Modern Hopfield Networks on the CIFAR10 dataset

Haosheng Wang, Edrick Guerrero, Alfonso Gordon Cabello de los Cobos

Introduction

Memory involves the efficient storage and retrieval of information, and it comes in various forms—short-term, long-term, sensory, procedural, among others. Hopfield networks, also known as associative memories, are a class of recurrent neural networks (RNNs) designed to function as content-addressable memory systems. A defining characteristic of these networks is their ability to reconstruct entire patterns from partial or noisy inputs.

The original Hopfield network, introduced by John J. Hopfield in 1982, was based on binary feature representations and binary activation functions. Since then, significant advancements have been made. Modern Hopfield networks generalize the original model to continuous states, dramatically increasing storage capacity and stability. These developments, particularly in networks with continuous dynamics and large memory capacity, have been explored in a series of works since 2016.

In this project, we focus on replicating and analyzing some of the main components of the 2020 paper *"Hopfield Networks is All You Need"*, which presents a modern formulation of Hopfield networks. Specifically, we focus on the task of “image remembering” – given a corrupted (masked / noise-added) image (of the stored images), retrieve the most similar image remembered/stored by the network.

Methodology

Our implementation exists on a Python Notebook, but can also be run on an external Python file. Given that the original implementation was made in PyTorch, we reimplemented it in TensorFlow and used a different dataset of images, specifically CIFAR10. We also have experimented with different architectures to achieve better performance on the “image remembering” task.

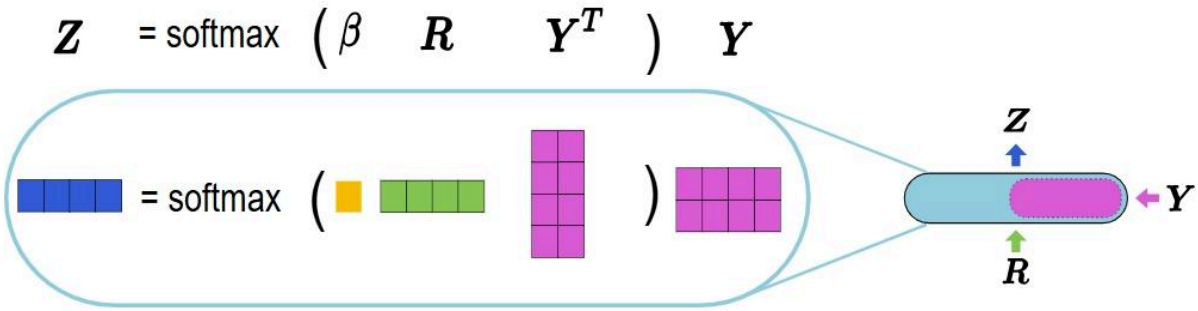

Figure 1: HopfieldLayer architecture. In our case, the stored images correspond to Y, the corrupted images correspond to R, and the scaling factor corresponds to beta.

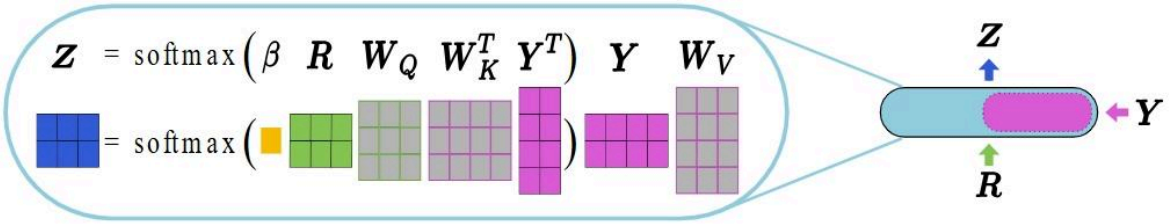

Figure 2: Hopfield architecture. The signatures in the model have similar meanings as demonstrated in Figure 1.

We began by choosing a number of images from the CIFAR10 dataset (for the network to remember). The original dataset contains 60000 32x32 rgb images. We preprocessed by taking a subset of the images and made a gray-scale copy of it, resulting in train_rgb and train_gray[1]. Then we corrupt the images in either of the two methods – (1) mask the bottom half of the image (2) add Gaussian noise (with custom mean and standard deviation) – and get test_rgb and test_gray[2]. The train set represents the original images to be remembered by the network, while the test set represents the corrupted images used for the network as query to reconstruct the original images.

