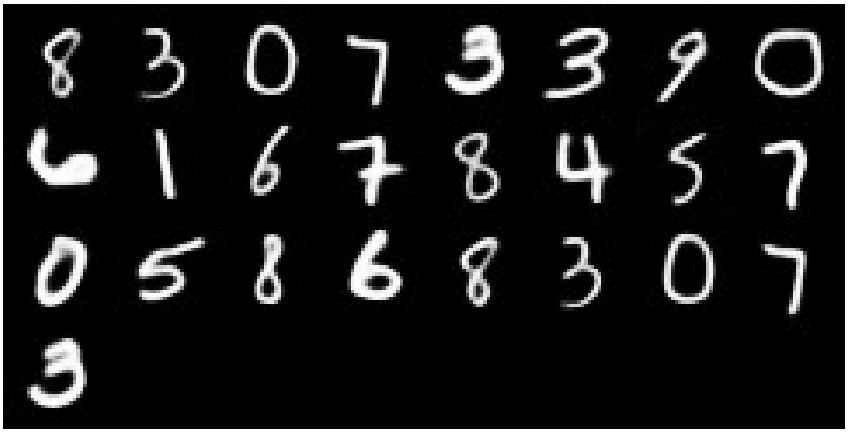
Abstract

As we learned in class, one of the main ideas in deep unsupervised learning is the autoencoder. Deepmind's recent (March 2017) paper, Generative Temporal Models with Memory (GTMM) (https://arxiv.org/pdf/1702.04649.pdf), presents a new machine learning algorithm for learning long-term dependencies in sequence data. Their model is a new type of Variational Autoencoder, designed for representing sequences.

In this project, I examine one of the seemingly hand-designed components of this algorithm and its architecture, and try to replace them with the regularization techniques which we learned in this class. My goal was to better understand the sparsity dynamics involved in training this model, and potentially improve the training process. Like many algorithms which use a fixed-size memory, the GTMM accesses its memory by taking a weighted average over it. Unlike most, however, it generates the weights using the softplus function (followed by normalizing them), rather than a softmax. In the paper, the authors explain that training takes longer to converge when a softmax is used.

Since this model does not have a public implementation, the first half of the project was simply replicating the results shown. I focus on the easiest task, "perfect recall", in which the sequences studied consist of MNIST digits, with the last five items equal (pixel-wise) to the first five. In the second half, I explore the effects of sparsity-enforcing regularization for the read heads. Since the desired behavior of each read head is to select exactly one location from memory, it makes sense that sparsity effects would be quite important. In the end, while my changes did help me to improve my understanding of the sparsity dynamics, none of them actually improved the training process very much.

Sample sequence for the Perfect Recall task. The last 5 digits are identical to the first five.



Terminology

Let us first elaborate on what is meant by "read heads". In the GTMM model, the autoencoder generates a prediction at each time step for the next frame, based on the previous frames it has seen. Transformed versions of the previous frames are stored in a memory of size equal to the length of the sequence. Thus, to generate this prediction, it relies on an LSTM controller, which must generate a linear combination of memory elements. The controller does

this by first generating R probability distributions $\mathbf{w_r}$ over the memory spaces, each of which is called a "read head". (In this paper, we always use $R = 5$.) The controller then chooses how much emphasis to put on each read head at every point in time, by generating R gate parameters $\mathbf{g_r}$ between 0 and 1. In this project, however, I focused on the weights on the read heads. Note that for each read head, its weights must sum to 1, since (as was just stated) they form a probability distribution.

Initial Observations and Hypothesis

After implementing the model, I examined the weights of each of the read heads, and noted how they evolved as training progressed. I observed that they stay roughly equal to each other for a long time, as the autoencoder learns the basics of how to encode and decode MNIST images. After that, gradually become more and more sparse. By the end of training, the controller has figured out the structure of the input sequences (namely that the first 5 images and the last 5 are identical). When this happens, and it needs to make a prediction which is already in memory, it outputs gate parameters which effectively zero out all but one of the read heads; and that remaining read head is a one-hot encoding of the right spot in memory.

