Neuromorphic Computing: Simulating the Brain's Visual Cortex

Team 26, Los Alamos High School

April 7, 2021

New Mexico Supercomputing Challenge
Final Report

Team Members

- Robert Strauss

Teacher

- Dr. Charlie Strauss

Mentor

- Dr. Garrett Kenyon

Contents

1 Executive Summary

Neuromorphic systems have the potential for thousands of times more energy efficient computation, if we could only figure out how to program them. Silicon retina have the potential for very high frame-rate, very high dynamic range recording, but have to be processed to reconstruct recognizable data. I trained spiking neural networks to classify moving spiking handwritten digits from silicon retina data, and to reconstruct still images from silicon retina data. To do this I used an extension the back-propagation algorithm from artificial neural networks. I iteratively tested different model architectures, going through redesign after redesign and eventually found the optimal architecture and capacity for a spiking neural network digit classifier. The final model chose the correct digit 95% of the time. I found the silicon retina to image reconstruction spiking neural network generalized to images of all sorts of classes, including natural images despite only being trained with handwritten digits. The network often produced black-and-white, inverted, line-drawing versions of the images because of what it learned to infer from handwritten digits. I discovered the way it produces an output is like a hand writing a digit, by tracing a path out from one end to another. This showed the potential for spiking neural networks to do inference tasks, be more efficient than artificial neural networks, and generalize very well.

2 Background

2.1 Neuromorphic Computing

The biological brain does many very complex tasks, such as object recognition, visual perception, and fine motor tasks, which require many calculations. Traditional digital computers can be programmed to do these things, but at high resolution and in real-time these require lots of power, usually on the order of hundreds of watts, and thus produce lots of heat and require cooling.

Many interrelated activities, like seeing a ball and catching it, are done simultaneously. The brain does all that in small volume and real-time, yet even without cooling fans it doesn't overheat, and requires very little power. Could we achieve this with a traditional computer? It seems unlikely to be possible with digital, Von Neumman, computing, but we also don't want to give up the advantages of digital computing and go fully analog.

How does the brain do it?

Neuromorphic computing is a different architecture inspired by the neuron transmission line signal processing in the brain. Discrete pulses ("spikes") travel down transmission lines (synapses) and neurons accumulate charge from the rate of spikes (not their amplitudes) then fire their own spikes. In such a system, all the neurons can send and receive spikes asynchronously; there's no clock, and a downstream neuron isn't waiting for an upstream neuron to finish its computation. The only speed limit is the speed of light. Memory and processing are effectively stored in the neurons themselves, eliminating inefficient back and forth between a digital memory bank to retrieve data or instructions (requiring many transistor logic gates and clock cycles to toggle for every bit of data or instruction fetched, and bottle-necked over a very limited number of parallel bus lanes).

A set of interconnected neurons, because it is a network of neurons that spike, is called a spiking neural network, unsurprisingly. Because the name is similar it's important for the reader to understand that these are very different from what is generally called an artificial neural network. Artificial neural networks are algorithms that run on traditional computers. Digital values are inputs and outputs of nodes which internally perform linear algebra and non-linear math functions of finite size digital values. Everything is synchronized and before the neuron in an ANN can begin to compute, all the inputs have to be valid digital values, usually the result of the completed calculations by a prior layer of neurons. Parallelism is weak, hardware bandwidth limited, and most of the operations have to be serial.

Spiking neural networks have their own hardware, which could even be a trivial single transistor analog electrical circuit. They use a leaky integrate-and-fire neuron model, in which each neuron collects spikes from many other neurons and sends its own spikes to many other neurons. When a neuron receives a spike, its membrane potential increases (which might be the charge on a capacitor). After enough spikes, it crosses a threshold and emits a spike, resetting its membrane potential. It also slowly leaks off membrane potential via exponential decay (which might be a resistor). Thus the neuron won't fire if the average input spike rate is below a threshold set by that decay rate. Neurons also have a refractory period; a duration after spiking in which it cannot fire another spike. This sets an upper bound on the firing rate.

Neuromorphic computers can potentially use thousands of times less energy than a traditional computer. Intel's Loihi chip, one existing neuromorphic computer, is demonstrated to have orders of magnitude better energy efficiency than a digital computer of similar capacity [2].

The catch is that it's extremely difficult to make a neuromorphic computer do a task because they can't be controlled with a programming language to implement an algorithm like traditional computers. **Instead, I wanted to see how well neuromorphic computers could be programmed, or trained, to do a task.**

For this I built a model of the human visual cortex.

2.2 Silicon Retina

A silicon retina is a type of video camera which emits spike signals rather than frames. When a pixel changes in intensity by a large enough amount, the silicon retina fires a single spike in that pixel's synaptic channel. There are two spike channels per pixel to signify a negative change or a positive change.