The core of our implementation exists in the hopfield_tf class, which is “mostly” the tensorflow version of the pytorch HopfieldLayer as shown in Figure 1. The model has two modes – static mode and non-static mode. Static mode of the model maintains no trainable variables. At each, call the model will take in a number of stored images (K, key) and a number of corrupted images (Q, query), and a number of output projection images (V, value)[3]. The input stored images, corrupted images and output projection images are treated as K,Q,V. The K and Q are associated to get a score which is repeatedly fed through softmax until either the max update

[1] In our notebook demo (FinalProjectNB.ipynb), train_gray is defined as train_set and later renamed to stored. This is because we first experimented with the gray scale image and then added the same experiment on rgb images.
[2] Similarly, these are named a little differently in the notebook (e.g. test_set, masked, noisy_imgs, …)
[3] For our task, the output projection images (V) are just the stored/remembered images (K).

step is reached or the energy change[4] between update steps are smaller than some epsilon. The association scores are then used to map with output projection images to get the final output (i.e. reconstructed images). The non-static mode (as shown in Figure 2) follows the same procedure except the model contains trainable variables K, Q (and optional V) matrices. The K and Q matrices will map the input stored images and corrupted images into some hidden associative space, where the feature association happens. The optional V matrics can further map output to some other dimension for other tasks[5].

A main divergence we included from the original was a different method of comparing patterns in the images: calculating the distance between the pixels rather than the dot product[6]. This distance revision can actually break the stability of the original model, but greatly improves the performance on our task in some settings.

We implemented two algorithms to measure the difference and similarity between images – MSE and Vector Similarity. MSE (Mean Standard Error), calculates the pixel-by-pixel difference of every image pair. Vector Similarity is calculated using OpenAI's CLiP model, which vectorized images and calculates their similarity using the COSINE similarity function. These two algorithms can be used in several places in our project. First, it can be used to compare the reconstructed images and original images to measure the performance of our model. Second, it can be used on the original images and masked images to measure the quality of corrupted images (i.e. how much do the corrupted images deviate from the original images). Third, it can be used on the original images to measure how similar the original images are to each other (because correlated images can greatly decrease the performance of the model)[7].

Results

We have applied the model on both gray scale and RGB images. The images are corrupted using either methods of adding mask or adding noise. We have initialized the model with different scaling factors with static mode. We have first experimented with dot products (same as pytorch) for association, but then moved to use distance for association.

We have done a variety of experiments using different settings and found some really interesting results. The hyperparameters we have played with are combinations of the following:

1. The number of images to be stored
2. The scaling factor
3. The corrupting method (mask/noise)
4. The max number of update steps

---

[4] The measure of change in energy is measured implicitly by the change in the state of the network, please refer to the original paper for more details.
[5] this is not needed in our task for now, so we didn't include it yet
[6] Set test_mode = True when init the model for using distance for association
[7] In our current implementation, we have just applied the metrics on the reconstructed images and original images to measure the performance.

5. Gray or RGB images

There are also other settings to be experimented with but we haven't yet (because of limited time), for example:

1. Different amount of noise (mean, std)
2. Different amount of mask (instead of bottom half, maybe random pixel mask or others)
3. Different quality of stored images (i.e. highly correlated or less)
4. Non-static mode (OR feature extraction for preprocessing)
5. (important) Other potential association methods (e.g. dot product, distance, others…)

We have included some representative graphs here for a quick idea, but please refer to our github repo for more experiment results.

First, let's take a look at the experiment result for the following setting: 100 rgb images to store, scaling=1e6, mask bottom half, max update step = 5 (Figure 3, 4, 5). Figure 3 and 4 shows examples of success and fail cases of our model. Figure 5 shows the overall res of the 100 images. Our model generally performs better than the pytorch version under this setting. The MSE is far better, but the vector similarity suggests that only a relatively small number of images are correctly retrieved (as only a vector similarity at 1 shall be considered correct retrieval).