Below are some examples of what happens before then, when the weights aren't sparse enough yet. In the early stages of training, 5 or 6 of the positions in $\mathbf{w_r}$ are relatively large (i.e., above 0.1).

Sample Predictions as Training Progresses

From Iteration 1620

From Iteration 2070

From Iteration 2510

From Iteration 3270

Thus, there is an interesting tension here: in the end, the weights will be either irrelevant or extremely sparse; but in the earlier stages of training, it is important not to discard possible memory positions too quickly. This explains why softmax might not work as well as a normalized softplus: for producing weights: the softmax of a zero-centered real-valued vector tends to be closer to a sparse vector than a normalized softplus. The softmax is a normalized exponential, so if one term is much bigger than the others, it completely dominates; whereas in the softplus case, a large value dilutes the others only linearly.

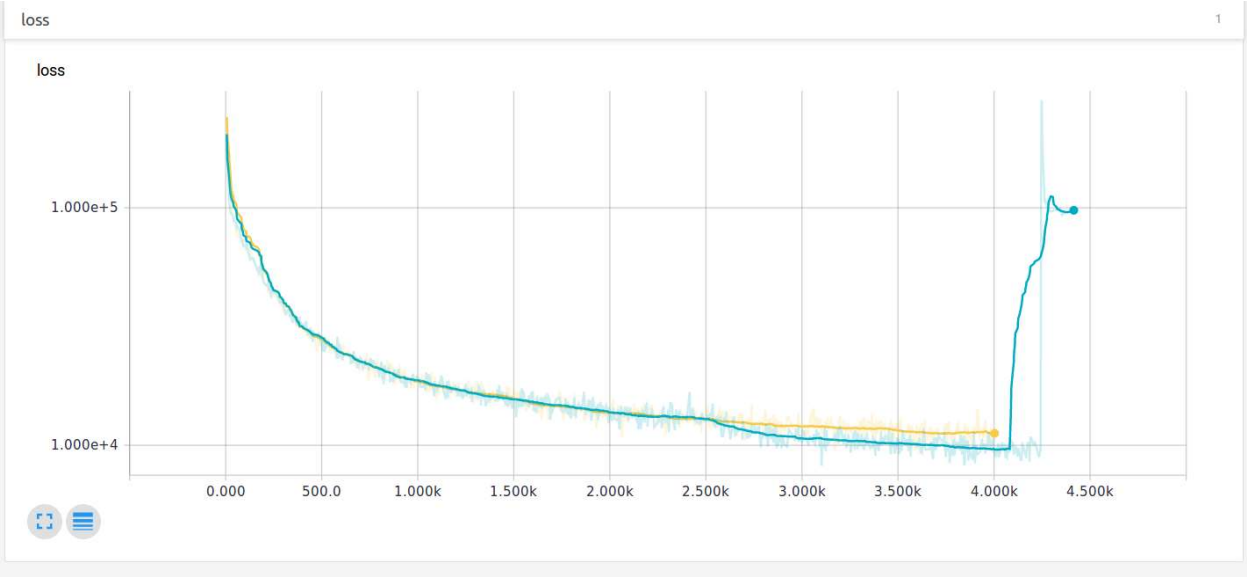
Experiments

To confirm this idea, I tried adding a regularization term to the loss function which encouraged sparsity. I chose to use the l-0.5 norm of the read head weights, i.e. the sum of the square roots of the elements of w_r. (Clearly this is obviously not a convex loss, but since we're dealing with a deep neural network, that doesn't concern us too much, as the problem wasn't convex to begin with.)

With this added sparsity regularization, the training process went worse. Of the 5 read heads, 4 became sparse but useless - for example, one of them was a one-hot encoding of a location guaranteed to be 0. (This behavior is appropriate during the beginning of training, but not later on.) Those 4 read heads were ignored by the controller, leaving it to focus on just one, which was not very sparse. In general, zeroing out memory slots wasn't helpful, since it happened too early.

Note that the loss itself didn't change very much, as shown below:

Yellow is the modified version, Cyan is the original. (Please ignore the sudden spike at the end of the cyan.)



