Supplement: PixelRNN: In-pixel Recurrent Neural Networks for End-to-end-optimized Perception with Neural Sensors

Haley M. So[1], Laurie Bose[2], Piotr Dudek[2], Gordon Wetzstein[1]
[1]Stanford University, [2] The University of Manchester
https://www.computationalimaging.org/publications/pixelrnn/

1. Additional Details on Baseline Comparisons

1.1. Baseline Architectures

In this section, we discuss additional details of the baseline approaches and network architectures considered in the paper. For each baseline, we list the equations modeling the on-sensor computation. The output of this step is read off the sensor, transmitted to a host processor, and processed by a full-precision fully-connected layer there. This fully-connected layer is trained end to end for each baseline, but omitted in the following equations. The size of the weight matrix of this layer is $N \times M$, where N is the number of elements of the output of a baseline (i.e., it varies per baseline) and M is the number of categories to classify in the respective dataset ($M = 9$ for hand gesture recognition and $M = 4$ for lip reading). Note that this off-sensor model can be further optimized for a given task, but here we use the simplest off-sensor model to demonstrate the benefits of our spatio-temporal feature encoder. **Source code for all baselines will be made available.**

Table 1 shows an overview of all baselines, including their Top-1 accuracy for the two tasks, the number of parameters in the feature encoder, the memory required to store intermediate features in the forward pass, and the number of values outputted.

1.1.1 RAW Mode

The RAW camera mode simply outputs the input with a resolution of 256×256 pixel at 8 bit precision at every time step. This data is sent off the sensor and processed by a fully-connected layer there. The output layer on the sensor, $\mathbf{o}_t$, is simply the input $\mathbf{x}_t$ at each time step t:

$$\mathbf{o}_t = \mathbf{x}_t. \tag{1}$$

The accuracy of this model, as well as other baselines without an RNN, is calculated by averaging the scores over the 16 frames and use the largest as the prediction for the video.

1.1.2 Difference Camera

The difference camera is a simple temporal feature encoder.

$$\mathbf{h}_t = \mathbf{x}_t \tag{2}$$

$$\mathbf{o}_t = \psi_o\left(\mathbf{x}_t - \mathbf{h}_{t-1}\right) \tag{3}$$

where $\psi_o\left(x\right) = \begin{cases} -1 & \text{for x} < -\delta \\ 0 & \text{for } -\delta \leq \text{x} \leq \delta \\ 1 & \text{for } \delta < \text{x} \end{cases}$, for some threshold δ. The difference camera's output is sparse. For comparison, we output 64×64 ternary frame indicating the polarity of the pixel at every time step t. For this purpose, we add a hidden state $\mathbf{h}_{t-1}$ that "remembers" the last intensity value of each pixel. The output at each frame is then the difference between the current intensity value at a pixel and the hidden state, i.e., the intensity recorded at the last frame.

1.1.3 Naive Spatial Downsampling (NSD)

We test naive spatial downsampling of the RAW input by downsampling from 256×256 to 64×64 sized images with bilinear interpolation. Each frame is outputted to the decoder, resulting in $16\times$ more data readout than our encoder.

1.1.4 Naive Spatio-Temporal Downsampling (NSTD)

We further downsample the naive spatial downsampling case temporally to achieve our level of compression. Every 16 frames, a single spatially downsampled image is sent out to the decoder. Note that our proposed method is theoretically able to learn the naive downsampling case.

1.1.5 CNN-only Encoder

We test just the CNN-only encoder that has a single convolutional layer of 16 kernels operating on the 64×64 downsampled image. This is the same CNN used in tandem with the RNNs. In this case, the features are sent to the off-sensor decoder every timestep.

1.1.6 Recurrent Neural Network (RNN) Architectures

We next describe several RNN architectures that we have tested at full precision and also using binary weights. We list the equations that readers will be most familiar with in the following. Note that we do not use bias values for any of these RNNs. Moreover, in the binary setting, all nonlinear activations functions ψ are replaced by the sign function in the forward pass and the gradient of the tanh function for backpropagation during training (this is smoother and more robust than naively binarizing the weights at the end of training or using the straight-through-estimator) unless otherwise noted. In all cases CNN $(\cdot)$ refers to a CNN-based feature encoder operating directly on the intensity, $\mathbf{x}_t$, at time step t. Unless otherwise noted, the output is read from the sensor only every 16 time steps and then processed off-sensor by the aforementioned fully-connected layer. The number of values output from each RNN is fixed to 4096 values every 16 time steps, as this is the number our hardware platform can achieve in practice. In the main paper, we also show how decreasing bandwidth deteriorates accuracy.

Simple Convolutional RNN (SRNN) The SRNN baseline is the simplest RNN that uses a hidden state $\mathbf{h}_t$ at each time step as

$$\tilde{\mathbf{h}}_t = \mathbf{w}_h * \text{CNN}\left(\mathbf{x}_t\right) + \mathbf{u}_h * \mathbf{h}_{t-1} + \mathbf{b}_h, \tag{4}$$
$$\mathbf{h}_t = \psi_h(\tilde{\mathbf{h}}_t), \tag{5}$$
$$\mathbf{o}_t = \tilde{\mathbf{h}}_t \tag{6}$$

where $\psi_h = \tanh()$ function. As mentioned before, we set bias values $\mathbf{b}_h$ for this architecture and all following to 0.