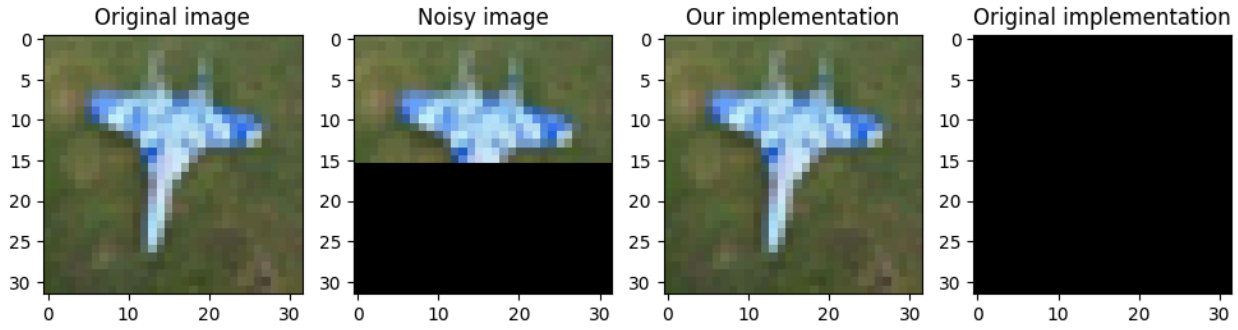

Figure 3: res_100rgb_1e6_mask_success1

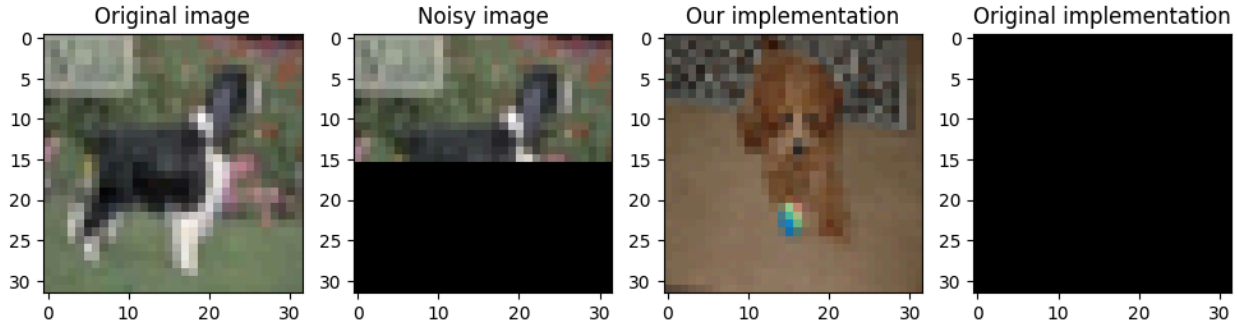

Figure 4: res_100rgb_1e6_mask_fail1

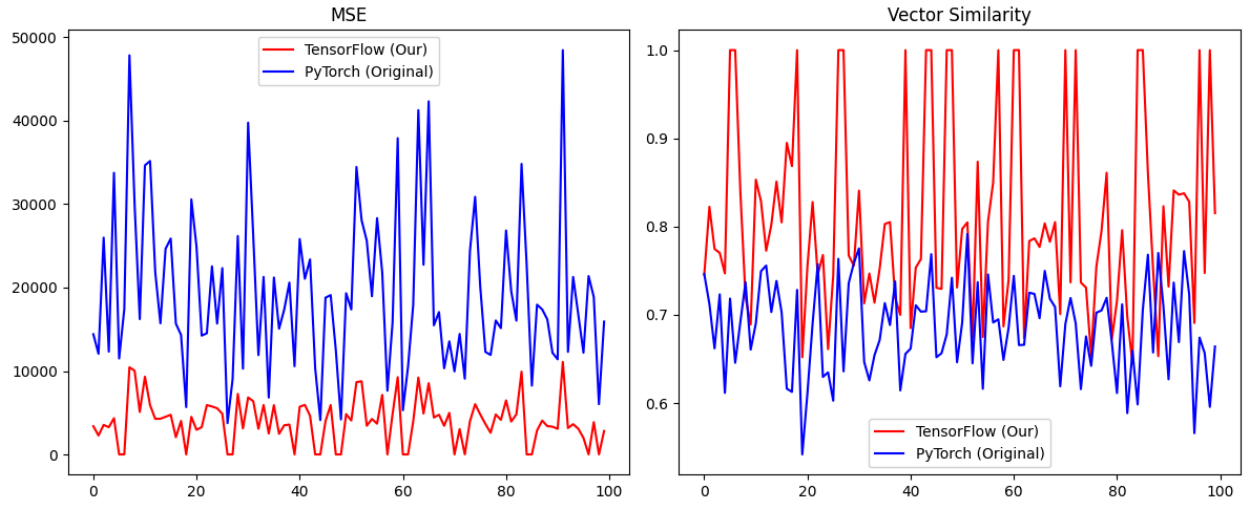

Figure 5: plots_100rgb_1e6_mask

We also tested on 20 gray scale images(Figure 6, 7, 8, 9, 10, 11). Notice that larger max update steps and larger scaling factor both correspond to more clear retrieval (fig 6,7). This is because both would result in a final softmax score of greater variance. However, larger max update steps is not the same as larger scaling factor, as Figure 8 correctly retrieves the original image by increasing the scaling factor while Figure 7 didn't with a larger max update steps.

Figure 9,10,11 demonstrates that the quality of corrupted image shall be considered carefully when measuring the performance of the model. The Gaussian noise with a mean of 0 and std of 5 shall be considered much less corruption as compared to completely mask the half bottom of the images. That's why our model seems perfect in Figure 11 (i.e. MSE 0 and Vector Similarity 1). But the interesting thing is that the pytorch original version model performs much worse than expected in such setting (Figure 11). We believe this is because the original model have used dot product for association, which is buggy. For example, if we let the network remember a bunch of images and a full white image, then whatever image we used for retrieval would result in the full white image as the output. This is because the full white image has 255 in every pixel (thus every feature), which would always result in the biggest dot product making it almost always 1 after softmax. We observed this and thus have changed the way of calculating association scores. Instead of dot product, we calculate the absolute value of the per-pixel (per-feature) difference. We would consider two items more similar if their distance are smaller. This significantly improves the performance as compared to the original version (Figure 5,9,10,11) and achieves perfect performance in some settings (Figure 11). Yet this could also break the stability of the model in some sense. As modern Hopfield networks are modeled as energy models, the original paper have mathematically proved why they would converge to three types of limit points in almost always one step. Our revision may break some of it.