In a silicon retina there is also no notion of a clock, as there is in a conventional framing video camera; the pixel fires a spike asynchronously when its light intensity changes. This means it has a temporal resolution much finer than that of typical cameras, equivalent to a frame-rate in the megahertz. A very good framing camera could replicate this kind of frame-rate, but would be inefficient because it would be recording data values for all the pixels even when most of them have not changed. It also would have a much more serious defect: pixels in a framing camera have some maximum dynamic range (e.g. 8 bits) and this puts a limit on the dimmest change or brightest signal it can report. A small variation on a bright signal is poorly resolved or even lost if the pixel is already at the largest value (e.g. 256). A silicon retina also has some limits but it is much easier for them to see small changes on top of a large signal without saturating since they report change not brightness.

It sounds perfect – an efficient, very fast camera with very high dynamic range. But the trade-off of the silicon retina's change-based output is that if nothing in the scene is moving or changing, the camera produces no output. A simple solution to this is to move the camera back and forth or equivalently move something in front of the camera periodically. Human eyes do this; they perform jerking motions rather than steady gazing, know as saccadic motions. Another issue is that reconstructing a usable output in the form of a frame-based video isn't as simple as integrating the changes: the initial intensity pattern is not known, and small errors or slight lack of precision will integrate and grow. The silicon retina data is incomplete in this sense.

Recovering a normal video thus requires inference.

2.3 My model of the Visual Cortex

Clearly silicon retinas are well matched with spiking neural networks. So it's a very logical step to connect a neuromorphic computer to a silicon retina. The silicon retina will be the eyes of the system, and the spiking neural network the brain providing inference. This will also allow for experimentation with controlling a spiking neural network and processing silicon retina data.

3 Introduction

3.1 The Problem

Neuromorphic computers have the potential for completely parallel, scalable, very energy efficient computation. However, programming one is difficult.

Silicon retinas have the potential for very high frame-rate, high dynamic range, storage space-conserving video recording. However, getting usable data from them is an under-specified problem.

3.2 The Question

How can neuromorphic computers be controlled to perform a specified task? In this case, how can it be "programmed" to "see".

3.3 The Task

The idea of "seeing" something can be broken into two components. One is to visualize it, meaning to form a mental image of it. In the case of silicon retinas, this means integrating spikes through time ti form an intensity image, stabilizing the image, and inferring parts from incomplete data. The second component in "seeing" something is recognition of what it is. That's a classification task, and is commonly done with digital Artificial Neural Networks with already well-formed images. In this case, a spiking neural network will be doing this with moving, spiking, and incomplete silicon retina data.

One does not have to do these tasks sequentially. I note that for quick reaction times it might even be preferable to separate these tasks in the brain:

you would prefer to recognize a tiger lunging at you in the dark jungle before taking time to de-blur and visualize a recognizable image of it.

I set out to examine these two tasks by developing two separate spiking-based systems:

1. Classification: train a spiking neural network to identify the numeral represented by a moving handwritten digit recorded by a silicon retina.

2. Reconstruction: train a spiking neural network to reconstruct a still image from a silicon retina recording of a moving image.

4 Methods

4.1 Simulating and Training a Spiking Neural Network

Since I did not have access to physical neuromorphic hardware, I simulated a spiking neural network, modelling the actual voltage waveforms propagating from neuron to neuron. I did this by applying an Ordinary Differential Equation (ODE) solver to each neuron, modeling a leaky integrator's response to the spikes it receives, applying a refractory period and spike threshold, and generating an output spike waveform. Spike signals are convoluted with a Poisson spike kernel to simulate the temporal dispersion of a spike through a synapse transmission line [9]. Each synapse has a coefficient multiplying the spike signal, akin to the weights of an artificial neural network. Synapses can also have time delays, which offset the spike signal by a fixed positive amount of time.

To teach the spiking neural network what to do, I score it by taking the difference of it's output and the correct answer (a value called loss or error), and then correct for this error by applying the chain rule to calculate how to change the parameters to decrease the error (a process called back-propagation). The parameters here are the coefficients and delays applied to the signals by synapses, and these are updated by stochastic gradient descent. This is a common way of training *artificial* neural networks. But there are a few complications with spiking neural networks. The back-propagation has to go not only through multiple layers of neurons, but also through the ODE solver simulating the temporal evolution of each neuron. Complicating this, the derivative must pass through the spike threshold, a Heaviside step function, which is non-differentiable. To solve this, the derivative is replaced

