Summer 2019

Zefyr Scott

Exploring Use of Neural Networks to Train a Multiple Encoder Quantization System for Classification

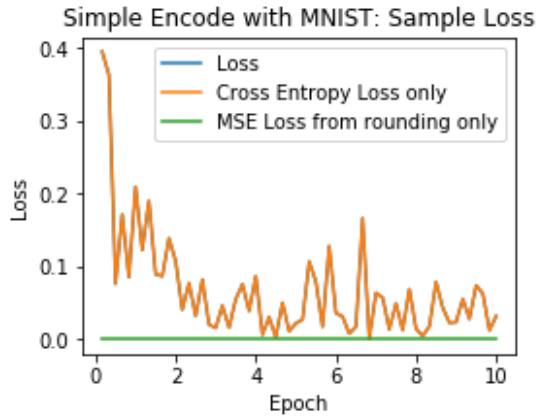
One of the problems of communications systems is the size of the data being communicated. This size is limited by how much it is quantized. If quantization is very small and the data is nearly continuous that size is also very large. Large quantization can bring the size of the data down considerably, but that may be at the cost of some of the desired information. Data may also be broken up into smaller portions so that each portion is a smaller size, or it may originate as separate portions of data. This data may then be combined into the complete set of data (such as a complete image) for interpretation. The purpose of this research was to examine a special case of this data combination. A set of data may be passed through a neural network for classification. Each portion of this data might be quantized at a low bit rate and transmitted over a communications system, then combined and decoded for interpretation by a classification neural network. This encoding and decoding process might be implemented using a neural network. This research explores the effectiveness of such an approach.

The design of such a system is as follows. A set of input data is divided into reasonable portions. For example, an image might be divided into quadrants, or if in color might be divided into red, blue, and green layers. Each portion is processed by an encoder which transforms the data into a designated number of values and makes a soft decision on them. This output for all portions of the input is processed by a decoder, which combines all of these encoded portions into one set of data and transforms it into the appropriate dimensions for the classification network. The portion of the encoding and decoding processes which are implemented by trainable neural network layers may be trained along with the classification network at the end of the process, or may be trained to provide the desired output for a pretrained classifier. Once training has been completed, a hard decision is used on the encoder output instead, such that each value may be communicated using a bit. Therefore the encoder/decoder design problem is not just about producing data which may be accurately classified, but also about minimizing the accuracy gap between classification based on a soft decision and classification based on a hard decision.

Procedures and Data

My initial efforts involved the MNIST dataset. I found an example classifier for this dataset in a tutorial [1]. This had a fairly high accuracy rate, so I used this as my classifier for this dataset as well as adapted it for use as an encoder. This classifier has two fully connected layers separated by a ReLU. I used a hidden layer of 1000 neurons for it as a classifier, which I found to perform better than the size of 500 proposed in the tutorial. Within five epochs, this classifier achieves training accuracy of 98.78% and testing accuracy of 98.01%.

Approaches for encoding this dataset follow. Each of these uses this same structure of two fully connected layers separated by a ReLU, and vary only in how they handle the data after the second fully connected layer. Additionally, I limited the encoder hidden layer to 200 neurons. In each case I flattened



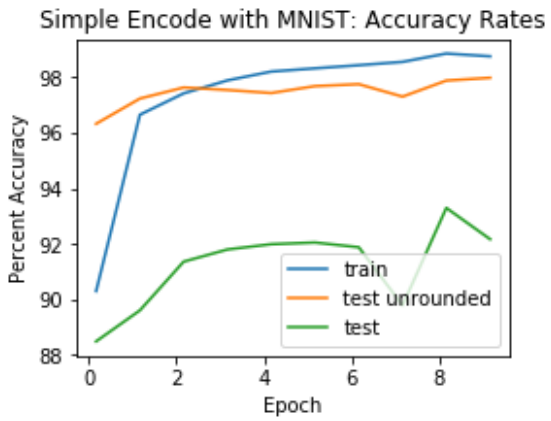
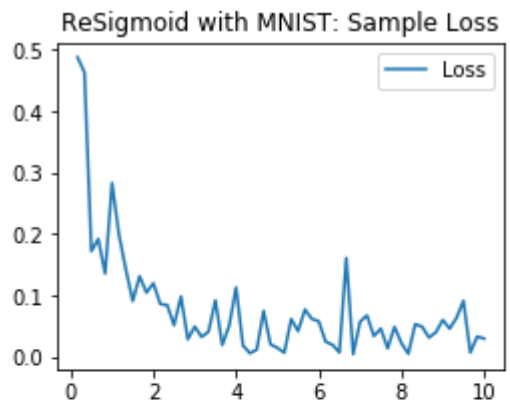


Figure 1



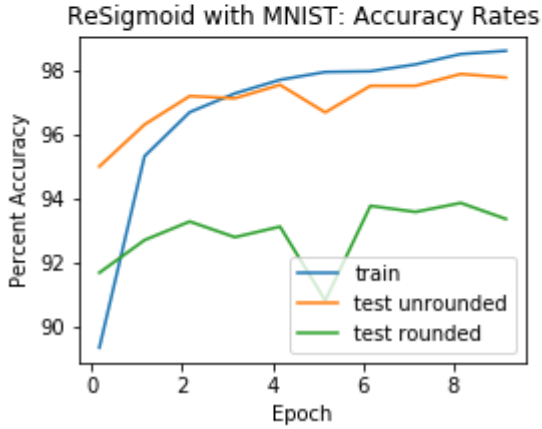


Figure 2

and split the records evenly across four encoders, and each encoder output four bits, for a total of sixteen bits. The batch size used was 100. These used Adam for the optimizer with a learning rate of 0.001, and Cross Entropy Loss for the loss function. I also tried using SGD for the optimizer, but did not find this added any value. These all were decoded identically using a single fully connected layer with no nonlinearity to transform the data back to its original size.

Note that unless stated otherwise, all trials included in this report used a batch size of 100, learning rate of 0.001, and optimizer of Adam. Trials were attempted with different values but these generally obtained comparable or worse accuracy, or simply slowed down the trials. A couple examples are included in this report.

The most basic adaptation of this structure for the encoder was to follow the two fully connected layers with a sigmoid for the soft decision, and then apply rounding during testing for the hard decision. Later I learned to apply a batch norm layer before the sigmoid, in order to slow down training. This is the “Simple 1D” encoder. Without batch normalization, at 9 epochs this had a training accuracy of 98.775%, an unrounded test accuracy of 97.47%, and a rounded test accuracy of 93.66%, and then starts to drop. With the batch norm layer the accuracy after ten epochs is 98.76% for train, 97.99 for unrounded test, and 92.16 for rounded test. See Figure 1 for results with batch norm, and SimpleEncode in supermodel.py.

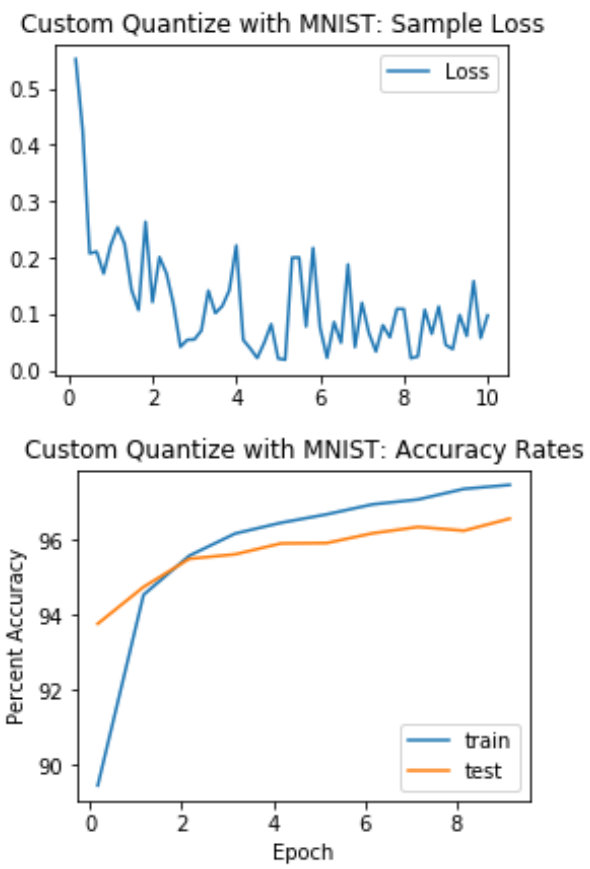


Figure 3

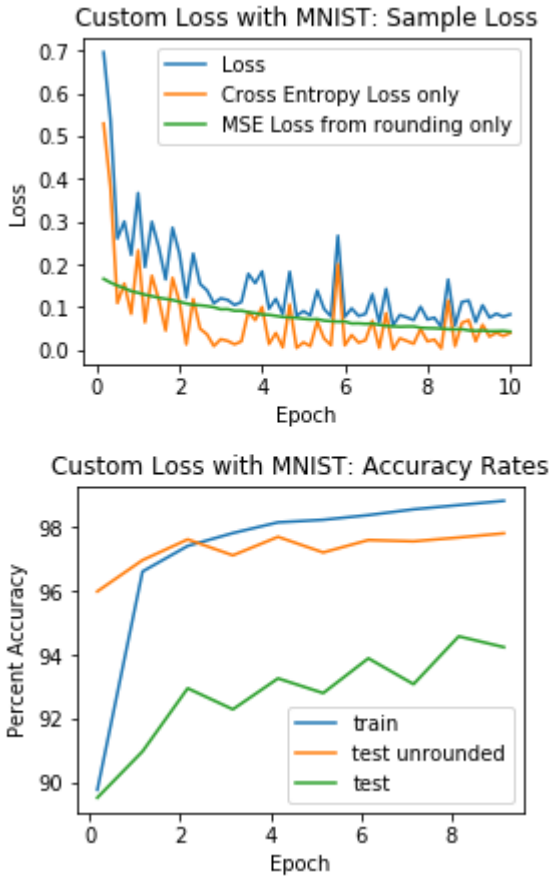
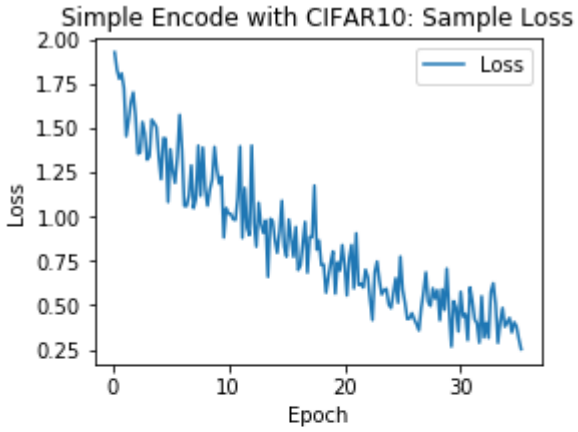