Convolutional Long Short-term Memory (LSTM) Long short-term memory (LSTM) models [4, 7] are among the most well-known RNN architecture. Originally introduced to mitigate the vanishing gradient problem during RNN training, an LSTM usually provides a hidden state $\mathbf{h}_t$, cell state $\mathbf{c}_t$, an input $\mathbf{i}_t$, output $\mathbf{o}_t$, and forget $\mathbf{f}_t$ gates. We use a convolutional variant of an LSTM:

$$\mathbf{f}_t = \psi_f\left(\mathbf{w}_f * \text{CNN}\left(\mathbf{x}_t\right) + \mathbf{u}_f * \mathbf{h}_{t-1} + \mathbf{b}_f\right), \tag{7}$$
$$\mathbf{i}_t = \psi_i\left(\mathbf{w}_i * \text{CNN}\left(\mathbf{x}_t\right) + \mathbf{u}_i * \mathbf{h}_{t-1} + \mathbf{b}_i\right), \tag{8}$$
$$\mathbf{o}_t = \psi_o\left(\mathbf{w}_o * \text{CNN}\left(\mathbf{x}_t\right) + \mathbf{u}_o * \mathbf{h}_{t-1} + \mathbf{b}_o\right), \tag{9}$$
$$\tilde{\mathbf{c}}_t = \psi_{\tilde{c}}\left(\mathbf{w}_{\tilde{c}} * \text{CNN}\left(\mathbf{x}_t\right) + \mathbf{u}_{\tilde{c}} * \mathbf{h}_{t-1} + \mathbf{b}_{\tilde{c}}\right), \tag{10}$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t, \tag{11}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \sigma(\mathbf{c}_t) \tag{12}$$

where $\psi_f = \psi_i = \psi_o = \psi_{\tilde{c}} = \sigma()$, the sigmoid activation function. Note that the LSTM in its standard notation has an output gate $\mathbf{o}_t$, but (confusingly), this is not what is actually read off the sensor. For the LSTM, the hidden state $\mathbf{h}_t$ is actually read off the sensor and further processed by the full-connected layer.

Convolutional Gated Recurrent Unit (GRU) The GRU [3] is among the most popular RNN architectures. It provides two gates: an update gate $\mathbf{z}_t$ and a reset gate $\mathbf{r}_t$. GRUs have been introduced as leaner variants of LSTM that offer similar performance in many application with fewer parameters. The convolutional GRU we use is

$$\mathbf{z}_t = \psi_z\left(\mathbf{w}_z * \text{CNN}\left(\mathbf{x}_t\right) + \mathbf{u}_z * \mathbf{h}_{t-1} + \mathbf{b}_z\right), \tag{13}$$
$$\mathbf{r}_t = \psi_r\left(\mathbf{w}_r * \text{CNN}\left(\mathbf{x}_t\right) + \mathbf{u}_r * \mathbf{h}_{t-1} + \mathbf{b}_r\right), \tag{14}$$
$$\tilde{\mathbf{h}}_t = \psi_{\tilde{h}}\left(\mathbf{w}_h * \text{CNN}\left(\mathbf{x}_t\right) + \mathbf{u}_h * (\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h\right), \tag{15}$$
$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t, \mathbf{o}_t = \mathbf{h}_t, \tag{16}$$

where $\psi_z = \psi_r = \sigma$ and $\psi_{\tilde{h}} = \tanh()$.

Similar to the LSTM, for a GRU the hidden state $\mathbf{h}_t$ is read off the sensor and further processed by the full-connected layer.

Convolutional Minimal Gated Unit (MGU) MGU [9] were introduced as the minimal variant of gated units, i.e., they only use a single gate which is treated as the forget gate $\mathbf{f}_t$. The convolutional variant we use is

$$\mathbf{f}_t = \psi_f\left(\mathbf{w}_f * \text{CNN}\left(\mathbf{x}_t\right) + \mathbf{u}_f * \mathbf{h}_{t-1} + \mathbf{b}_f\right), \tag{17}$$
$$\tilde{\mathbf{h}}_t = \psi_{\tilde{h}}\left(\mathbf{w}_h * \text{CNN}\left(\mathbf{x}_t\right) + \mathbf{u}_h * (\mathbf{f}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h\right), \tag{18}$$
$$\mathbf{h}_t = (1 - \mathbf{f}_t) \odot \mathbf{h}_{t-1} + \mathbf{f}_t \odot \tilde{\mathbf{h}}_t, \mathbf{o}_t = \mathbf{h}_t, \tag{19}$$

where $\psi_f = \sigma()$ and $\psi_{\tilde{h}} = \tanh()$.

Again, the hidden state $\mathbf{h}_t$ is read off the sensor and further processed by the full-connected layer.

