Deep Structured Similarity Matching

Aaditya Singh

Abstract

In this project, I decided to investigate Deep Structured Similarity Matching (DSSM). Specifically, I attempted to reproduce some of the simulations in the recent paper on DSSM of Obeid et al. The paper focuses on showing an equivalence between DSSM and a biological neural network trained using Hebbian and Anti-Hebbian learning rules. At the end of the paper, some empirical results on the MNIST dataset are stated. In this project, I aim to take these results one step further by empirically analyzing the network dynamics, seeing how costly it is to simulate them completely, and trying to cut down on the training time for such a network by taking shortcuts during the network dynamics phase.

1. Introduction

In the field of modern deep learning, most networks are supervised and trained with gradient descent (the "backpropagation" algorithm specifically). However, backpropagation is likely very different than the true learning algorithms used by the brain. Ever since its conception in 1949, Hebbian learning has been posited as a biologically plausible learning rule. The basic premise of Hebbian learning is "neurons that fire together, wire together" Typically, this translates to some weight update for update w_{ij} related to the product of the input x_j and output r_i. Often, a term is subtracted for stability (one can think of this as an analog to weight decay in neural networks).

Recent research in Hebbian Learning based network primarily focuses on unsupervised representation learning. Since Hebbian learning works without a lable, it's amenable to be used to compress higher data (which is also a plausible use of Hebbian Learning in the brain. In this project, I consider the paper from (Obeid et al., 2019), which shows the equivalence between Hebbian (and anti-Hebbian) learning update rules and Deep Structured Similarity Matching.

Similarity matching has a simple objective: preserve input dot products (with the representation's dot products) subject to some regularization. Deep Similarity Matching is just like Similarity matching, but with multiple layers. Deep Structured Similarity Matching as formulated in the paper arises from adding constraints on weights connecting layers. For example, a common choice made in standard computer vision is the use of Convolutional Neural Networks, which have a local receptive field. DSSM allows for such local interactions to be emphasize. For brevity, I leave out the more involved math and refer the interested reader to read the original paper. The reader may also find it useful to refer to (Pehlevan et al., 2018) for some of the other math. We now go into more detail into the actual architectures considered, the format of the training iterations, and how they might be really slow.

2. Methods

In this paper, we focus on the MNIST results (section 6.3 in Obeid et al. (2019)). Surprisingly, this section is not devoted much time. The architecture is mentioned to consist of a one layer locally connected biological neural network with lateral inhibition (also local). We denote the forward weights as W and the lateral weights as L. Furthermore, the parameter the authors investigate is also architectural the "Neurons per site" (NPS), which indicates how many neurons are at each "spatial" location in the output layer.

In terms of actual training for the empirical results, very little is said in the relevant sections. From earlier in the paper, we know that each iteration consists of two steps:

1. Run the recurrent network dynamics with constant input $\mathbf{x}$ (the flattened image) until convergence:

$$\tau \frac{d\mathbf{u}}{ds} = -\mathbf{u} + \mathbf{W}\mathbf{x} + (\mathbf{L} - \mathbf{I})\mathbf{r}$$

$$\mathbf{r} = f(\mathbf{u}) = tanh(\mathbf{u})$$

2. Update the weights according to the Hebbian and Anti-Hebbian rules:

$$\Delta W_{ij} = \eta(\mathbf{r}_j \mathbf{x}_i - W_{ij})$$

$$\Delta L_{ij} = \frac{\eta}{2}(\mathbf{r}_j \mathbf{r}_i - L_{ij})$$

Note that I have simplified the updates from what is given to the paper since the network we consider only has one layer. No other hyperparameters on training are given.

Also note that, the unsupervised network will just output representations. To assess these representations, the authors train a Linear Support Vector Classifier (LSVC).

For my extension of the paper, I started from the author's code in the paper. This code was extremely messy at first read; the majority of the code constructs a single huge matrix to encapsulate $\mathbf{W}$ and $\mathbf{L}$. I presume this was done to vectorize as many operations as possible for performance reasons. Despite the numerous steps taken for better performance, I added my own enhancements:

1. To train the LSVC, the authors sequentially run each image through the network. My first modification to their code was to vectorize the network dynamics simulation to allow for batching. This change led to about a **60x speed improvement** in calculating representations over multiple images.

2. The convergence criterion in the code seemed to be flawed (at least for the one layer case). I modified this to reflect convergence more accurately. My criterion was simply $||\delta\mathbf{u}||/||\mathbf{u}|| < T$ for some threshold T.