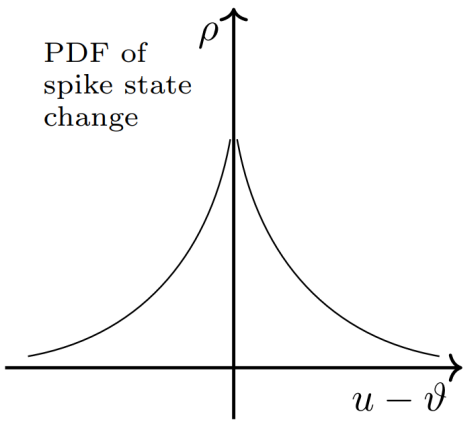


Figure 1: The probability of a neuron spiking if the membrane potential (u) changes by a small amount (v) of unknown magnitude. Credit to Sumit *et al.* [9].

by a probability distribution (figure 1) modeling the chance the membrane potential exceeds the threshold after a tiny change to it (a sort of probabilistic dx). This can alternatively be thought of as a blurred-out approximation of the actual derivative, which is the Dirac delta function, so there is a gradient to learn on.

In artificial neural networks, back-propagation takes the gradient through the model to calculate how to change the weights in the last layer to decrease the error. Then it uses the gradient again to calculate how to change the membrane potentials in the previous layer to decrease the error. It continues to the weights of the layer before that, then the neuron values, and so on. But in a spiking neural network, membrane potentials don't propagate down synapses, spikes do. But the gradient of the spike function can't tell us how to change the membrane potential to change the spike train. This replacement of the spike function's derivative gives the back-propagation algorithm a gradient, so it knows how to change a membrane potential to change a spike train. In machine learning terms, this is a way of encoding the understanding that higher membrane potential means higher chance of spiking in the prior distribution.

This protocol for simulating and training a spiking neural network is not my own work, but the work of Bam Sumit *et al.* [9].

4.2 Implementation Approach

To do this I used Python to specify, construct, and test alternative neural synaptic architectures, drive Differential Equation solvers modeling the temporal dynamics of the neural response, back-propagate loss across time as well as neurons for training, and evaluate the trained networks on test sets. The operations were carried out on a GPU for speed given the intense computations needed for ODE solving and stochastic gradient descent on networks with over 100,000 neurons. I also used Python to track the training and testing, create animations, images, and graphs as well as to load and batch the enormous raw time series silicon retina data for training.

All of my annotated code is available online at github (`https://github.com/robertstrauss/spikingneuralnetworks`). I wrote well over 1,000 lines of original code developing this, using the Jupyter Notebooks [7] interface and incorporating varied numerical, graphical, and data-streaming libraries (see annotated code for details.) On top of that I wrote my own user interface module [10] for Jupyter Notebooks in JavaScript, HTMK, and CSS.

I used libraries including Matplotlib [6], PyTorch [8], TensorBoard [1], TorchVision [8], and SLAYER [9]. Matplotlib is a visualization library that can plot data in different forms such as images and line plots. PyTorch is a library commonly used with artificial neural networks, with high-performance linear algebra and back-propagation capabilities. TorchVision is related to PyTorch and had tools for manipulating image data and acquiring some of the data-sets I used. SLAYER, as discussed in other sections of this paper, adds capabilities for simulating neurons and back-propagating through the neuron model.

The final code set is too large to include here but can be browsed at github: `https://github.com/robertstrauss/spikingneuralnetworks`. Jupyter Mosaic, my extension of Jupyter Notebooks, is also on github: `https://github.com/robertstrauss/jupytermosaic`.

In the past two years I have written GPU kernels for solving systems of differential equations (large scale, high performance computing ocean wave modeling) as well as worked to backpropagate through differential equations to train Neural Differential equations. This made me especially well suited for this project because I had experience with neural networks, differential equations, and back-propagation through the combination of the two. However, for this project I started from scratch, using none of that code, and instead took advantage of new libraries (such as SLAYER and PyTorch) that

implement the underlying methods needed to simulate synaptic networks and apply Bam Sumit *et al.*'s training protocol [9]. This allowed me to focus on the bigger problem: experimenting with designs for visual cortex architectures and processing silicon retina data.