Figure 4

I thought it might be valuable to try to train the system based on encoder output which was as close as possible to a hard decision. One method which I tried was to take the output from a sigmoid, and then scale and shift it and run it through another sigmoid. The total applied function, which I refer to in the code as ReSigmoid, is:

$$R(x) = \frac{1}{1+e^{-(14x-7)}}$$

This scaling and shifting is based on the observation that the sigmoid function maps values in the domain of [-7,7] to the range of values between 0 and 1, while values less than -7 are mapped approximately to 0 and greater than 7 mapped approximately to 1. Repeating this process three times resulted in an output from the encoders that was nearly a hard decision, while still maintaining a nonzero gradient. This is the “Simple ReSigmoid” encoder. With batch normalization before the first sigmoid, after ten epochs this had a training accuracy of 98.6%, with an unrounded test accuracy of 97.77% and rounded test accuracy of 93.35%. See Figure 2, and SimpleEncodeReSigmoid in supermodel.py. Without batch normalization, after ten epochs this achieves a training accuracy of 88.915%. Unrounded and rounded test accuracy are nearly matched, at 90.22 and 90.21% respectively. A problem with this strategy however is that it forces the network to saturate quickly, which limits learning.



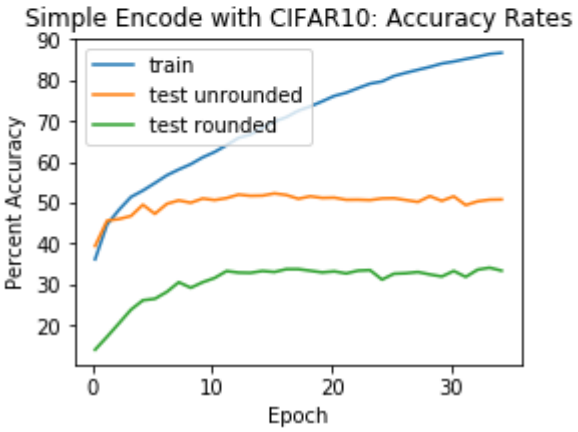
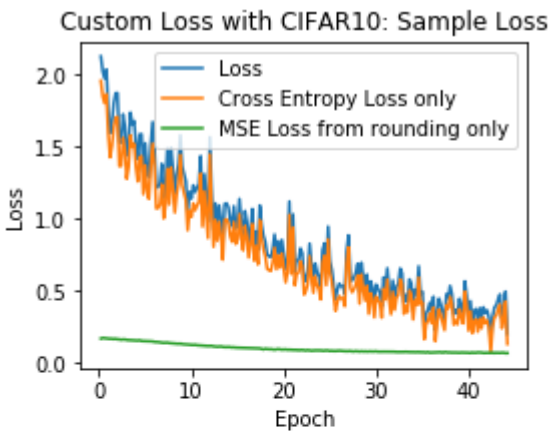


Figure 5



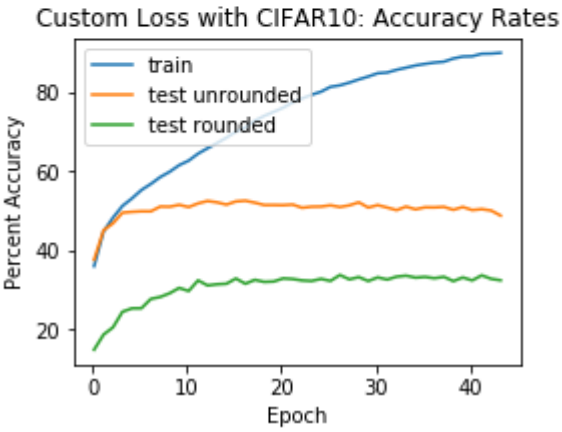


Figure 6

Another method aiming to address the gap directly was to apply a hard decision during training. Normally the gradient of this hard decision would be zero, which would then prevent the network from learning anything before that decision. However, instead I passed the difference between rounded and unrounded values as the gradient: in other words, heavy rounding was treated as a more extreme gradient. This is the "Simple Quantize" encoder. See Figure 3, and SimpleEncodeQuantize in supermodel.py. In ten epochs this method achieved 97.46% training accuracy and 96.56% testing accuracy. However, this is functionality that makes more sense to implement in the loss function.

Through discussion with the lead researcher on this project, we decided to explore a loss function which was the sum of the overall Cross Entropy Loss plus an extra factor. This extra factor was taken as the average MSE loss between the rounded and unrounded values for all encoder outputs of a particular record, thus treating the rounded values as a theoretical target. This average MSE loss was also multiplied by a constant for some trials. See LossFunction and MSELossMean in supermodel.py. This can be summarized as:

$$L = \text{Cross Entropy Loss} + \alpha \left(\frac{\sum_{\text{encoders}} \text{MSE Loss}\left(\text{encoder output, rounded encoder output}\right)}{\#\ \text{encoders}} \right)$$

Using this with the simplest encoder model above for MNIST (including batch normalization) achieved similar results, with 98.83% training accuracy, 97.81% unrounded test accuracy, and 94.24% rounded test accuracy. See Figure 4.

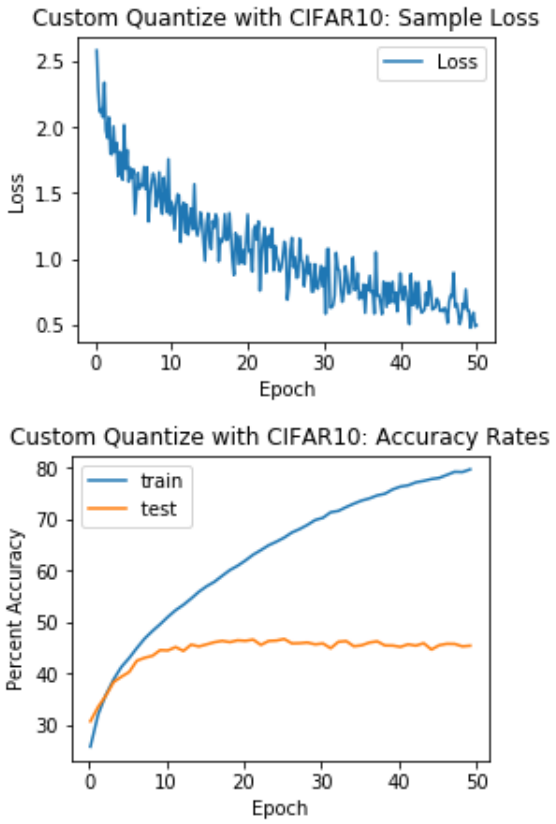


Figure 7

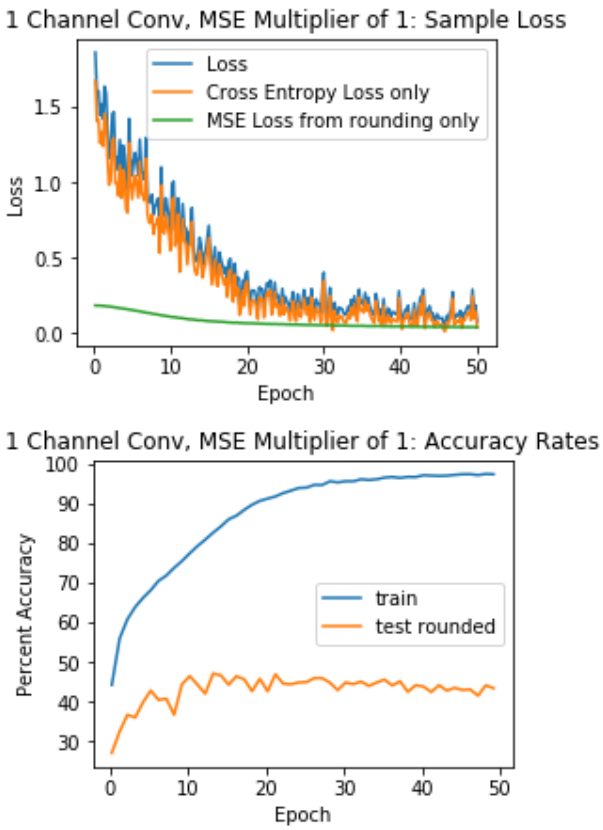


Figure 8

Further testing was pursued with the CIFAR10 dataset, as the simplicity of the MNIST dataset made it difficult to observe the relative success of different methods. After attempting use of several possible classifiers, including AlexNet, ResNet, and WideResNet, the Pytorch Playground pretrained classifier was used [2]. The same fully connected encoder with batch normalization from before had significantly worse performance with either loss function. Example tests were performed dividing the images by color layers into three equal portions, and encoding at 8 bits, for a total of 24 bits. See Figure 5 for application of this encoder with cross entropy loss. Training accuracy climbs to 85.668% after 33 epochs, but the unrounded test accuracy plateaus at around 51% after 10 epochs and the rounded test accuracy plateaus shortly after, at 13 epochs with around 33%. See Figure 6 for application of the custom loss function. These results were not significantly different, with training accuracy climbing to 85.746% after 33 epochs, unrounded test accuracy plateauing around 51% after 8 epochs, and rounded test accuracy plateauing around 31% after 14 epochs. Out of curiosity I also took a look at results of applying the custom quantization function. Although this keeps training consistently through 50 epochs, getting up to nearly 80% train accuracy, the test accuracy saturates at around 10 epochs with 44-45%. See Figure 7.