After these fixes, I was ready to test my own ideas. I primarily focused on how long the network needs to be simulated for good performance. In the original code, the network simulation had a cap of 3000 iterations and a threshold value of $T = 10^{-4}$; I experimented with changing these numbers (see Results). There was also a decaying step size for the simulation (with a minimum step size). I did not modify this. A similar decaying step size was used for the actual weight updates. Neither of these decay terms were mentioned in the original paper.

Besides changing the threshold and maximum number of iterations for the neural dynamics, I experimented with *continuous inputs*. In this scenario, the $\mathbf{x}$'s are streaming. I held the $\mathbf{x}$ constant for a certain (small) number of iterations then switched to the next input (without re-zeroing out $\mathbf{r}$). I tried this for changing $\mathbf{x}$ every iteration and every 10 iterations of the dynamic. Note that this equates to the usual update, just now we initalize our neural dynamics with non-zero $\mathbf{r}$ corresponding to the $\mathbf{r}$ at the end of the previous training iteration.

For monitoring training, the network was trained in "epochs" of 1000 random images from MNIST (with replacement). This is also in agreement with the author's original code. Every few "epochs" a new LSVC was trained using the new representations and train and test accuracies were recorded. I also recorded the number of iterations it took the neural dynamics to converge (relevant for runs where the threshold value is changed) and the values of $||\delta\mathbf{u}||/||\mathbf{u}||$ per iteration of the neural dynamics for every iteration of training.

3. Results

3.1. Accuracy

As seen in Figure 1 below, the most surprising result is that seeing 1000 images (or 15000 images in the case of the default settings) is enough for each network to peak in performance (in terms of its representation). With respect to thresholding and truncating iterations, we see the expected tradeoff of accuracy for speed (since smaller iteration caps and smaller thresholds are significantly faster). The benefit of continuous input is also interesting, since the paper does not consider this possibility. As suspected, allowing the network to converge further does always yield better results. However, the effect does seem to diminish if the network is run for 10 iterations (so maybe it's not necessary if the network is allowed to converge). It's also interesting that batching retained the performance of not batching methods, as it's three times as fast (but likely not biologically plausible).

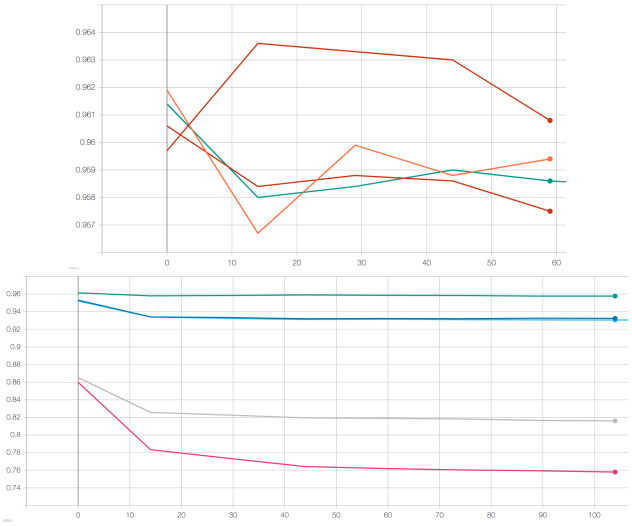

Figure 1. Plots of Test Accuracies over "epochs". These accuracies were obtained every 15 epochs by training an LSVC on the network-simulated representations for each training image in MNIST. For reference, the performance of LSVC on MNIST directly is 91.5%. (top) Effect of varying convergence threshold, batch size and random iteration through images on accuracy. Highest red curve are the default settings (batch size 1, random seed 0, $T = 10^{-4}$). Green curve is $T = 10^{-2}$, batch size 1, random seed 0. Bottom red curve is $T = 10^{-2}$, batch size 1, random seed 1. Orange curve is $T = 10^{-2}$, batch size 100, random seed 0. Note how all accuracies peak early then decrease (surprisingly). (bottom) Effect of using a fixed small number of iterations and/or continuous input. Green curve is same as in top subfigure for reference. Dark blue is 10 iteration cap within continuous input (every 10 iterations). Light blue is 10 iteration cap without continuous input. Silver is 1 iteration cap with continuous input and pink is 1 iteration cap without continuous input. Note that these curves also all decrease with more epochs. Continuous input always helps.