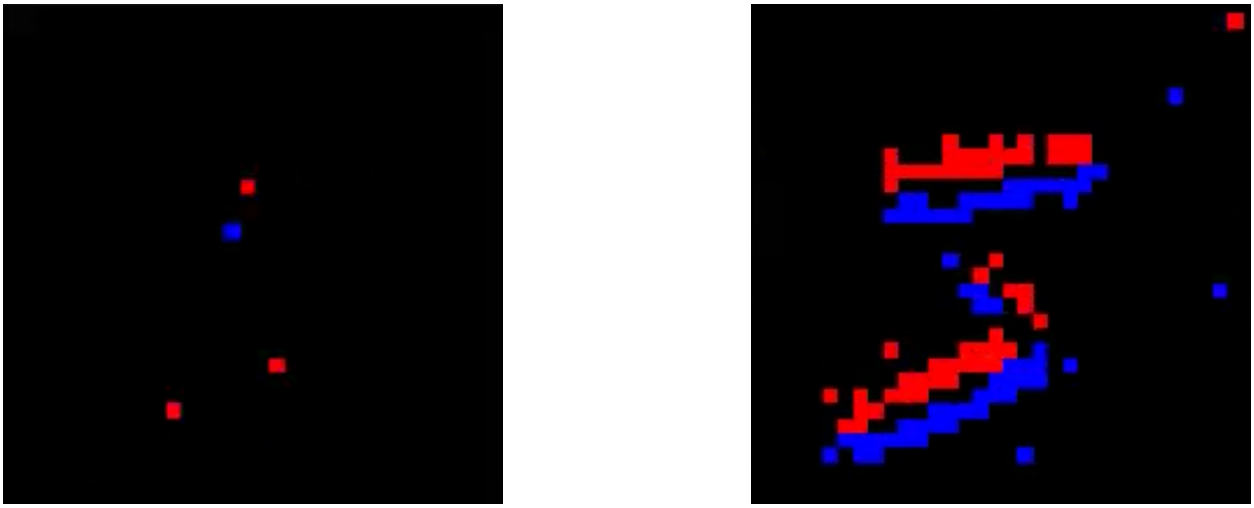
4.3 Data

MNIST is a public data-set of images of hand drawn digits that is well-known in machine learning. The images show many different handwriting styles. There are tens of thousands of examples. The MNIST data-set is conveniently pre-processed so all the digits are centered, are of the same size, have the same contrast, and are at the same resolution.

The Neuromorphic-MNSIT [4] (N-MNST) is an adaptation of the MNIST for spiking neural networks made by Garrick Orchard. The images were displayed on a monitor with a real silicon retina was pointing towards it. The silicon retina was moved around in three saccadic motions by a mechanical device to generate spiking recordings of the moving MNIST handwritten digits. This means the data have unknown experimental noise and imperfections.

Caltech-101 is another data-set with many other images of various types, depicting things such as airplanes, motorcycles, animals, drawings, and more. The images are classified into 101 categories based on what they contain. Neuromorphic-Caltech101 [3] is the neuromorphic adaptation of this, also by Garrick Orchard, created in the same way. While the digits in MNIST are line drawings with carefully controlled contrast, the Caltech-101 set are varied images including full color natural photographs with varying sizes and shapes and thus lack the more distinct edges, feature widths, and similar size and resolution of the handwritten digits.

Still-image examples of the (moving) silicon retina recordings from these data-sets can be seen in figures 2a and 2b). It's impossible to show these sparse spiky videos in a printed image. To help the reader see this slightly better, in figure 2b, the data has been pre-processed to give the reader an impression of what digit is moving and how it leaves trails of positive- and negative-going spikes as it moves. The raw silicon retina data looks more like figure 2a. But even that doesn't show the temporal fluctuation and visually random sparsity of these spike events well. The spiking neural network was not given these enhanced pre-processed images. It was only given the raw data of spike time series from the silicon retina.



(a) (b)

Figure 2: 2a: A still image showing a few spikes of different polarity (red/blue pixels) to give an idea of what kind of data is in a silicon retina recording. This is not a frame because silicon retinas do not have frames, so these spikes couldn't be captured at exactly the same time like this. 2b: A visualization of the moving digit recorded by the silicon retina. The spikes shown occurred over a spread out period of time, and are grouped together to make sense of the data. Notice the digit leaves a trail of negative spikes on one side of the line, and a trail of positive spikes on the other side. The network does not get data that looks like this, this is just for visualization purposes.

5 Model architectures

In machine learning for computer vision, neurons in a neural network are arranged into 2-d grids of values representing "images," or in this case time-series spike signal "videos." A convolution multiplies a small feature image, or kernel, and the region of a larger input image it overlaps and sums the product, then moves the feature image over and repeats this for every position of the feature image on the input image. This produces an output image as each positioning of the feature is reduced to one value, or pixel. A convolution can also have multiple channels, which means the input and the features are actually stacks of images. If a convolution has multiple features, the output is a stack of images where each image is the output from a convolution by one of those features. The pixels of the feature image are the trainable parameters in a convolutional neural network.

The classification model had 3 convolutional layers, reducing a 34 by 34 silicon retina recording with 2 channels to a 28 by 28 spike time series with 32 channels, followed by one fully connected layer, reducing the output to a 10-long one-hot vector. Figure 3 is a diagram of this architecture.