I also tried using a convolutional encoder. I used two general approaches to this. One was similar to using a convolutional classifier, in which some number of convolutional layers are followed by linear

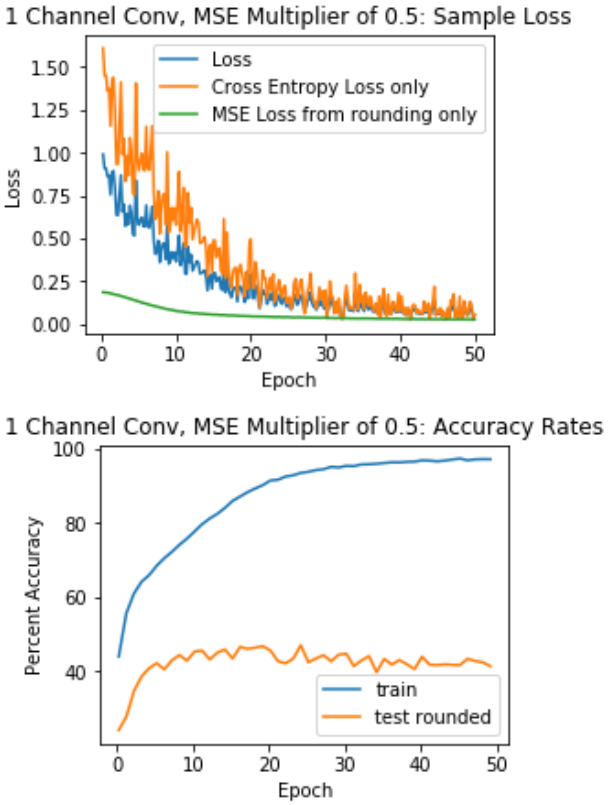


Figure 9

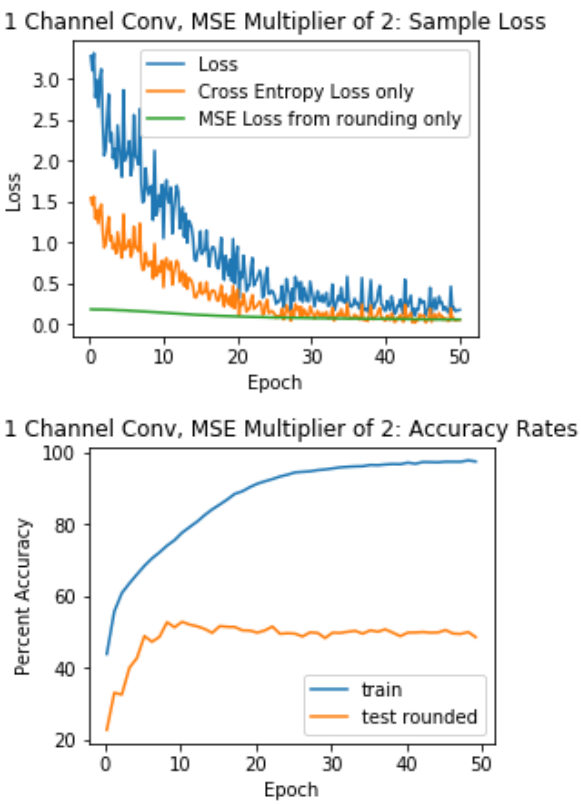


Figure 10

layers. In this case, rather than the output of the final linear layer having a size matching the number of classes, the size matched the number of bits for the output of that encoder. The other approach removed the linear layers entirely. This approach required that the number of bits each encoder output was a square number such that it could be the size of the output of a convolutional (or maxpool or similar) layer. Most of the convolutional encoder trials were attempted based on diving the image into three parts by color layer, but I did also attempt preserving the color and dividing the image into quadrants. To decode the former encoder approach, I used a linear layer and reshaping to transform the combined portions to the correct dimensions. For the fully convolutional encoder approach, I explored combinations of upsampling and padded convolutional layers to bring the combined data back up to size.

One approach for the basic application of a classifier as an encoder used a model from a Pytorch tutorial [3]. The classifier provided by the tutorial uses two convolutional layers, each followed by a MaxPool2d and a ReLU, and each using a 5x5 filter. The first brings input to 6 layers and the second to 16. These are followed by three fully connected layers separated by ReLUs, with output sizes of 120, 84, and finally the number of classes. To use this as an encoder, the output size used is the number of encoder output bits. The final fully connected layer is followed by a batch normalization layer and then a sigmoid, plus rounding when not training. See “1 Channel Conv” in supermodel.py. This was used along with the decoder implemented by a fully connected layer and reshaping; see “1 to 3 Channel Conv” in supermodel.py. This offered a better chance to look at the behavior of the custom loss function when multiplying the MSE portion by a constant. For an output of 12 bits per encoder (RGB separation, so 36 bits total) I tried multipliers of 1, 0.5, and 2. For a constant of 1, see figure 8. Training plateaus around

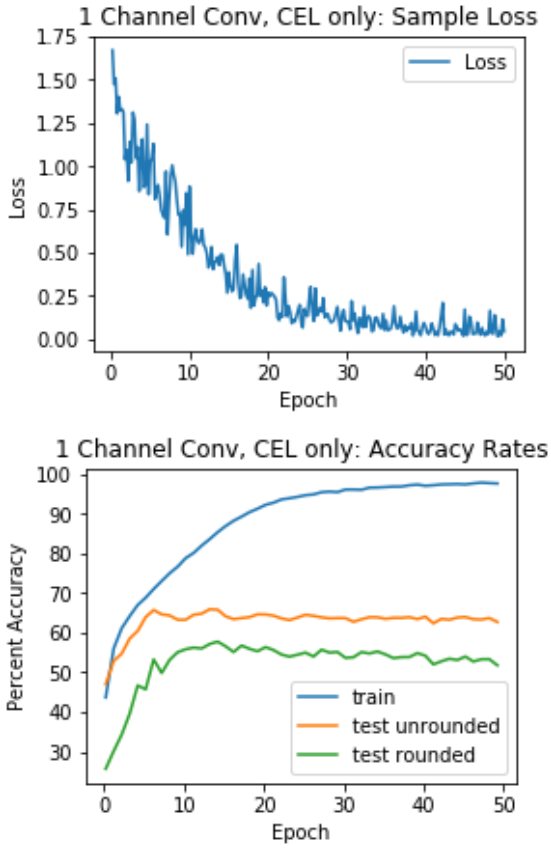


Figure 11

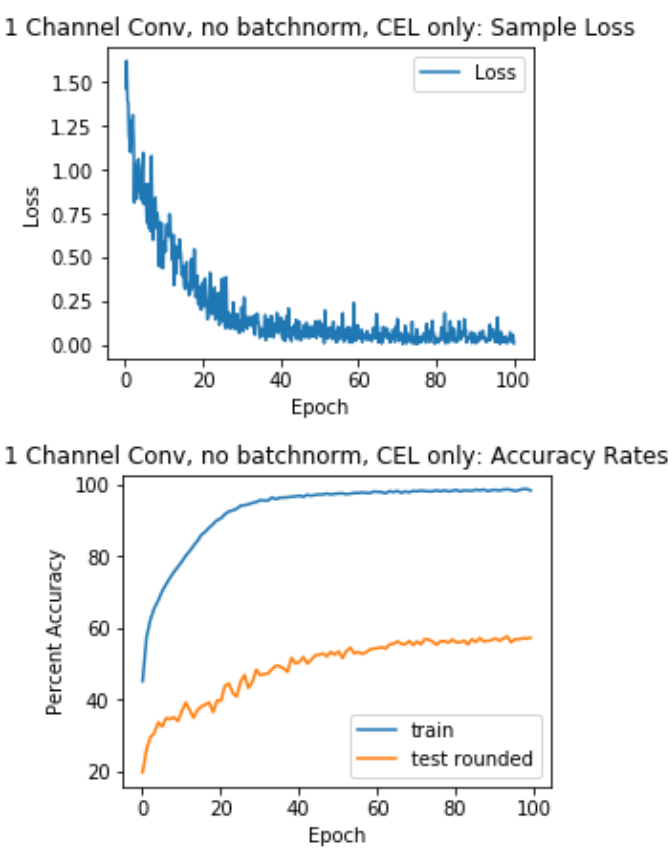


Figure 12

97% by the 42nd epoch. However, the rounded test saturates by epoch 11 with around 46%, and drops by a few percentage points by the time the training accuracy saturates. For a constant of 0.5, see figure 9. Training again caps out around 97%, but takes 45 epochs to get there. The rounded test accuracy has a smoother curve, and reaches the maximum accuracy of around 46 by epoch 17 before dropping a few percentage points. For a constant of 2, see figure 10. The training accuracy hits 97% by epoch 41. This trial had the highest performing test accuracy for these three trials: it peaks around 52% by epoch 9, and then like the others drops a few percentage points. Compare also to an identical trial using standard cross entropy loss, in figure 11. This trial reached the 97% saturation point by epoch 39, unrounded accuracy saturates around 65% by epoch 7, and unrounded accuracy saturates around the same point at 53%. Although most results without batch normalization have not been included in this report, one example is presented in figure 12 for comparison. Note that this trial is for 12 bits only. This trial reaches 97% training accuracy by epoch 43 and achieves 53% rounded test accuracy by epoch 49, but learns more slowly to get there and is not yet saturated, reaching nearly 58% in epoch 94. This is contrary to the expectation of batch normalization will slow down learning. Histograms of the encoder output confirmed however that the encoder output was consistently saturating to 0 and 1 much more slowly with batch normalization.