Developing the PixelRNN Architecture We start with a simple, intuitive formulation of an RNN, inspired by the GRU and MGU. We call this the general RNN (**genRNN**)

$$\mathbf{f}_t = \psi_f\left(\mathbf{w}_f * \mathbf{CNN}(\mathrm{x}_t) + \mathbf{u}_f * \mathbf{h}_{t-1} + \mathbf{b}_f\right) \tag{20}$$
$$\tilde{\mathbf{h}}_t = \psi_{\tilde{h}}\left(\mathbf{w}_h * \mathbf{CNN}(\mathrm{x}_t) + \mathbf{u}_h * \mathbf{h}_{t-1} + \mathbf{b}_{\tilde{h}}\right) \tag{21}$$
$$\mathbf{h}_t = (1 - \mathbf{f}_t) \odot \mathbf{h}_{t-1} + \mathbf{f}_t \odot \tilde{\mathbf{h}}_t \tag{22}$$
$$\mathbf{o}_t = \psi_o\left(\mathbf{w}_o * \mathbf{CNN}(\mathrm{x}_t) + \mathbf{u}_o * \mathbf{h}_{t-1} + \mathbf{b}_o\right) \tag{23}$$

where $\psi_f = \psi_{\tilde{h}} = \sigma()$ and ψ_o an optional non-linear activation. The hidden state is updated by interpolating between the previous hidden state $\mathbf{h}_{t-1}$ and the new candidate hidden state $\tilde{\mathbf{h}}$. In the highly quantized binary setting, we simplify these equations even more by setting

$$\tilde{\mathbf{h}}_t = -\mathbf{h}_{t-1} \tag{24}$$

which is computationally simple and proved to work well in practice when the hidden state is re-initialized to all ones every K frames for each new gesture and the weights are either -1 or 1. We can simplify the genRNN form and arrive at **PixelRNN**'s architecture:

$$\mathbf{f}_t = \psi_f \left(\mathbf{w}_f * \text{CNN}\left(\mathbf{x}_t\right) + \mathbf{u}_f * \mathbf{h}_{t-1}\right), \tag{25}$$
$$\mathbf{h}_t = \mathbf{f}_t \odot \mathbf{h}_{t-1}, \tag{26}$$
$$\mathbf{o}_t = \psi_o \left(\mathbf{w}_o * \text{CNN}\left(\mathbf{x}_t\right) + \mathbf{u}_o * \mathbf{h}_{t-1}\right). \tag{27}$$

where ψ_o is the identity operator. We found having a separate output gate increases performance as it allows the hidden state to focus on accumulating information temporally while the output can choose what features to transmit from the hidden state and the input, instead of having the hidden state do both.

1.1.7 Synchronous Event Camera Approximation

As a temporal-only encoder, we look to the event camera. The ideal event camera is asynchronous and would call for a completely different kind of neural network architecture. In practice, events are time-stamped, effectively creating a synchronous event camera. One of the advantages of event cameras is that the readouts are often sparse if most of the scene is not changing. Here, we approximate a synchronous event camera that sends out a 64×64 binary image encoding the locations of events each timestep. The genRNN architecture can approximate this when the CNN layer is omitted, i.e. CNN() is an identity function, $\mathbf{w}_f = \mathbf{w}_h = \mathbf{w}_o = 1$, $\mathbf{u}_f = \mathbf{u}_o = -1$, $\mathbf{u}_h = 0$, ψ_h is the identity function. This leads to

$$\mathbf{f}_t = \psi_f(\mathrm{x}_t - \mathbf{h}_{t-1}) \tag{28}$$
$$\mathbf{h}_t = \mathbf{f}_t \odot (\mathrm{x}_t - \mathbf{h}_{t-1}) + (1 - \mathbf{f}_t) \odot \mathbf{h}_{t-1} \tag{29}$$
$$\mathbf{o}_t = \psi_o(\mathrm{x}_t - \mathbf{h}_{t-1}) \tag{30}$$

where $\psi_f\left(x\right) = \begin{cases} 1 & \text{for } |\text{x}| > \delta \\ 0 & \text{for } |\text{x}| \leq \delta \end{cases}$

and $\psi_o\left(x\right) = \begin{cases} 1 & \text{for } x > \delta \\ 0 & \text{for } |\text{x}| \leq \delta \\ -1 & \text{for } x < \delta \end{cases}$

We include the baseline comparison here in the supplement as an additional temporal-only encoder. The readout bandwidth is calculated as 64×64 'events' (values are -1 or 1). Unlike the RNNs, the event camera approximation does not compress temporally, so we evaluate the accuracy by averaging the scores over the 16 frames and use the largest as the prediction for the video.

1.2. Additional Experiments

Binarization Regimes As mentioned above, the nonlinearites in the RNN architectures become sign functions when operating in the $-1, 1$ binary regime, unless otherwise noted. The hidden states are similarly binarized. Alternatively, we could operate in the 0 and 1 regime. These lead to slightly different interpretations. In the $0, 1$ case, information can be lost when it is set to 0. In the $-1, 1$ case however, no information is "lost." Rather the values in the hidden state will flip between -1 and 1. To determine which regime to use, we tested all the architectures with both binarizations. To train the models effectively, we use the following gradient estimators.