The reconstruction model was entirely convolutional, with 4 convolutional layers reducing a 34 by 34 silicon retina recording with 2 channels to a 28 by 28 intensity image with 1 channel. Figure 4 is a diagram of this architecture.

In the diagrams, each layer of neurons is shown with a stack of squares, each square representing one channel of an image. The kernels for convolution are shown with smaller squares connecting to the next layer. The dimensions of the layers are written above, in the format number of channels @ width x height.

6 Results

After training a neural network, it is quite common to see it get 100% correct answers when evaluating it. This is generally not because it mastered the task, but because it memorized the data it was trained with (the training set). So to evaluate models, one holds out a subset of the data called the testing set. If it performs worse on the testing set, the model likely memorized data it was trained with and did not learn the general rules. This is called over-fitting or over-training. I periodically evaluated the model's accuracy (portion of examples it classified correctly) on the testing and training sets as it trained to see its progress.

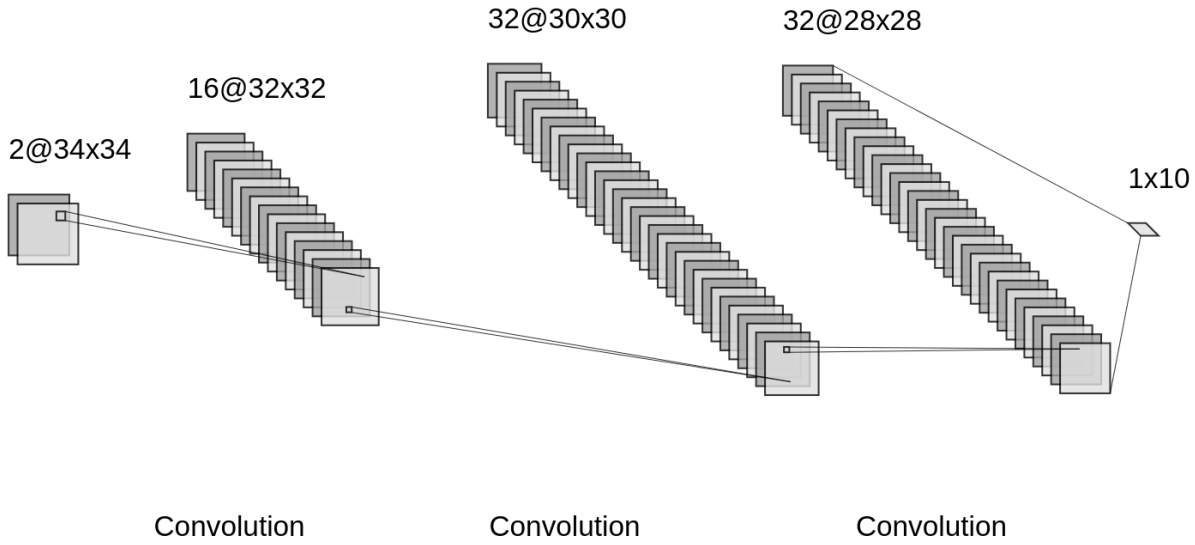


Figure 3: A diagram of the architecture of the classification model. Each layer is a a stack of squares representing the channels of an image. There are three convolutional layers and one fully connected layer. The input is 34 by 34 with 2 channels, the size of the N-MNIST silicon retina data. The output is a 10-long vector, where each channel codes for one of 10 answers.

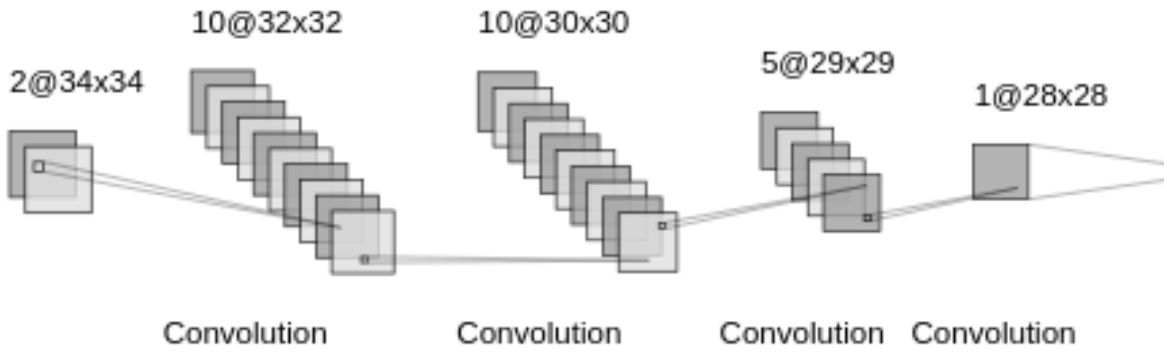


Figure 4: A diagram of the architecture of the reconstruction model. Each layer is a a stack of squares representing the channels of an image. There are four convolutional layers. The input is 34 by 34 with 2 channels, the size of the N-MNIST silicon retina data. The output is 28 by 28 with one channel, the size of the MNIST digit images.