To address the possibility that the testing accuracy was saturating early, I tried simplifying this model, as well as beginning to record the test accuracy both with and without rounding (trials before this point which show unrounded test results have been rerun). “1 Channel Conv B” in supermodel.py is

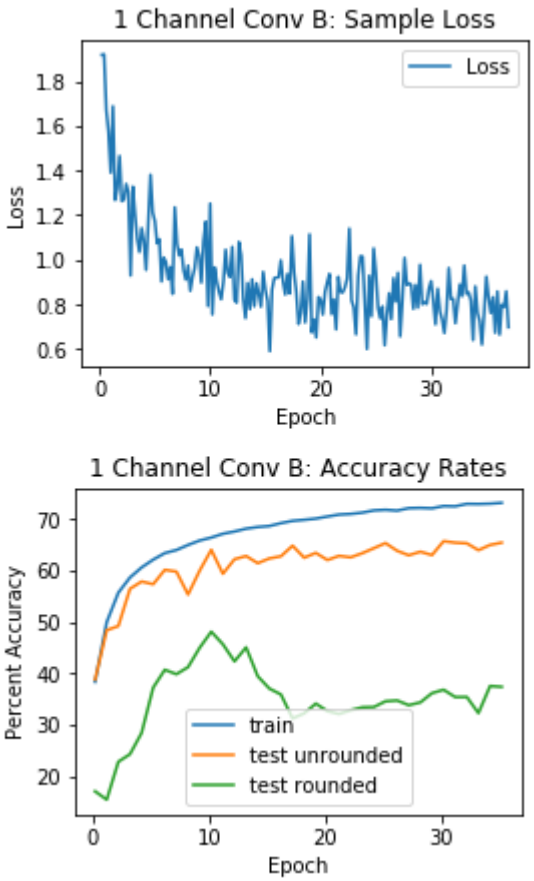


Figure 13

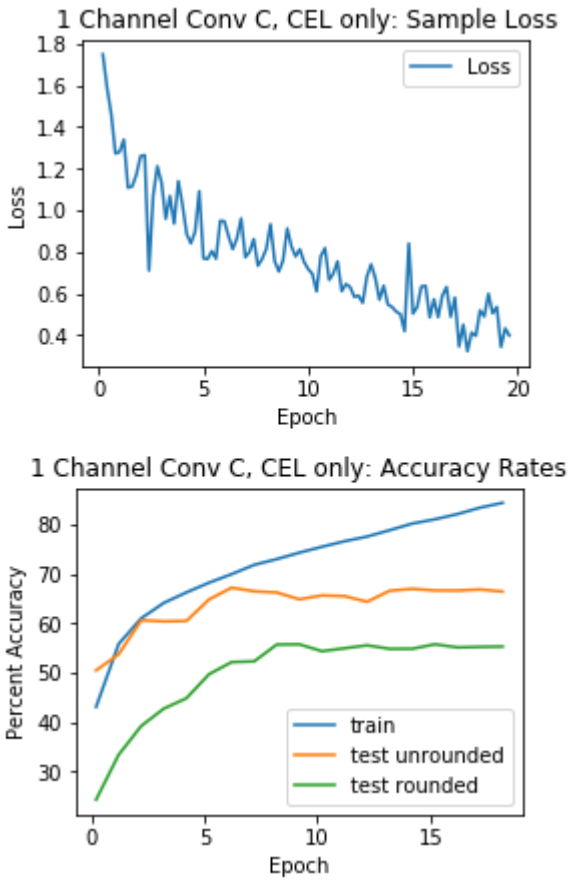


Figure 14

a modification of the previous encoder, in which the three fully connected layers are replaced with a single layer immediately transforming the reshaped convolutional layer output to the encoder output bit size. This layer is followed by a ReLU, which likely had a negative impact on overall performance. See figure 13 for a trial using 8 bits per encoder (24 total) and cross entropy loss only. This time the training accuracy gets up to about 73% by epoch 35 and may still be learning, while the unrounded test accuracy hits around 64% by epoch 11 and hovers above 60% after, and the rounded test accuracy hits 48% on the same epoch 11 but by epoch 21 is hovering around 32%.

An additional modification of this model adds back one of the fully connected layers, with an intermediate hidden size of 84 as used in the original tutorial model. This is “1 Channel Conv C” in supermodel.py. See figure 14 for a trial using 12 bits per encoder and cross entropy loss as the loss function. This has reached around 84% by epoch 20 and is still learning, but the unrounded test accuracy saturates around 67% by epoch 7 and the rounded test accuracy saturates around 55% by epoch 9. See figure 15 for a trial using 4 bits per encoder (12 total) and the custom loss function. By epoch 20 the training accuracy has still managed to reach 80%; by epoch 30 it seems to still be learning, at around 88%. Again though the unrounded accuracy saturates early, hitting 65% by epoch 11, and rounded accuracy saturates around 42% by epoch 15.

The other major approach I took to the combination convolutional and fully connected encoder was to use or adapt a popular high-performing classifier. One which I tried was AlexNet. As with “1 Channel

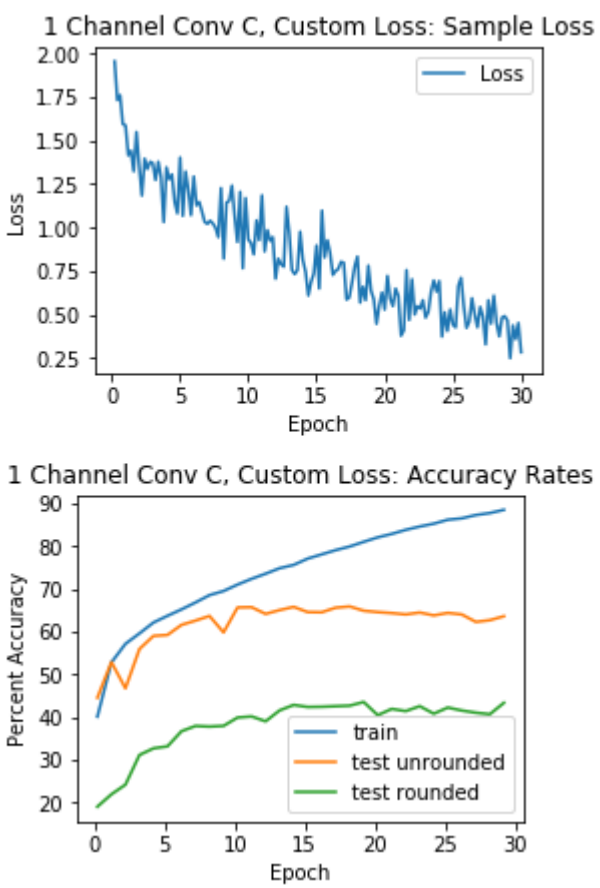


Figure 15

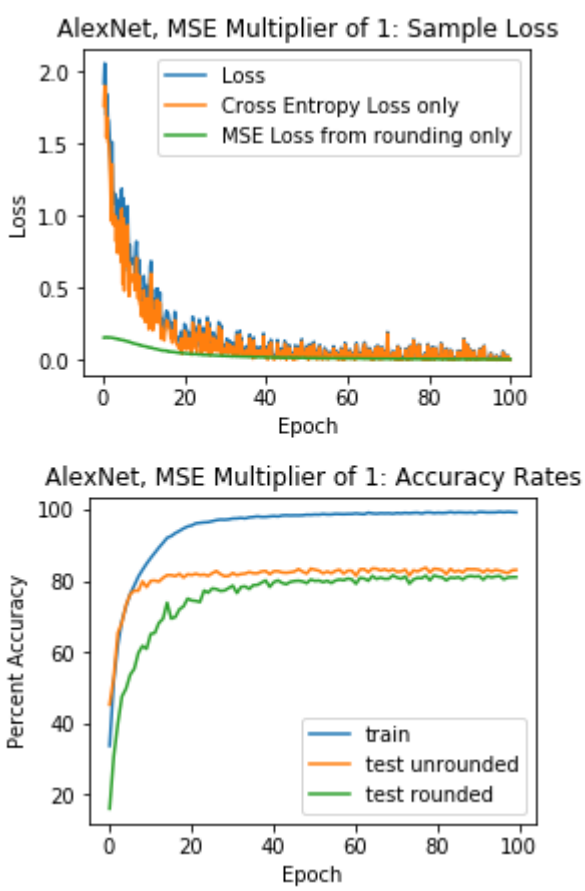


Figure 16

Conv", I took the existing classifier and followed it with a batchnorm and a sigmoid, plus rounding for testing. The Pytorch models library implementation of AlexNet was used. See "AlexNetEncode" in supermodel.py. See figure 16 for a trial using 8 bits per encoder (24 total) with the custom loss function and a multiplier of 1. The training accuracy hits 97% by epoch 27, though it continues to learn after, hitting 99% by epoch 71. The unrounded test accuracy is about the same as the training accuracy until it diverges around epoch 8, when the training accuracy is around 81% but testing accuracy is only 77%. Within ten epochs it gets up to around 82% and stays there the rest of the trial. The rounded test accuracy at the epoch 8 divergence point is about 60%, but by epoch 39 this has nearly reached the unrounded test accuracy, at about 80%, and sometimes hitting 81%. For cross entropy loss only, see figure 17. Training accuracy hits 97% by epoch 27 here too, and is still learning. Unrounded test accuracy diverges from training accuracy around epoch 10, when the training accuracy is nearly 82% but the test accuracy is only 76%. By epoch 26 this gets up to 82%. The rounded test accuracy is about 63% at epoch 10, and gets to 75% by epoch 25. At 30 epochs, where this trial ended, the unrounded test accuracy is at 82.73% and the rounded test accuracy is at 74.81%; at that same point with the custom loss function, these rates are at 81.38% and 77.96% respectively.