In the $-1, 1$ case, we utilize the gradient of $\tanh(mx)$ as a gradient estimator for the sign function

$$\text{Forward: } w = q(\tilde{w}) = \text{sign}(\tilde{w}) \tag{31}$$
$$\text{Backward: } \frac{\partial q}{\partial \tilde{w}} \approx m \cdot (1 - \tanh^2(m\tilde{w})) \tag{32}$$

where m can be used to tune how steep the transition between -1 and 1 is. We test both our PixelRNN architecture as well the general architecture outlined in equations 20-23. Tanh approximates the sign function well, especially if m is large. Tanh is also differentiable, so its gradient can be used as a good gradient estimator for the sign function. However, if m is too large, there will be more saturated neurons. Empirically, we found networks trained with $m = 2$ had the best validation performance with around 37.8% saturated neurons for the converged network.

In the $0, 1$ case, we utilize the analogous gradient of the sigmoid function $\frac{1}{1+e^{-mx}}$ to estimate the gradient

$$\text{Forward: } w = q(\tilde{w}) = \psi(\tilde{w}) \tag{33}$$
$$\text{Backward: } \frac{\partial q}{\partial \tilde{w}} \approx \frac{m \cdot e^{-mw}}{(1 + e^{-mw})^2} \tag{34}$$

where $\psi_f\left(x\right) = \begin{cases} 1 & \text{for } w > 0 \\ 0 & \text{for } w \leq 0 \end{cases}$
and m is used to tune the steepness of the transition between 0 and 1. We found binarizing to -1 and 1 had higher performances than when binarizing to 0 and 1.

In addition, we tested the stochastic binarization and the straight-through-estimator (STE) techniques for training binary networks. We opted for the gradient of the tanh function for it's slight edge over the STE and stochastic binarization. For the 1CNN+PixelRNN, tanh yielded 85.6% / 80.0% while STE yielded 80.0% / 80.0% for *hand gesture / lip reading* respectively.

Hidden State Initialization We tested different hidden state initializations, initializing to all zeros or all ones every K frames. In most cases, the initialization did not make a difference. In some cases, however, initializing to all ones performed slightly better on Hand Gesture recognition and Lip reading. For this reason, we initialize the hidden state to all ones every K frames.

Extended Architecture Results Along with the tested architectures shown in the main paper, we include the performances of a simple spatial downsampling to 64×64 for each

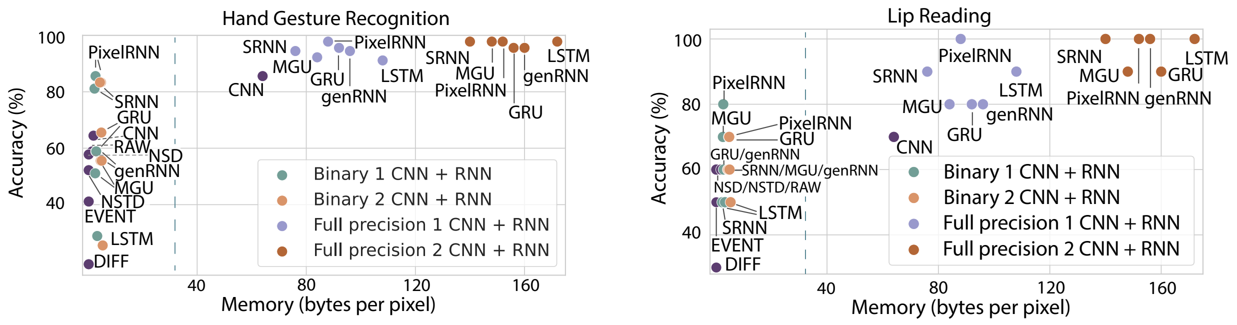

Figure 1. **Extended network architecture comparisons.** We plot the accuracy of several baseline architectures on two tasks: hand gesture recognition (left) and lip reading (right) on the test sets of the Cambridge Hand Gesture dataset and the Tulips1 dataset, respectively. As baselines, we use RAW, difference camera, approximate event camera, naive spatial downsampling (NSD), naive spatio-temporal downsampling (NSTD), binary and full precision CNN-only encoders (CNN), as well as the following RNN architecture: SRNN, MGU, GRU, LSTM, genRNN, and PixelRNN, each with 1- and 2-layer CNN encoders and binary or full 32 bit floating point precision. Note that the memory plotted on the x-axis represents all intermediate features, per pixel, that need to be stored when running an input image through the RNN. We do not count the network parameters in this plot, because they do not dominate the memory requirements and are shared among all pixels.

frame, naive spatio-temporal downsampling which further downsamples temporally by $16\times$, CNN-only encoder, our starting architecture genRNN, and the synchronous event camera approximation in the extended ablation on the same Cambridge Hand Gesture Recognition dataset and the Tulips1 lip reading dataset. Extended results can be seen in Table 1 and Figure 1. We evaluated all baselines described above on the hand gesture recognition task and lip reading task. The figure shows the the Top-1 accuracy in addition to the memory required by intermediate features of the respective network. This memory footprint is normalized per pixel and does not include the network parameters themselves, because these are shared across all pixels and, in the binary case, add a negligible amount of memory. The intermediate features are the bottleneck, especially for higher-resolution sensors using small RNN architectures with binary weights. Each RNN baseline is tested with the one- or two-layer CNN feature encoder as well as using full 32 bit floating point precision and 1 bit binary features.