6.1 Classification

To gauge accuracy, I divide the number of examples in which the model got the answer right, putting the most spikes into the channel for the correct digit, by the total number.

In figure 5a, the blue line initially increases along with the orange line, but diverges as the orange lines begins to plateau around 0.6. The blue line accuracy continues all the way to almost 1.0, or 100%.

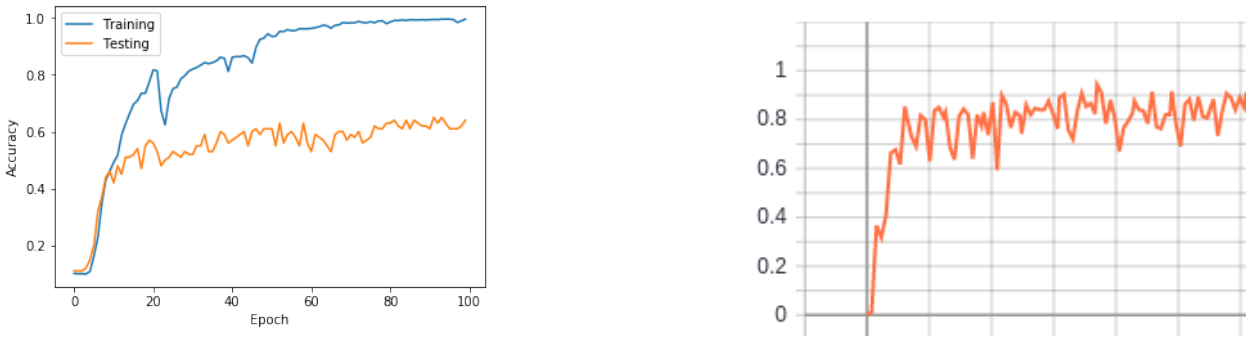
In this figure, the blue line is the training set accuracy of the first model I created and the orange line is the testing set accuracy of the first model. This means after being trained, the model had near 100% accuracy on the training set, but much lower, 70%, accuracy on the testing set. This indicates over-fitting: the model was memorizing the examples it trained on, not learning how to recognize digits.

I improved the model incrementally by changing the architecture depending on how well it performed. Over-fitting occurs when the model's capacity, the number of neurons or parameters it has, is very large. Over-fitting happens because a model with large capacity can memorize tiny unimportant features of training examples rather than learning the general rules [5]. For example, rather than learning to recognize a handwritten six, the model may memorizes the tiny imperfections that occur in the sixes in the training set. This leads to worse performance on the testing set because it has never seen the examples in the set and so can't use any tiny details it memorized.

To prevent this, counter-intuitively, one can dumb the model down by reducing the capacity so it has less space to memorize tiny details and can only fit general rules. Another solution is to increase the size of the data-set, so there are so many examples it can't memorize all the specific features and has to learn the general rules instead. I used these two techniques to improve the generalization of the model. I made models with smaller capacities until I went too low and the model could hardly predict better than random guessing even on the training set. Once I found a good capacity, the testing and training accuracy were similarly high. This shows generalization rather than memorization.

The final architecture (figure 3) achieved a testing accuracy of 95% (figure 5b).

I also tested adding random noise. With random noise, which added on average 600 spikes per pixel over the duration of the recording, the accuracy of any model decreased by about 5% on average.



(a) (b)

Figure 5: 5a: Accuracy (percent of answers correct) of the first model on the training set (blue) and held-out testing set (orange) as it trains. X-axis is number of epochs it has trained for. The blue line reaches higher than the orange line, meaning the model had better accuracy on the training set than the testing set. This means it was over-fitting. 5b: Testing accuracy of the final model on the held-out testing set as it trains. X-axis is number of epochs it has trained for. The line reaches a maximum of 0.95, or 95 percent accuracy, a great improvement over the previous models.

6.2 Reconstruction

In this case, because there isn't a simple correct answer, I can't score the model with accuracy to evaluate it. The model has to reconstruct a still intensity image, so I sum the number of spikes in a pixel and use that as the intensity. This means it doesn't have to send spikes to a pixel at a constant rate to give that pixel a high intensity. It could give that pixel a lot of spikes at the beginning, or wait till later. To judge the performance of the model, I looked at it's output on testing set examples myself.