I attempted a customization of the AlexNet classifier as well, to see if simplifying it would resolve the early saturation problem. I reduced the Pytorch implementation of AlexNet to have only the first two convolutional layers and their associated ReLUs and MaxPools, and then removed the second to last fully connected layer and its ReLU. As before, this is followed by batchnorm and sigmoid, plus rounding

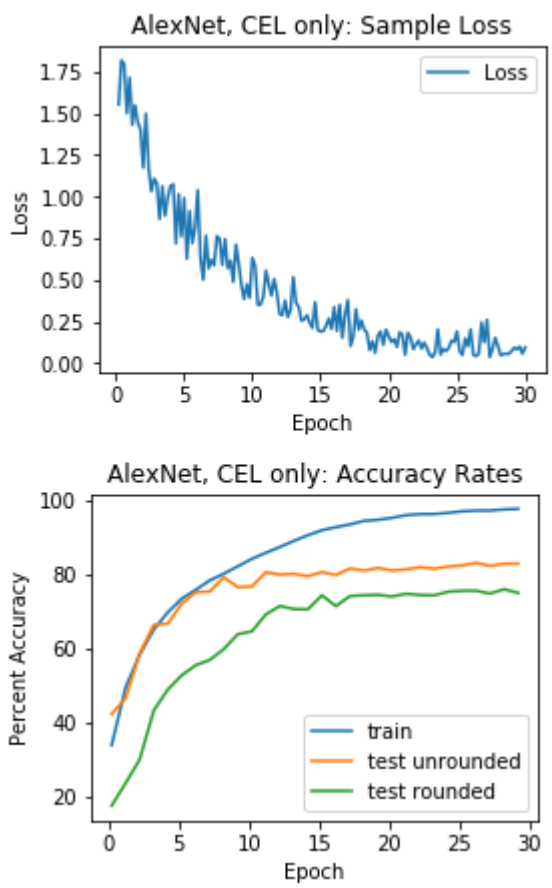


Figure 17

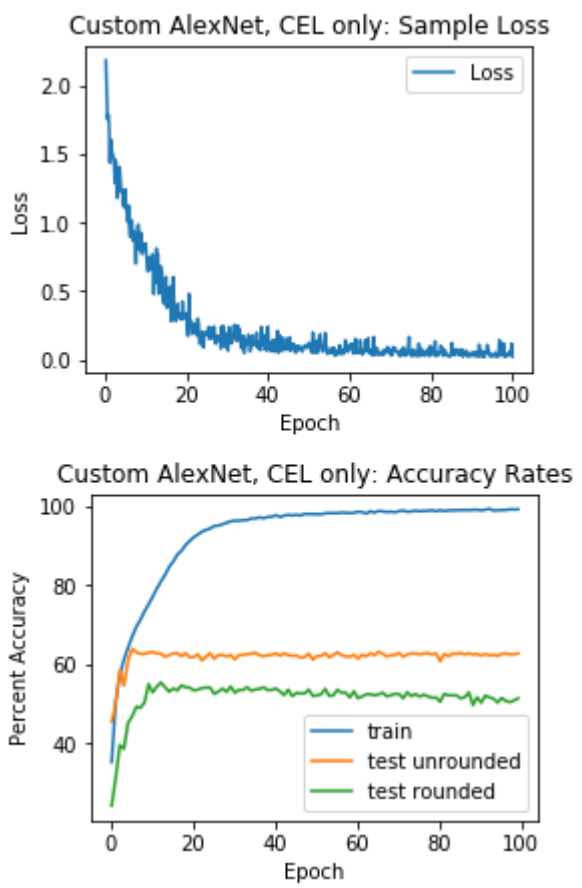


Figure 18

on rounded tests. These changes meant that it was no longer necessary to resize the images to 224x224 before processing. This is implemented as "CustomAlexNetEncode" in supermodel.py. See figure 18 for a trial using 16 bits per encoder (48 total) and cross entropy loss only. The training accuracy hits 97% by epoch 30 and 99% by epoch 92. The unrounded test accuracy diverges at epoch 6, when it nears 64% and generally stays there, settling to a bit above 62%. The rounded test accuracy hits 54% at epoch 10 and settles a bit more, down to around 51%.

A similar attempt was made with ResNet. I used the ResNet 18 implementation in the Pytorch models. This was followed by a batchnorm, sigmoid, and sometimes rounding, as before. This is "ResNetEncode" in supermodel.py. See figure 19 for a test using 8 bits per encoder (24 total) with cross entropy loss only. Due to the slowness of ResNet this trial was not able to run as long as desirable, but by epoch 10 the training accuracy has reached almost 94% and is still rising. Unrounded test accuracy hits 80% by epoch 8 and hovers there the remaining couple epochs of the trial, and rounded test accuracy is still growing at epoch 10 with 68% accuracy. See figure 20 for the same trial using the custom loss function with a multiplier of 1. At epoch 10, the training accuracy is about 93%, the unrounded test accuracy is about 78%, and rounded test accuracy is about 65%. The training accuracy keeps growing through the end of the trial, to 98% at epoch 20. The unrounded test accuracy hits its peak of around 82% in epoch 11 and stays around there the rest of the trial. The rounded test accuracy peaks around 70% in epoch 14 and stays around there the rest of the trial. These trials took 35 minutes to an hour per epoch using GPU with Google Colaboratory, and thus regularly ran into the twelve-hour timeout this

service applies. It is possible that the cross entropy loss function may not experience the same saturation that is experienced using the custom loss function, and unfortunate that the trial ended before a clear saturation point was reached.

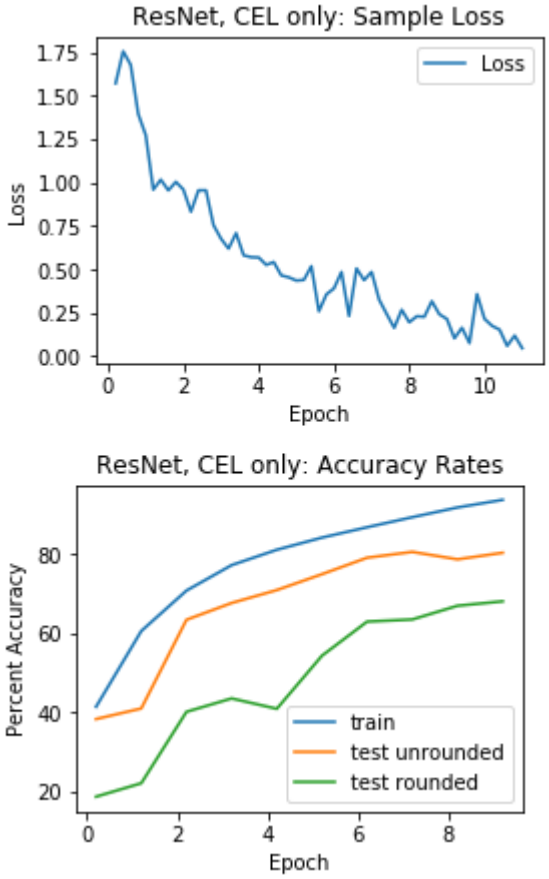


Figure 19

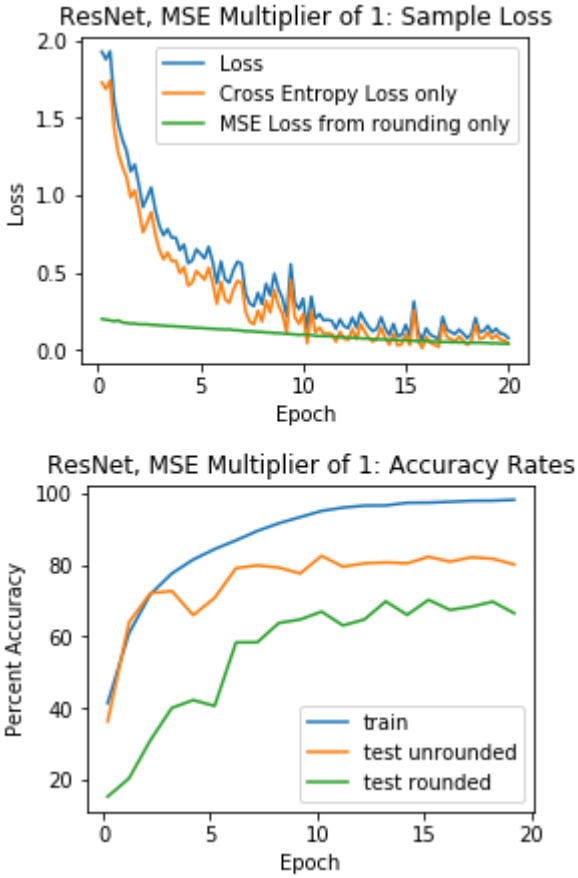


Figure 20

Remaining efforts used a fully convolutional approach. For these, the bit size for the encoder output is a square number such as 4, 9, 16, etc. Note that in the code, the bit size specified is used as the square root of this number: for example if bit size 4 is specified, the output for the encoders that follow will be 16 bits per encoder.

The first approach used the tutorial convolutional classifier mentioned previously, minus the fully connected layers and with modified convolutional layers. The first layer still outputs six channels, but the second layer scales this back to three and has a filter size of 3x3. The final convolutional layer scales back to one channel, and uses a filter whose sides are seven minus the square root of the encoded bit size. For example if each encoder is to output 16 bits, seven minus the square root of 16 is three, so the filter will be 3x3. This has an obvious limitation of 6x6 on the output bit size, at which point the final convolutional layer will have a 1x1 filter and may not work particularly well; most likely this design will have optimal performance with encoder output up to 5x5. As usual this configuration was followed by batch norm, sigmoid, and rounding on test. This encoder is “1 Channel Conv Square” in supermodel.py. Decoding this for a more flexible bit size proved complicated, and so I attempted some decoders for specific bit sizes rather than taking a generalized approach in order to see what might work. The first