For the full-precision networks, all RNNs perform well. Comparing the networks in the binary setting, PixelRNN offers the best accuracy. Larger architectures, including GRUs and LSTMs, do not perform well when used with binary weights. This can be explained by the increasing difficulty of reliably training larger RNN networks with binary parameter constraints as the exploding gradient problem worsens [5]. In highly quantized settings, popular RNNs like LSTMs may suffer due to excessive information loss caused by binarization through many gates. Leaner networks, such as SRNN and PixelRNN, better avoid this issue and can be trained more robustly and reliably in these settings.

Bandwidth Study. We include the statistics corresponding to the bandwidth studies in the main paper for hand gesture and lip reading in Table 2. Each experiment was run 10 times. We report mean and standard deviation. The larger standard deviation of lip reading is due to the small dataset.

Bit depth of readout for quantized implementations. The bit width can vary for the binary versions of the RNNs. Binary LSTM, GRU, and MGU architectures output the binary hidden state values as their outputs. However this limits the quantized performance significantly. The SRNN and PixelRNN architectures have an additional output convolution using the output of the CNN and the binary hidden state. This additional gate allows for higher precision output and leads to higher performance when working with the quantized non-linearities.

Practically, PixelRNN can be implemented with 2-bits read out per value. The output in this case would be the binary $\text{CNN}(\mathbf{x}_t)$ and binary $\mathbf{h}_{t-1}$ concatenated. As there is no non-linearity used in the calculation of the outputs $\mathbf{o}_t$ in the SRNN and PixelRNN, the convolution operation, (a linear operation), can simply be absorbed into the fully connected off-sensor decoder.

Larger Decoders. While we used a single linear layer as our decoder, another model can be swapped in for a given task. A larger decoder can boost performance, as shown in table 3, however, enlarging the decoder does not solve the original problem of decreasing the bandwidth readout between the sensor and processor. In this work, we focus on encoding as much information into a reduced bandwidth through our PixelRNN module, which can be used to complement any off-sensor model.

Temporal output rate: In Table 4, we show the performance as we vary the temporal output rate. We found that outputting once every K=16 frames offered balance between performance and compression.

Application to EgoGesture. EgoGesture is a benchmarking dynamic hand gesture recognition dataset, containing 83 classes [2, 8]. It has 24,161 RGB-D gestures collected in diverse indoor and outdoor settings. While the performance of the binary version of 1CNN+PixelRNN is limiting, our encoder at floating point precision can effectively encode the information into a reduced bandwidth for this challenging task. We convert the RGB data to grayscale as our input to our encoder and use the ResNeXt backbone model from [6] for the off-sensor decoder. Our encoder can compress the readout bandwidth by $8\times$ and still achieve 88.9% accuracy, while naive temporal downsampling resulted in 74.5%. Table 5 presents further comparisons at various compression levels and highlights how effective our encoder is at retaining important features. While this version of our encoder is at higher precision, this is in line with the expected increased memory future sensor–processors will host.

2. Additional Details

2.1. Additional Simulation Details

Training Details The Cambridge hand gesture dataset has 900 videos while the Tulips dataset has 96 videos. In both cases, we use a 80:10:10 train:validation:test split. As they are small datasets, we utilize data augmentation techniques. For each video, we select 16 evenly spaced frames starting with the first frame. Input images are resized to 256×256. During training, an optional, random temporal offset is also applied to the train set, if available. For videos with less than 16 frames, we randomly duplicate a subset of the frames until 16 frames are available. We also apply a random spatial offset to the training videos for further augmentation. During training, we use the Pytorch Adam optimizer and a cross entropy loss. All models were trained for 300 epochs, or until convergence, with learning rate sweeps from $1e-1$ to $1e-4$ and a learning rate scheduler that reduces when the loss and accuracy improvement have plateaued. Best scores for all models were reported.

2.2. Additional SCAMP-5 Prototype Details

Memory Allocation. SCAMP-5's analog and digital registers are limited in number and present different challenges. Analog registers cannot hold values for long periods before decaying. The decay is exacerbated if one moves information from pixel to pixel such as in shifting an image. We found using analog registers with a routine to refresh their content to a set of quantized values inspired by [1] helped circumvent some of the challenges. This allowed the storage of binary weights for convolutions and the hidden state for prolonged periods of time. The remaining memory registers were used for performing computations and storing intermediate feature maps.