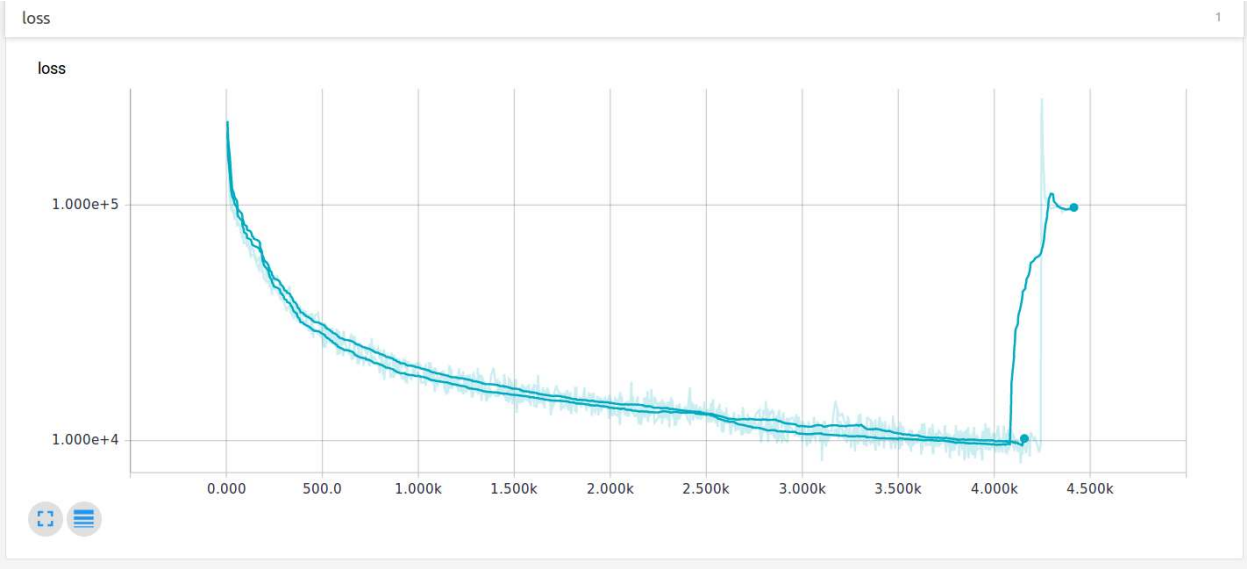
However, the quality of the predictions dropped significantly. For example:

After 3151 iterations, the Cyan (original) was generating almost-perfect images, while the Yellow (modified) still struggled. This was supposed to be the sequence 8, 3, 0, 7, 3.

My second experiment was to loosen the sparsity, rather than tighten it. Even with a normalized softplus, it's very easy to create almost-zero probabilities, which might be hard to reverse later. So I had the controller output the weights $\mathbf{w_r}$ without any sort of normalization or exponential post-processing, and instead added a regularization term of $b\ (\text{sum}(\mathbf{w_r}) - 1)^2$ for each read head r. (I chose b to be a large constant, much larger than the regularization constant used in the first experiment.) This of course means that some of the elements of $\mathbf{w_r}$ may be negative, but there's nothing mathematically wrong with that in the model.

As we see on the graph below, this modification did not help us with respect to the loss function.

Unfortunately, both curves are the same color. The modified one is always above the original, until they intersect at slightly after 4.000k, due to the sudden spike at the end of the original



Unlike our first modification, this one still produces good image predictions, though:

 (*Iteration 2700)*

 (*Iteration 2820)*

 (*Iteration 3780)*

 *(Iteration 4140)*

These are comparable in quality to the original, and (I think) are in fact slightly better.

Conclusion

Using the regularization techniques shown in class, I was able to gain a deeper understanding of the role that sparsity plays in memory-augmented machine learning algorithms. While the output of the trained attention network will indeed be sparse, it is unwise to add sparsity in the beginning. In fact, the quick training of the GTMM algorithm relies on there *not* being too much sparsity in the initial stages of training. In the future, I hope to develop an approach for these sorts of situations which can make the training even faster.

All code can be found at https://github.com/calvinleenyc/resolute/.