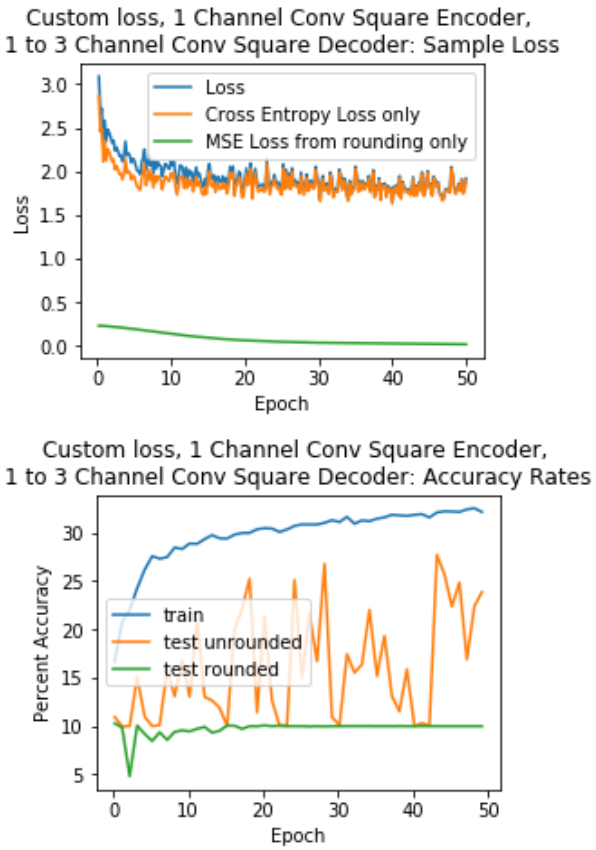


Figure 21

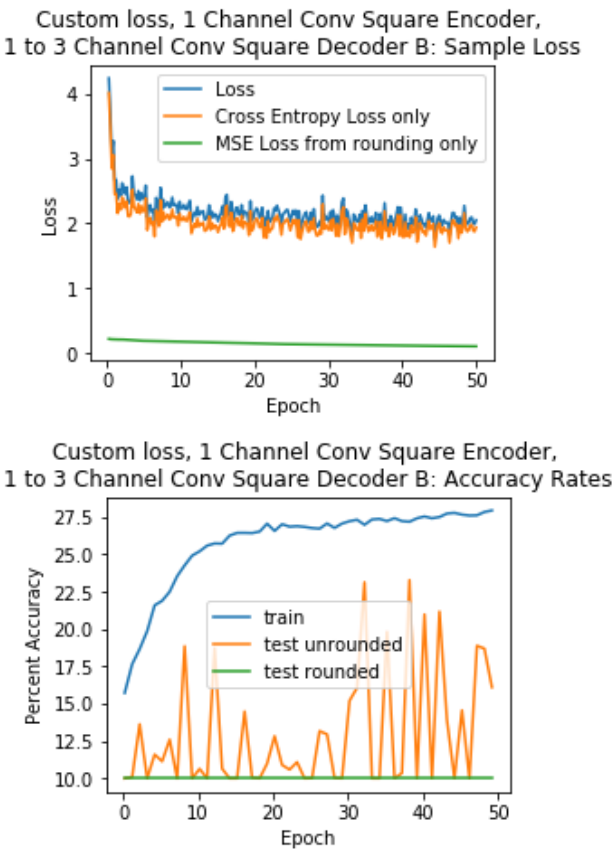


Figure 22

fully convolutional decoder paired with this is “1 to 3 Channel Conv Square” in supermodel.py, and is explicitly written for 3x4x4 input (4x4 output from this encoder, recombined as RGB). This decoder first upsamples with a scale factor of 2 in bilinear mode, bringing the image up to 3x8x8. This is followed by a convolutional layer with a 3x3 filter, padding of 2, and 6-channel output (output is now 6x10x10), followed by a ReLU. This upsampling, convolution, and ReLU pattern is repeated twice. The second upsampling layer is the same as the first, with output of 6x20x20. The second convolutional layer has a 5x5 filter, padding of 3, and 12-channel output, for output size of 12x22x22. The third upsampling layer is the same as the others, with output of 12x44x44. The third convolutional layer has 6 channel output and a 7x7 filter with no padding, for an output dimension of 6x38x38. This is followed by a final convolutional layer and ReLU, without upsampling first. This final convolutional layer outputs 3 channels, and has a 7x7 filter, for an output size of 3x32x32 to match the original size of the images. See figure 21 for an example using 16 bits per encoder (48 bits total) and the custom loss function with a multiplier of 1. By epoch 44 the training accuracy hits 32%, and it’s unclear if there is learning after that point as of epoch 50 or if it has saturated. The unrounded accuracy is all over the place between 10% and a maximum of nearly 28% at epoch 44. The rounded accuracy generally stays around 10%, or random choice, with some dips below that in the first few epochs.

This decoder seemed like it might be learning mainly from the edges created by the upscaling, and the early saturation on the training accuracy might mean that the model was too complicated, so I made an alternate approach. See “1 to 3 Channel Conv Square B” in supermodel.py. As before, this is designed specifically for 3x4x4 input. This simply upscales once with a scale factor of 8 and bilinear mode to take

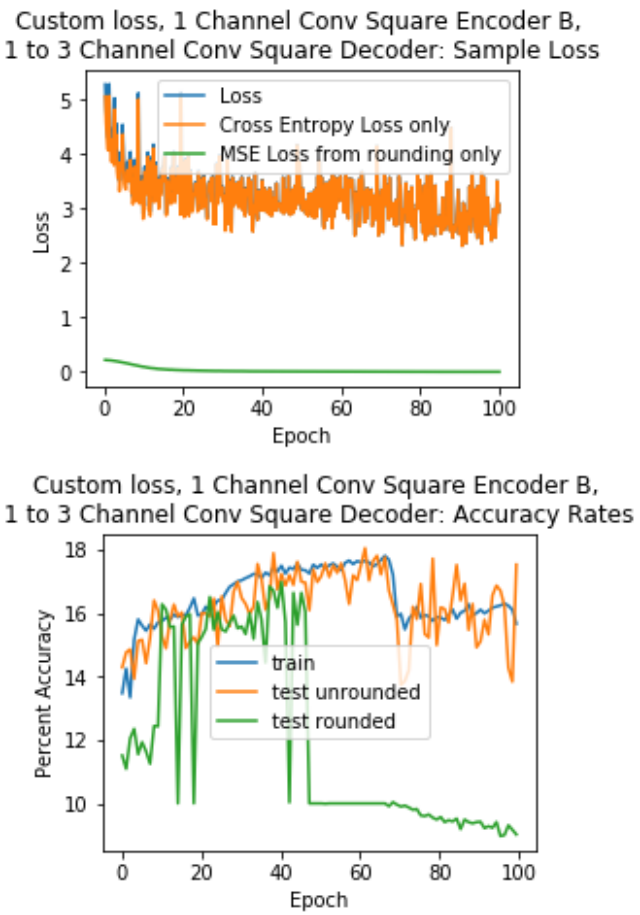


Figure 23

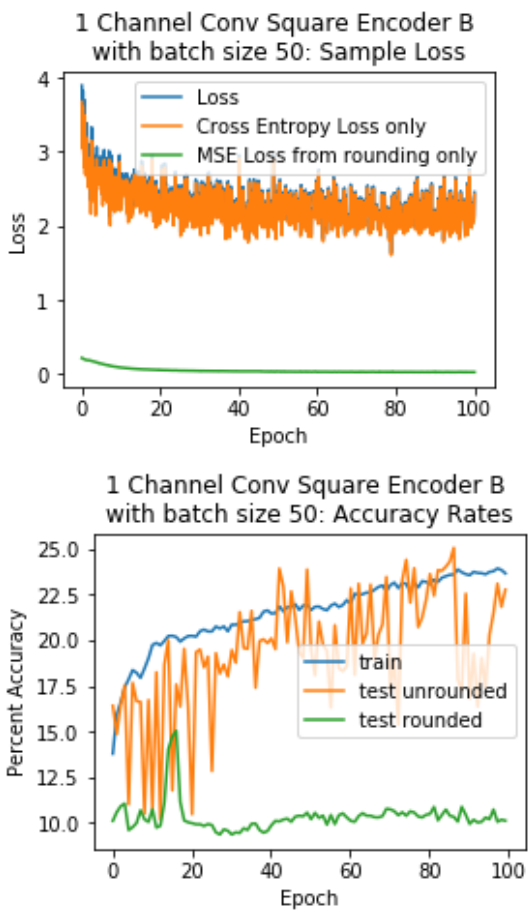


Figure 24

the image to 3x32x32, and then applies a convolutional layer which maintains these dimensions, with three channel output, a 5x5 filter, and padding of 2. Although this still has padding, my hope was that this would minimize the effect of learning from that padding. This succeeded in slowing down learning and delaying saturation, but otherwise performed even worse. See figure 22 for a trial with 16 bits output for each encoder using the custom loss function with a multiplier of 1. The training accuracy slows down around epoch 11 at 25%, and by epoch 50 has edged up only to just under 28%. Unrounded test accuracy remains scattered between random choice at 10% and a maximum below the training accuracy; this time the maximum is around 23%, at epoch 39. The rounded test accuracy is a flat line at 10%.

I also tried a different version of the encoder along with the initial better-performing decoder. This is nearly the same as the initial fully convolutional encoder, but the second convolutional layer scales up to 16 channels rather than dropping to 3. This is implemented as “1 Channel Conv Square B” in supermodel.py. See figure 23 for a trial with 16 bits output for each encoder using the custom loss function with a multiplier of 1. This was allowed to run for longer, and shows that there may also be an issue with a local minimum, as the training accuracy has a significant drop around epoch 70. This is also interesting because the unrounded test accuracy, while more scattered than the training accuracy, tends to be pretty close. Before epoch 50, the rounded test accuracy is also frequently nearby. However, all of these accuracies are below 20% and so have questionable significance. The peak training accuracy

is nearly 18% around epoch 67, at which point the unrounded test accuracy is around the same value and the rounded test accuracy is 10%. The peak rounded test accuracy is nearly 17%, at epoch 41.