SCAMP-5 Protocol. We retrained PixelRNN with varying levels of Gaussian noise added prior to quantization of the signal everywhere in the model. We then load the trained binary model weights into two of the register planes in SCAMP-5. Weight storage can be further optimized in the future. The 16 image sequence for each test video is extracted from the same pre-processing as in simulation and uploaded sequentially to SCAMP-5. The image is thresholded according to a learned parameter in simulation and subsequently binarized. To effectively use the whole sensor, we split the sensor into a 4×4 grid of 64×64 blocks of processing elements (PE) to perform 16 convolutions in parallel. The CNN operation is illustrated in figure 2. The output of the convolution at a single pixel is the 5×5 kernel, element-wise multiplied by the local 5×5 image region, then summed. Since the weights and input signal are both binary, the convolution operation can be calculated just by adding 1 or subtracting 1 from a running sum held in an analog register. Where the weight AND signal = 1, we add 1, otherwise, we subtract 1. In practice, to increase the signal-to-noise ratio and to circumvent some of the effects of analog noise, we add and subtract intervals of $\rho = 10$ to the analog register. In simulation, we used binary regime $\{-1, 1\}$. When stored in the digital bit registers on SCAMP-5, weights and signals that are -1 are stored as 0, so the 'multiplying' operation in the binary convolution becomes a XNOR operation, as opposed to the AND operation. The output of the convolution is then thresholded with a learned parameter during training and binarized. This process is used for the CNN as well as for the convolutions in the RNN.

The RNN operation is shown in figure 3. The RNN uses just 4 of the 16 available PEs as just 4 convolutions are needed for the gate computations. At the beginning of each video sequence, the hidden state is initialized to all ones, and then the hidden state is updated and binarized each time step. The outputs for each video, i.e. the encoded images are saved out every 16 frames. We perform this process on the whole dataset. We use the outputs of the train set to fine-tune the off-sensor model, which is in this case a fully connected layer. The SCAMP-5 outputs of the test set are then fed into the fully connected layer and the class is predicted. Adding Gaussian noise with mean 0 and standard deviation 0.6 helped approximate some of the noise in SCAMP-5.

Table 1. **Network architecture comparison.** We evaluate the accuracy of several baseline architectures on two tasks: hand gesture recognition and lip reading. We show RAW, the difference camera, the event camera, naive spatial downsampling, CNN-only encoder, and naive spatio-temporal downsampling, as well as the following RNN architecture: SRNN, MGU, GRU, LSTM, genRNN, and PixelRNN, each with 1- and 2-layer CNN encoders and binary or full 32 bit floating point precision. As PixelRNN has better accuracy than the genRNN and fewer operations and memory required, we chose PixelRNN to be our final architecture. In this table, we also list the number of parameters for each model, which includes the CNN feature encoder and the respective RNN. The feature memory column lists the memory footprint of all intermediate features that need to be computed during a forward pass through each network. This does not include the model weights. Finally, the final column lists the number of values that are read out in total within a 16 frame sequence.

Model Name	Hand Gesture Top-1 Accuracy % (1 bit / 32 bit)	Lip Reading Top-1 Accuracy % (1 bit / 32 bit)	# Model Params	Feature Memory in bytes per pixel (1 bit / 32 bit)	Values Read Out (per 16 timesteps)
RAW	— / 58.89%	— / 60.00%	0	— / 1.00	$256^2 \cdot 16$ = 1,048,576
Difference Camera	18.89% / —	30.00% / —	0	0.25 / —	$64^2 \cdot 16$ = 65,536
Event Camera*	41.11% / —	50.00% / —	0	0.25 / —	65,536
Naive Spatial Downsampling	— / 57.78%	— / 60.00%	0	— / 0.25	65,536
CNN-only Encoder	64.44% / 85.56%	60.00% / 70.00%	401	2.00 / 64.00	65,536
Naive Spatio-temporal Downsampling	— / 52.22%	— / 60.00%	0	— / 0.25	64^2 = 4,096
1 CNN + SRNN	81.11% / 94.44%	50.00% / 90.00%	451	2.38 / 76.00	4,096
1 CNN + LSTM	28.89% / 91.11%	50.00% / 90.00%	601	3.38 / 108.00	4,096
1 CNN + GRU	58.89% / 95.56%	60.00% / 80.00%	551	2.88 / 92.00	4,096
1 CNN + MGU	51.11% / 92.22%	70.00% / 80.00%	501	2.63 / 84.00	4,096
1 CNN + genRNN (0,1)	58.89% / 94.44%	60.00% / 80.00%	553	3.00 / 96.00	4,096
1 CNN + genRNN (-1,1)	84.44% / 94.44%	80.00% / 80.00%	553	3.00 / 96.00	4,096
1 CNN + PixelRNN	**85.56% / 97.78%**	**80.00% / 100.00%**	501	2.75 / 88.00	4,096
2 CNN + SRNN	83.33% / **97.78%**	60.00% / **100.00%**	852	4.38 / 140.00	4,096
2 CNN + LSTM	25.56% / **97.78%**	50.00% / **100.00%**	1,002	5.38 / 172.00	4,096
2 CNN +GRU	65.56% / 95.56%	70.00% / **100.00%**	952	4.88 / 156.00	4,096
2 CNN +MGU	55.56% / **97.78%**	60.00% / 90.00%	902	4.63 / 148.00	4,096
2 CNN + genRNN (0,1)	55.56% / 95.56%	60.00% / 90.00%	954	5.00 / 160.00	4,096
2 CNN + genRNN (-1,1)	83.33% / 95.56%	60.00% / 90.00%	954	5.00 / 160.00	4,096
2 CNN + PixelRNN	**87.78% / 97.78%**	**70.00% / 100.00%**	902	4.75 / 152.00	4,096