Figure 6 shows the model output on testing set images as it trains, side by side with the true image. As you can see, by the end of training it was producing images sufficiently well.

I wondered how it was deciding to send spikes into a pixel. I expected it to wait for a moment while it tried to figure out what it was seeing, then spike all the pixels that it thought were bright. But what I found was much more interesting. Instead, the model starts spiking the pixels on one end of the line its drawing, then moves down the line, much akin to how a human draws a digit by starting at one end putting down ink as they trace out the digit.

The model is also not just producing the same, generic 9 (for example)

when it sees a 9, but producing a digit with the specific features and handwriting style shown in the true image. This shows the model is learning general rule of reconstructing an image from silicon retina data and not memorizing specific examples. Images within the same class have a lot of variation in details such as slant, stroke thickness, loop size, smoothness, and other small details of different handwriting styles. The reconstructions mimic these, even in the testing set, showing the model is learning general rules about how to reconstruct an image from silicon retina data and not memorizing specific examples.

When applied to Caltech101 images, containing many types of images that aren't handwritten digits, the model still produced output similar to the true images. This, shown in figure 7, is further proof of generalization. In many cases, the reconstructions are inverted, with a black background rather than a white one, and often only show the edges in the image. These outputs look like white on black line drawings versions of the true images which have smooth gradients and shaded backgrounds. It is surprising the model can generalize to a completely different domain of images. The fact it makes white on black line drawings shows that it learned from the handwritten digits to infer the background is black and to draw white lines where there are sharp gradients. This shows it is applying its knowledge of reconstructing an image, which is limited to white on black line drawings of digits, to other images.

Additionally, the Caltech-101 images are of different shapes, sizes, resolutions, and even aspect ratios. Because of the convolutional architecture of the network, it can process any size or resolution of image the same way. (Since the silicon retina data doesn't contain information about color, only intensity, the Caltech images are converted to grey-scale.)

7 Discussion

The two models, the image reconstructor and digit classifier, were entirely separate. The reconstructor was *not* an intermediate stage to make classification easier. The classifier did not reconstruct a still image first, but had to directly classify the digit from the moving silicon retina data. It could potentially have learned to stabilize and integrate the data into a still image, but I did not enforce this. It would be an interesting next step to combine the reconstruction model with another classification model. This could improve the

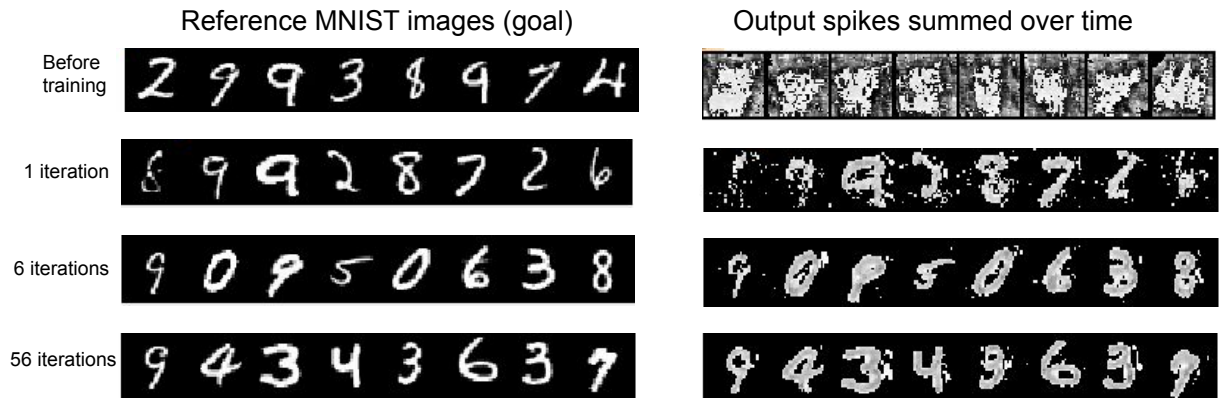


Figure 6: Model reconstructions (right) from silicon retina data of moving images of original MNIST digits (left) on examples from the testing set. With more training, the reconstructions more and more resemble the true digit images. Each reconstruction is faithful to the specific features and handwriting style of the true image, showing it is learning how to reconstruct image data from a silicon retina, not just how to draw digits.

classification accuracy, since classifying still intensity images of handwritten digits is a solved task, with state-of-the-art models getting 99.9% and higher accuracy.