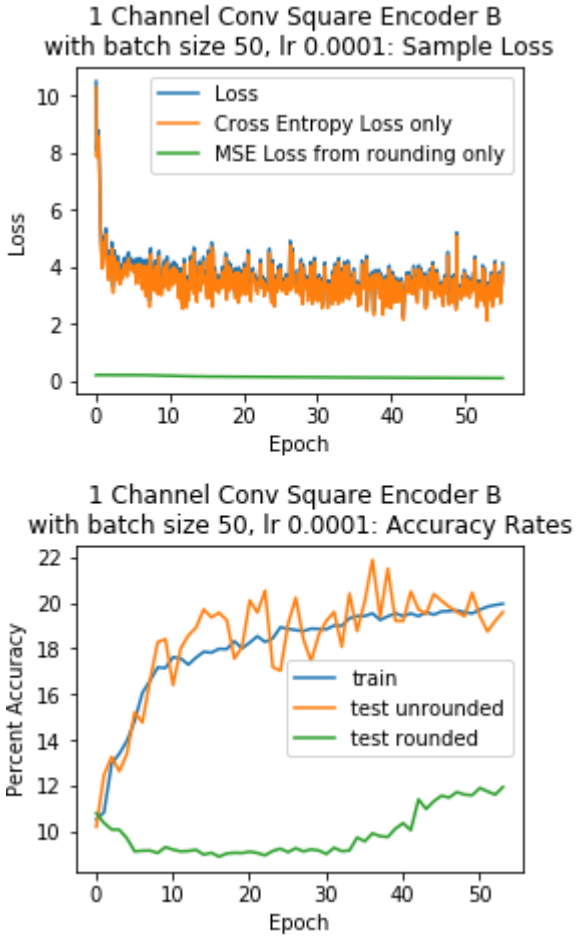


Figure 25

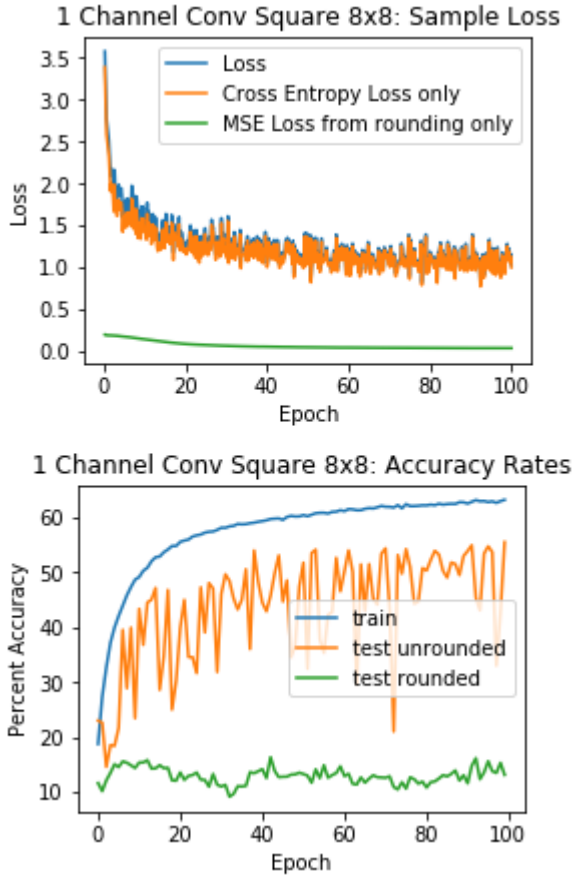


Figure 26

As the performance for this configuration was poor but interesting, I tried varying a couple hyperparameters. One thing was to adjust the batch size from 100 (used in all other trials presented in this report) down to 50. See figure 24. This produced better training accuracy, which gets to 23% by epoch 71 and may be slowly learning all the way through to epoch 100, when it has an accuracy of 23.676%. The unrounded test accuracy is still very scattered but the higher values are somewhat closer to the training accuracy, and the overall curve is similar to the training accuracy, reaching 22.78% at epoch 100. The rounded accuracy is no longer a flat line at 10%, but it doesn't appear to be learning, other than a small spike to 15% around epoch 16. At epoch 100 this accuracy is 10.11%. I also tried using this same batch size of 50 and dropping the learning rate from 0.001 to 0.0001. See figure 25. This trial ended while the rounded test accuracy was increasing, so it is unclear if that is a useful increase or simply an odd spike like that in the previous trial. However, it is increasing from somewhere below 10% to somewhere above, and this may not have any significance anyway. At epoch 54 where this trial ended, the training accuracy is 19.964% and appears to still be learning; the unrounded test accuracy is 19.6% and generally scattered near the training accuracy; and the rounded test accuracy is 11.93%.

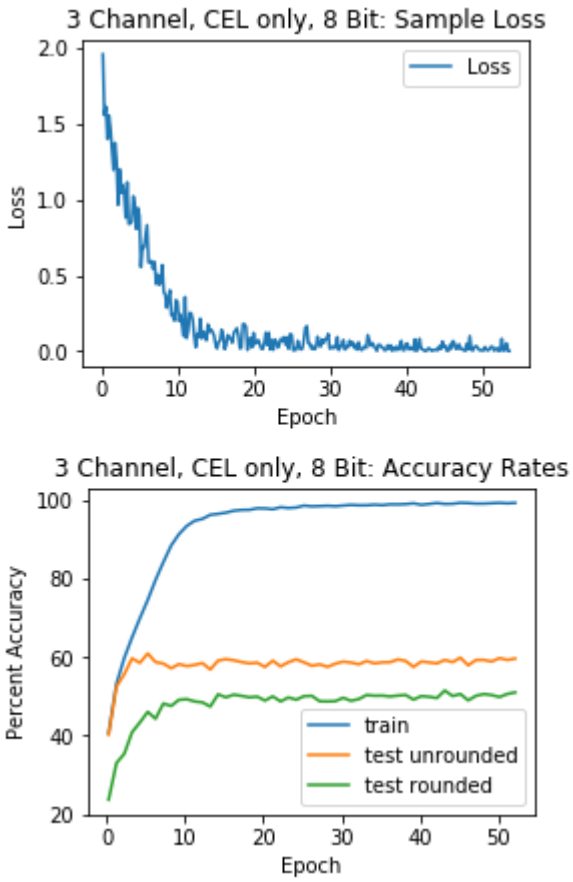


Figure 27

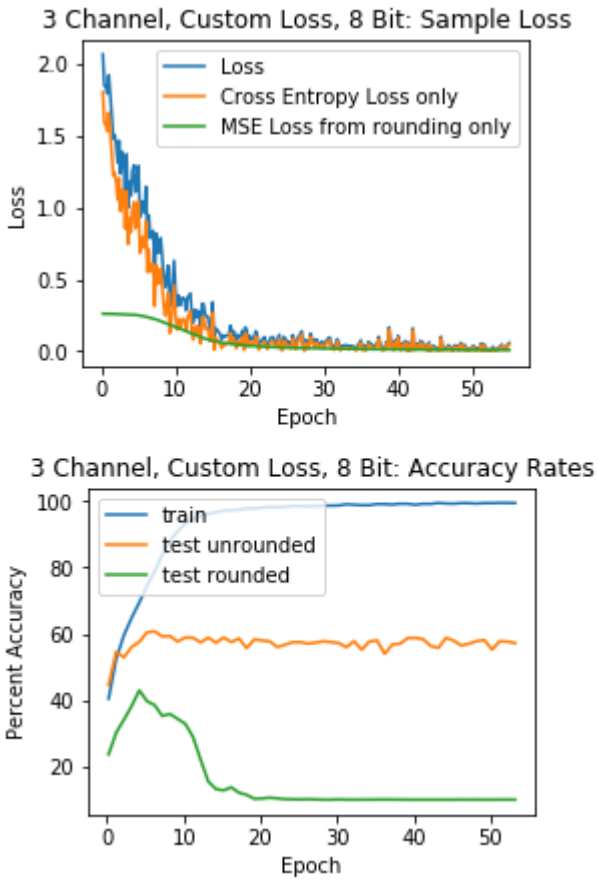


Figure 28

I also wanted to check the possible impact of the low bit size, so an additional effort is the encoder "1 Channel Conv Square 8x8" and decoder "1 to 3 Channel Conv Square 8x8" in supermodel.py. This encoder is largely similar to the second fully convolutional encoder. However, the second convolutional layer has a filter size of 5, and is not followed by a maxpool layer. This allows the final convolutional layer to use a 3x3 filter to create 8x8 output. The decoder was able to take advantage of the larger input size: it is implemented as a fusion of the other two. First it upsamples with a scale factor of 4, which brings it immediately to the 3x32x32 size. Then it uses three convolutional layers each followed by ReLUs. These layers follow a similar structure to the template classifier, going up to 6 channels, then 16, then back down to 3 rather than 1 as this will be the input for the classifer. These each use 5x5 filters like the classifier template, and use padding of 2 to maintain the size. See figure 26 for this test; for 64 bits per encoder this is a total of 192 bits. The custom loss function is used with a multiplier of 1, and I returned to the usual batch size and learning rate of 100 and 0.001 respectively. This overall achieves higher accuracy than the other fully convolutional attempts, but the performance is still not great. At epoch 100 the training accuracy is at 63.066% and may still be learning, although very slowly. The unrounded test accuracy is spiky as before, and shows a similar curve to the training accuracy, but has a noticeable gap of around 6-10% poorer performance than training at its peak. On the 100th epoch the unrounded test accuracy is 55.38%. The rounded test accuracy moves up and down in the 10-20% range, without any apparent ongoing learning. At epoch 100 the rounded test accuracy is 13.09%.

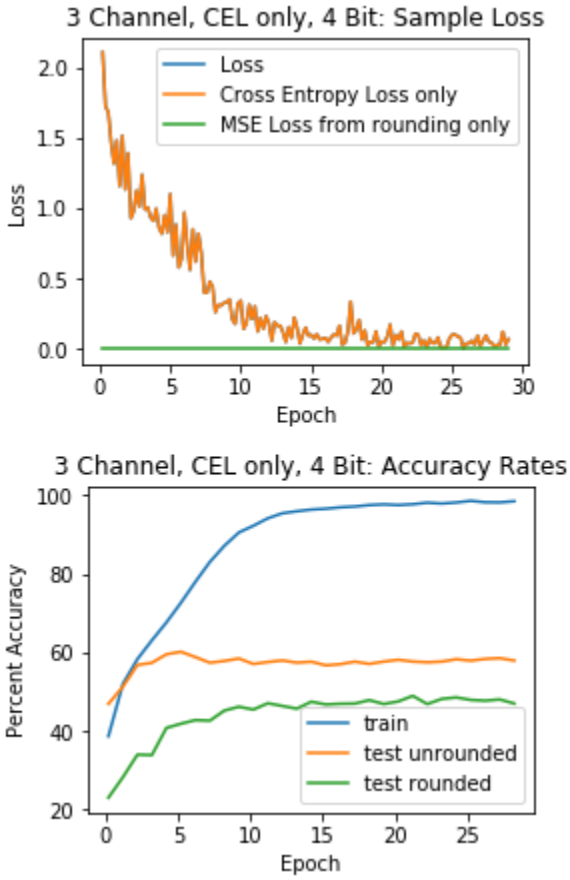


Figure 29

I also wanted to consider the possibility that separating by color layer might cause unexpected problems in learning, and explored the option of instead maintaining the color structure and separating the image itself into quadrants vertically and horizontally. This is performed with "3 Channel Small Conv" encoder and "3 Channel" decoder in supermodel.py, which are modifications of the combination convolution and fully connected approach. Images passed in to this encoder are already fairly small, so there are no maxpool layers. The encoder begins with two convolutional layers, each with 3x3 filters and padding of size 1, preserving their size. The first layer outputs 6 channels and the second 16. Each layer is followed by a ReLU. The image is then flattened and passed through three linear layers with the same hidden sizes as used in the tutorial: first 120, then 84, and the final output is the encoded size. The first two fully connected layers are followed by ReLUs. As usual the process ends with batchnorm and sigmoid, with rounding for the rounded test. The decoder is a fully connected layer followed by reshaping, as before, but designed for the different structural requirements. Note that although the encoder is designed to work for any size and number of subsections of the original image, the decoder was hardcoded for dividing the image into four parts. It can however be easily modified to take any number of parts using the total_subsets function in tools.py.