Table 2. **Bandwidth Study Statistics.** We report the numbers corresponding to the bandwidth figures in the main paper. We see we can potentially continue to decrease the bandwidth past 4,096 values. The number of values read out is modulated by a maxpooling operation prior to readout, resulting in the collection of readout values shown below.

Encoder	**Number of values read out**								
	4096	**1024**	**456**	**256**	**164**	**114**	**84**	**64**	**16**
Hand Gesture									
1CNN+PixelRNN	84.6 ± 3.7	81.6 ± 4.8	77.7 ± 4.3	72.3 ± 5.7	63.9 ± 5.9	61.3 ± 5.2	55.5 ± 4.6	43.1 ± 4.4	21.0 ± 3.9
2CNN+PixelRNN	85.2 ± 2.3	78.1 ± 2.5	65.0 ± 4.3	53.2 ± 3.5	49.5 ± 4.3	43.2 ± 5.5	37.7 ± 5.9	34.1 ± 4.9	22.1 ± 5.1
Lip Reading									
1CNN+PixelRNN	80.8 ± 6.7	72.5 ± 6.2	60.0 ± 7.4	43.3 ± 10.7	36.7 ± 6.5	38.3 ± 11.9	31.7 ± 11.9	25.0 ± 10.0	27.5 ± 8.7
2CNN+PixelRNN	73.0 ± 6.7	66.0 ± 10.7	57.0 ± 9.5	49.0 ± 9.9	47.0 ± 8.2	45.0 ± 9.7	45.0 ± 8.5	38.0 ± 10.3	20.0 ± 9.4

Table 3. **Larger Decoders.** Adding processing to the decoder can boost performance but does not reduce the readout bandwidth. In the 2 linear layer decoder, the hidden layer is 64 neurons whereas the 1 linear layer goes straight to the number of classes.

Encoder	**Decoder**	**Hand Gesture**	**Lip Reading**
RAW	2 linear	78.8%	60.0%
DIFF	2 linear	74.4%	60.0%
1CNN+PixelRNN	2 linear	**87.8%**	**85.6%**

Table 4. **Temporal output ablation.** Mean accuracy of outputting one every K frames using the binary 1CNN+PixelRNN model.

	K=4	K=8	K=12	K=16	K=20	K=24
Hand Gesture	86.0	84.8	84.1	84.2	82.5	80.8
Lip Reading	80.0	79.0	80.0	80.0	76.0	75.0

Table 5. **Application to EgoGesture.** Shown here is the readout compression level and the corresponding performance on the EgoGesture Hand Gesture Recognition Dataset that contains 83 gesture classes. The off-sensor model in all cases is the ResNeXt backbone model from [6].

	4×	8×	18×	36×
Naive Temporal Downsampling	82.4%	74.5%	63.6%	52.1%
1CNN+PixelRNN	**91.1%**	**88.9%**	**83.6%**	**71.8%**

References

[1] Laurie Bose, Piotr Dudek, Jianing Chen, Stephen J. Carey, and Walterio W. Mayol-Cuevas. Fully embedding fast convolutional networks on pixel processor arrays. In *Computer Vision – ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXIX*, page 488–503, Berlin, Heidelberg, 2020. Springer-Verlag. 5, 7

[2] Congqi Cao, Yifan Zhang, Yi Wu, Hanqing Lu, and Jian Cheng. Egocentric gesture recognition using recurrent 3d convolutional neural networks with spatiotemporal transformer modules. In *IEEE International Conference On Computer Vision (ICCV)*, 2017. 5

[3] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau,

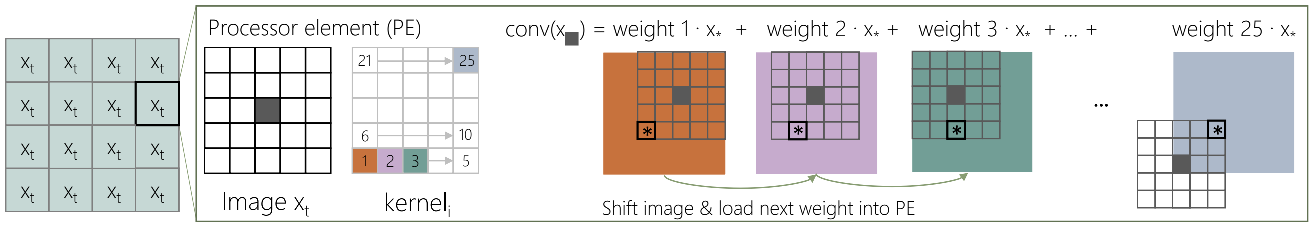
Processor element (PE)
conv(x) = weight 1 · x* + weight 2 · x* + weight 3 · x* + ... + weight 25 · x*
Image x_t
kernel$_i$
Shift image & load next weight into PE
21
25
6
10
1
2
3
5

