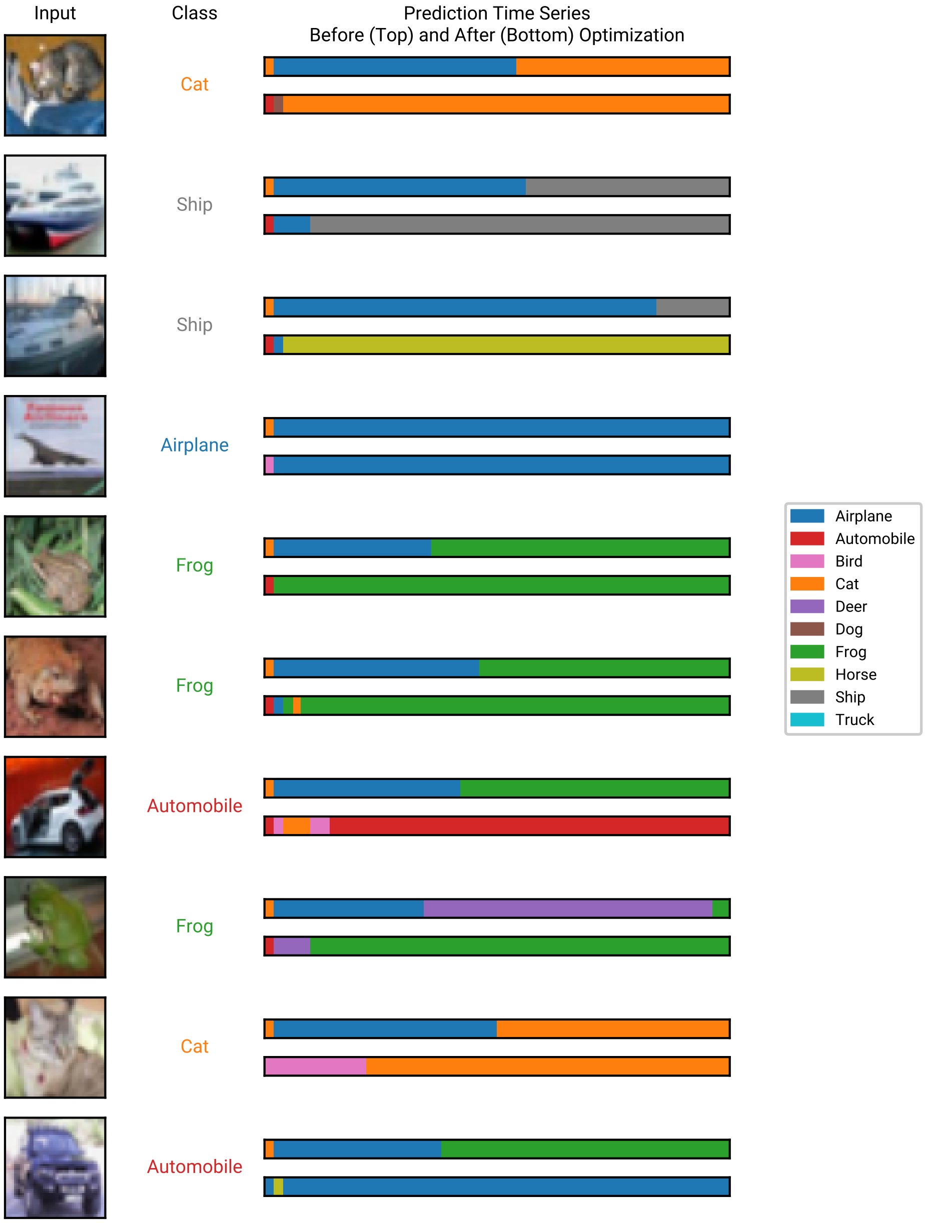

Figure B: **Prediction Evolution.** This figure shows example inferences on CIFAR-10. After our optimizations, the SNN converges much more quickly to the correct prediction.