Tests of this encoder/decoder combo are as follows. Figure 27 shows use of cross entropy loss for 8 bits per encoder (32 bits total). Training accuracy reaches 98% by epoch 23, and 99% by epoch 39. The unrounded test accuracy starts about the same as the training accuracy until epoch 4, when it gets

about 59% accuracy and stays there for the remainder of the trial. The rounded test accuracy learns for a bit longer; it is just under 41% at epoch 4, but edges up to 50% by epoch 15, and then stays there. Figure 28 is the same test using the custom loss function with a multiplier of 1. Training performance is similar, reaching 98% by epoch 21, and unrounded test accuracy is similar as well. However, the rounded test accuracy peaks at just under 43% at epoch 5 and then drops back down to random choice by epoch 20. Figure 29 returns to cross entropy loss, but explores the possibility of 4 bits per encoder (16 total). Like the 8 bit test, this reaches 98% accuracy by epoch 23. At epoch 4 the unrounded test accuracy is at 57%; it reaches 60% by epoch 6, and hovers between the two for the remainder of the trial. Rounded test accuracy learns more slowly at this bit size, reaching just under 34% by epoch 4; it peaks with nearly 49% by epoch 22, and hovers between 46% and that peak from epoch 11 onward.

Results are summarized in table 1 below. This table only summarizes the results described above. Numerous trials beyond those listed here were also attempted which seemed redundant; for example, a trial of the last encoder/decoder combo using the custom loss function with 4 bits, which performed far worse than its 8-bit counterpart.

Table 1: Highlighted results summary

Figure	Encoder	Decoder	Loss (Multiplier)	Dataset	Bits per Encoder	# Encoders	# Epochs for listed results	% Training accuracy	% Unrounded test accuracy	% Rounded test accuracy
1	Simple 1D 200 neurons	Simple 1D	CEL	MNIST	4	4	10	98.775	97.47	93.66
2	Simple ReSigmoid 200 neurons	Simple 1D	CEL	MNIST	4	4	10	98.6	97.77	93.35
3	Simple Quantize 200 neurons	Simple 1D	CEL	MNIST	4	4	10	97.46	n/a	96.56
4	Simple 1D 200 neurons	Simple 1D	Custom (1)	MNIST	4	4	10	98.83	97.81	94.24
5	Simple 1D 200 neurons	Simple 1D	CEL	CIFAR10	8	3	33	85.668	50.31	33.6
6	Simple 1D 200 neurons	Simple 1D	Custom (1)	CIFAR10	8	3	33	85.746	50.24	33.4
7	Simple Quantize 200 neurons	Simple 1D	CEL	CIFAR10	8	3	50	79.738	n/a	45.44
8	1 Channel Conv	1 to 3 Channel Conv	Custom (1)	CIFAR10	12	3	42	97.03	Not recorded	42.4
9	1 Channel Conv	1 to 3 Channel Conv	Custom (0.5)	CIFAR10	12	3	45	97.134	Not recorded	41.79
10	1 Channel Conv	1 to 3 Channel Conv	Custom (2)	CIFAR10	12	3	41	97.112	Not recorded	49.84
11	1 Channel Conv	1 to 3 Channel Conv	CEL	CIFAR10	12	3	39	97.114	63.87	53.84
12	1 Channel Conv (without batch norm)	1 to 3 Channel Conv	CEL	CIFAR10	12	3	100	98.446	Not recorded	57.27

13	1 Channel Conv B	1 to 3 Channel Conv	Custom (1)	CIFAR10	8	3	35	73.008	64.97	37.51
14	1 Channel Conv C	1 to 3 Channel Conv	CEL	CIFAR10	12	3	20	84.352	66.4	55.27
15	1 Channel Conv C	1 to 3 Channel Conv	Custom (1)	CIFAR10	4	3	30	88.452	63.54	43.27
16	AlexNetEncode	1 to 3 Channel Conv	Custom (1)	CIFAR10	8	3	100	99.222	83.07	81.04
17	AlexNetEncode	1 to 3 Channel Conv	CEL	CIFAR10	8	3	30	97.506	82.73	74.81
18	Custom AlexNetEncode	1 to 3 Channel Conv	CEL	CIFAR10	16	3	100	99.116	62.73	51.45
19	ResNetEncode	1 to 3 Channel Conv	CEL	CIFAR10	8	3	10	93.8	80.4	68.11
20	ResNetEncode	1 to 3 Channel Conv	Custom (1)	CIFAR10	8	3	20	98.254	80.18	66.57
21	1 Channel Conv Square	1 to 3 Channel Conv Square	Custom (1)	CIFAR10	16	3	50	32.154	23.86	10.0
22	1 Channel Conv Square	1 to 3 Channel Conv Square B	Custom (1)	CIFAR10	16	3	50	27.94	16.1	10.0
23	1 Channel Conv Square B	1 to 3 Channel Conv Square	Custom (1)	CIFAR10	16	3	41	17.498	17.23	16.98
24	1 Channel Conv Square B, batch size 50	1 to 3 Channel Conv Square	Custom (1)	CIFAR10	16	3	100	23.676	22.78	10.11
25	1 Channel Conv Square B, batch size 50, learning rate 0.0001	1 to 3 Channel Conv Square	Custom (1)	CIFAR10	16	3	100	19.964	19.6	11.93
26	1 Channel Conv Square 8x8	1 to 3 Channel Conv Square 8x8	Custom (1)	CIFAR10	64	3	100	63.066	55.38	13.09
27	3 Channel Conv	3 Channel	CEL	CIFAR10	8	3	53	99.316	59.61	50.99
28	3 Channel Conv	3 Channel	Custom (1)	CIFAR10	8	3	5	69.43	57.55	42.94
29	3 Channel Conv	3 Channel	CEL	CIFAR10	4	3	29	98.502	57.92	46.98

Analysis and Conclusions

Most trials in which unrounded test accuracy was measured showed this performance to saturate early, well before the training accuracy. This behavior was unexpected and could not be resolved. It was observed with both the MNIST and CIFAR10 datasets. It was also very difficult to narrow the gap between the unrounded and rounded test accuracy. The best CIFAR10 performance for test accuracy came from the use of AlexNet as an encoder, and the gap between rounded and unrounded test accuracy was effectively minimized using the custom loss function. A longer test could determine if the same minimization could be achieved using cross entropy loss only. ResNet showed strong performance as well, but it is unclear if over time it would perform comparably to AlexNet. It was definitely much slower. Slow performance of both of these models set a limit on the extent of testing which could be performed due to the limitations of Google Colaboratory.

The various customizations of classifiers I implemented as encoders generally performed worse than simply using the original classifiers which inspired them. Combination convolutional and fully connected encoders were able to achieve high training accuracy but the test accuracy usually saturated by the tenth epoch, well before saturation of the training accuracy. Separating the image into quarters with preserved color sometimes performed comparably to separating by color layer when using cross entropy loss, but use of the custom loss function in the trials attempted caused the rounded test accuracy to briefly learn and then unlearn everything. Although the customizations to AlexNet did not improve accuracy, they did significantly speed up performance and reduce overhead without a significant loss to accuracy, so it is possible there are other modifications which can find a compromise.

Histograms were used to observe the encoder output. Batch normalization was successful at slowing down the process of encoder output saturating to 0 or 1, but did not appear to address the issue of test accuracy saturation, and in fact may have increased some of the saturation problem.

The MNIST dataset proved difficult to test with because of its simplicity. Although this made the dataset poor for purposes of this research, it means that for some scenarios, if the data is simple enough, an extremely simple neural network can be quickly and easily trained to quantize pieces of the data for transmission with negligible loss of classification accuracy. The benefits of the custom loss function were marginal. The accuracy achieved for this very simple dataset from the custom quantize activation function was higher than that achieved with the custom loss function, and nearly that of the training accuracy; perhaps there may be a better way to implement a similar custom loss function to achieve the same performance. This high performance with the custom activation function was not observed with the fully connected approach to CIFAR10, and was not attempted as part of the combination convolutional and fully connected approach.

For the custom loss function, the use of a multiplier other than 1 did not provide a clear benefit. When comparing different multiplier behavior as well as cross entropy loss alone (1 Channel Conv tests), rounded test accuracy was slightly better with a multiplier of 1 than 05. It achieved a higher accuracy with a multiplier of 2, but saturates more suddenly. Cross entropy loss produced the highest rounded test performance for these trials. However, these trials did not all include recording both rounded and unrounded test accuracy; in the AlexNet trials the custom loss function brought those two closer together, as well as providing slightly better performance.

Better selection of a starting seed, as well as cross validation, could address some of the issues experienced in these trials. All were performed with a manually set starting seed of one. Some non-

systematic trials were performed with arbitrarily chosen seeds, without significant change to results, and so were not included.

References

[1] A. P, "Handwritten digit recognition using PyTorch," Medium, 21 May 2018. [Online]. Available: https://medium.com/@athul929/hand-written-digit-classifier-in-pytorch-42a53e92b63e.

[2] A. Chen, "Pytorch Playground," github, [Online]. Available: https://github.com/aaron-xichen/pytorch-playground.

[3] "Training a classifier," Pytorch, 2017. [Online]. Available: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html.