Before training, a model is initialized with randomly generated parameters. This causes the initial outputs to also be very random. In figure 6, the reconstruction from the beginning of training is not purely random noise, there are clumps of higher pixel intensities near the middle. This is because of the properties of the convolutional neural network. A convolutional network with random parameters will tend to spread out, or blur out, input. This explains the concentration of pixels in the center: the digit is centered. Additionally, each layer pads the edges of the image with zeros, to prevent the image from becoming too small as it gets reduced in size by each layer. This explains the darker edges: they weren't getting as much input and had a lot of zero-padding.

8 Conclusion

I have designed spiking neural networks that generalize very well. After many iterations of designs, the classifier I created is able to determine what digit is in the silicon retina data 95% of the time. The image reconstructor I made

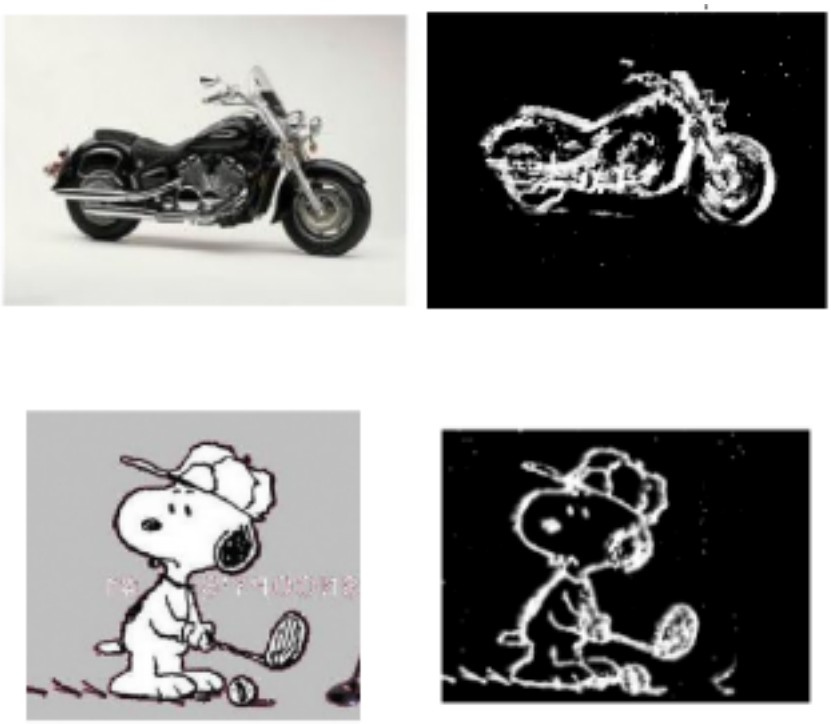


Figure 7: Model reconstructions (right) from silicon retina data of moving image (left). The model was only trained on handwritten digits, but it partially generalized to other types of images, creating recognizable reconstructions of the images. Interestingly, it reconstructs negatives of the images. This is likely because all the digits it was trained on had black backgrounds. It must have learned to infer a black background, and to draw white lines where there are sharp edges.

not only reproduces testing set images very well, including small details in the reconstruction, but can generalize to a completely different domain of data, reconstructing various types of images. The inferences it learned to make from handwritten digits cause an interesting behaviour in the reconstruction of non-digit images. It often produced inverted images, swapping black for white, and only highlighted the edges. I also discovered it drew out the reconstructions almost like a person with a pencil.

All this shows spiking neural networks have a great potential for not only more efficient computation, but great generalization.

References

[1] Martın Abadi et al. "Tensorflow: A system for large-scale machine learning". In: 12th {USENIX} Symposium on Operating Systems Design and Implemen 2016, pp. 265–283.

[2] Figure 5. A comparison between Movidius NCS and Loihi for inference... ResearchGate. URL: https://www.researchgate.net/figure/A-comparison-between-Movidius-NCS-and-Loihi-for-inference-speed-and-energy-cost-per_fig2_335373358 (visited on 01/18/2021).

[3] Garrick Orchard - N-Caltech101. URL: https://www.garrickorchard.com/datasets/n-caltech101.

[4] Garrick Orchard - N-MNIST. URL: https://www.garrickorchard.com/datasets/n-mnist.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. MIT Press, 2016.

[6] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: Computing in Science & 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[7] Thomas Kluyver et al. "Jupyter Notebooks – a publishing format for reproducible computational workflows". In: Positioning and Power in Academic Publ Ed. by F. Loizides and B. Schmidt. IOS Press. 2016, pp. 87–90.

[8] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[9] Sumit Bam Shrestha and Garrick Orchard. "SLAYER: Spike Layer Error Reassignment in Time". In: Neural IPS (2018).

[10] Robert Strauss. “Jupyter Mosaic – a Jupyter Notebook extension for creating customizable flexible code layouts”. In: 2021. URL: https://github.com/robertstrauss/jupytermosaic.