Figure 2. **In-Pixel CNN.** The CNN and the gates in our RNN are convolutional. Above is a block diagram of the data movement in SCAMP-5 to calculate a convolution. The 256×256 image plane is split into a 4×4 grid, with each space containing 64×64 processor elements (PE), each performing a parallel convolution between x_t and a kernel. In a convolution, the first weight in the 5×5 kernel multiplies with the image, and the result is added to a running sum in the pixel. The image is then shifted and the next weight is loaded, repeating this process for all 25 weights. See Bose et al. [1] for additional details.

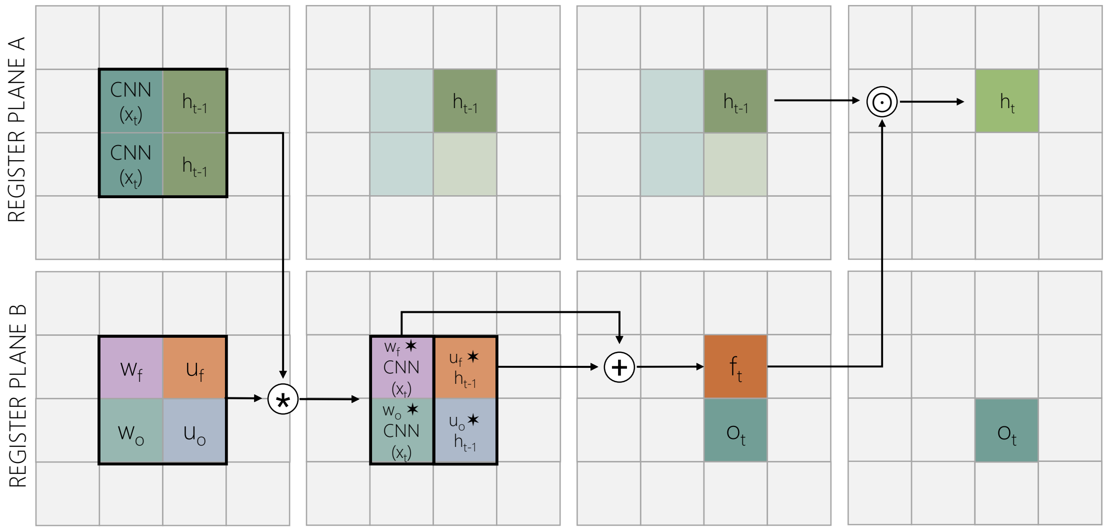

Figure 3. To implement the PixelRNN on SCAMP-5, the image plane is split into a 4×4 grid shown above, each grid space comprising a 64×64 block of processor elements. We make use of two analog register planes, A and B, in the four central PEs blocks. Above, we show the sequence of operations from left to right. The input from the CNN and the previous hidden state are duplicated in A. These 4 blocks of 64×64 PEs are convolved $*$ with the corresponding gate weights stored in plane B. The resulting convolutions in the second column are then added to compute the output $\mathbf{o}_t$ and the forget gate $\mathbf{f}_t$. Note that an in-place binarization is applied to $\mathbf{f}_t$. The hidden state $\mathbf{h}_t$ is updated via an element-wise multiplication $\odot$ of $\mathbf{h}_{t-1}$ and $\mathbf{f}_t$

and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, Doha, Qatar, 2014. Association for Computational Linguistics. 2

[4] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997. 2

[5] Lu Hou, Jinhua Zhu, James Kwok, Fei Gao, Tao Qin, and Tie-Yan Liu. Normalization helps training of quantized lstm. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2019. 4

[6] Okan Köpüklü, Ahmet Gunduz, Neslihan Kose, and Gerhard Rigoll. Real-time hand gesture detection and classification using convolutional neural networks. In *2019 14th IEEE International Conference on Automatic Face & Gesture Recognition (FG 2019)*, page 1–8. IEEE Press, 2019. 5, 6

[7] Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-kin Wong, and Wang-chun Woo. Convolutional lstm network: a machine learning approach for precipitation nowcasting. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, page 802–810, Cambridge, MA, USA, 2015. MIT Press. 2

[8] Yifan Zhang, Congqi Cao, Jian Cheng, and Hanqing Lu. Egogesture: A new dataset and benchmark for egocentric hand gesture recognition. 20(5):1038–1050, 2018. 5

[9] Guo Bing Zhou, Guo Bing Zhou, Chen Lin Zhang, and Zhi Hua Zhou. Minimal gated unit for recurrent neural networks. In *International Journal of Automation and Computing*, pages 13, 226–234. Springer, 2016. 2