3.2. Performance

Next, I dove a bit deeper into the runtimes of the methods. For the top of Figure 1, the $T = 10^{-4}$ curve ran for 7 hours, compared to just 1.5 hours for the $T = 10^{-2}$ curves and just 0.5 hours for the batched version (orange). Figure 2 (below) makes the reason for this evident. The default curve ($T = 10^{-4}$) often has to run its dynamics for thousands of iterations, compared to the $T = 10^{-2}$ ("thresh-2" in the figure) curve only requiring at most 50 iterations. Since runtime scales linearly with number of iterations, this slight drop in accuracy ($\sim 0.3\%$ based on Figure 1) is likely worth the nearly 5x speedup.

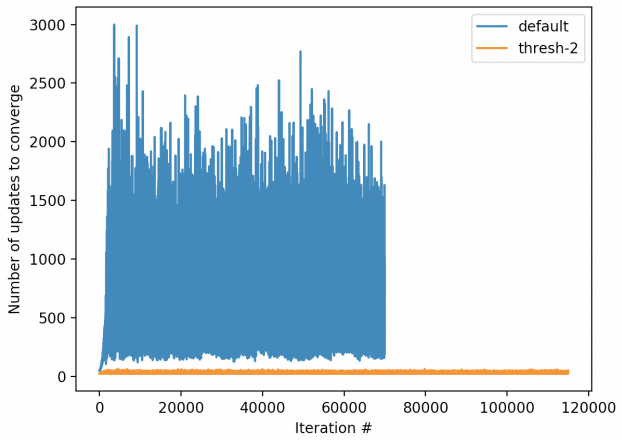

Figure 2. Plots of number of updates simulated of the neural dynamics per iteration of training for two runs. The blue curve shows the default settings, with $T = 10^{-4}$ and the orange curve shows $T = 10^{-2}$. The plot clearly shows how much faster (requires less updates to satisfy the threshold) it is to use a higher (less accurate) threshold.

4. Discussion

The baseline result in this paper matches the result provided in the original paper (our best accuracy: 3.79%, compared to 3.87% reported in the paper). However, this project goes through a lot of the gaps that are glossed over in the original paper. Furthermore, the project provides some key performance enhancements, without which the results could not be obtained. For example, one of the key findings it that the representation itself does not improve after the first few thousand images (despite the 60000 training images in MNIST). This discovery was possible due to the speedup of obtaining representations for all images to feed to the LSVC. It's also important to note, however, that the full model (network+LSVC) is still seeing all the images (since LSVC trains on all of them). An interesting direction for future work would be to limit LSVC to only train on the images shown to the network, and see if the DSSM then allows the joint model (network+LSVC) to generalize faster than canonical deep networks (which usually have to see all the data multiple times before generalizing). The hope with this future direction would be to show an empirical use for DSSM (instead of just using the technique as a biological parallel).

Another interesting result was the benefit of continuous input. Although further study would be necessary to quantify the exact effect, it appears that continuous input does not hurt the network, which is promising since biological networks are likely closer to continuous input than this instant "zero-ing" of all the outupt neurons between iterations.

With regards to the results on batch sizes, although the computational speedups are immense (similar to the difference between stochastic gradient descent and batch gradient descent in canonical deep networks), there's likely no biological analogue. A batch update would require neurons keeping a running history of activations while independently responding to other inputs in the batch.

Although I didn't verify the results from the paper for higher numbers of neurons per site (due to computational restraints), I think it's interesting that model performs better and better with more neurons per site (since this would make the representation grow larger and larger than the input dimension). It would be interesting to train a model (using some cloud computing service) and look at how sparse the representation is.

In the end, this project lent some performance improvements to the original code, contributed some bugfixes, and some greater understanding of the importance of network convergence before weight updates. Personally, I really enjoyed working on this project and found it very fulfilling.

5. Code

All code for this project can be found at https://github.com/aadityasingh/Deep_Structured_SM.

References

Obeid, D., Ramambason, H., and Pehlevan, C. Structured and deep similarity matching via structured and deep hebbian networks. In Wallach, H., Larochelle, H., Beygelzimer, A., dAlché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 15377–15386. Curran Associates, Inc., 2019.

Pehlevan, C., Sengupta, A. M., and Chklovskii, D. B. Why do similarity matching objectives lead to hebbian/anti-hebbian networks? *Neural Computation*, 30(1):84–124, 2018. doi: 10.1162/neco_a_01018. URL https://doi.org/10.1162/neco$_{a0}$1018.$PMID$: 28957017.