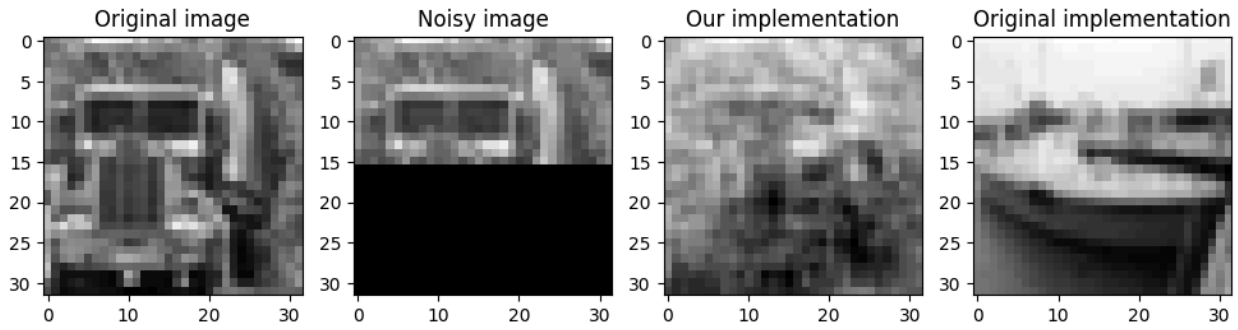

Figure 6: res_20gray_1e5_mask_step2_fail1

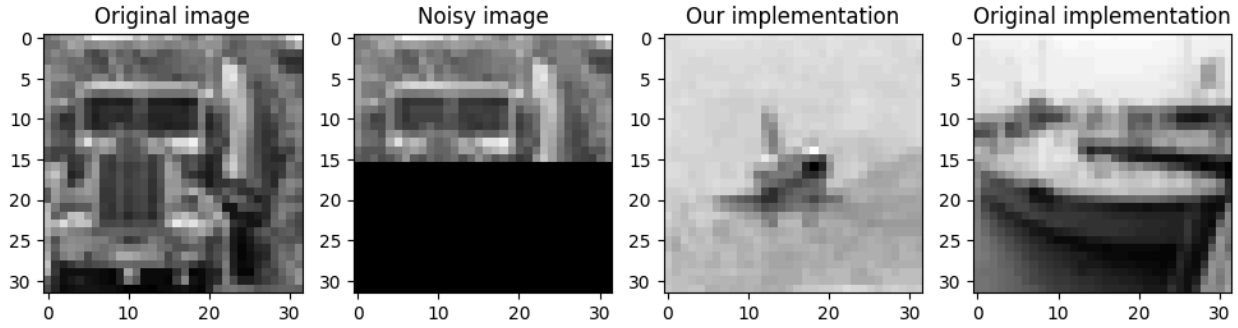

Figure 7: res_20gray_1e5_mask_step5_fail1

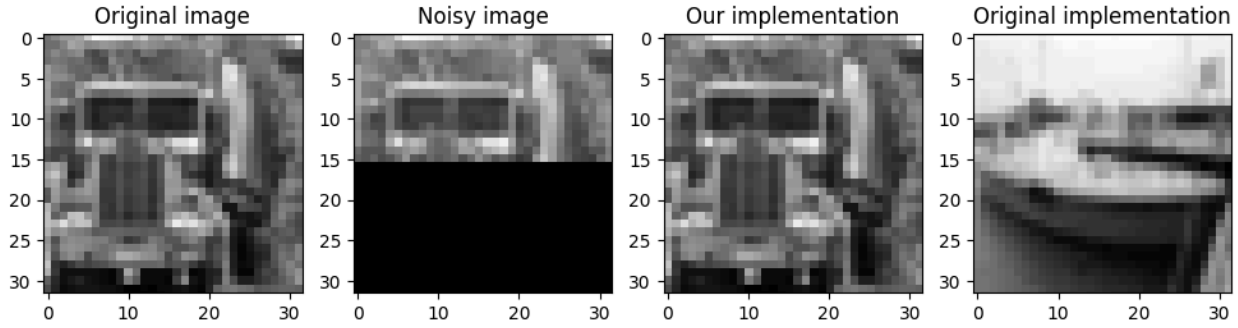

Figure 8: res_20gray_1e6_mask_step2_success1

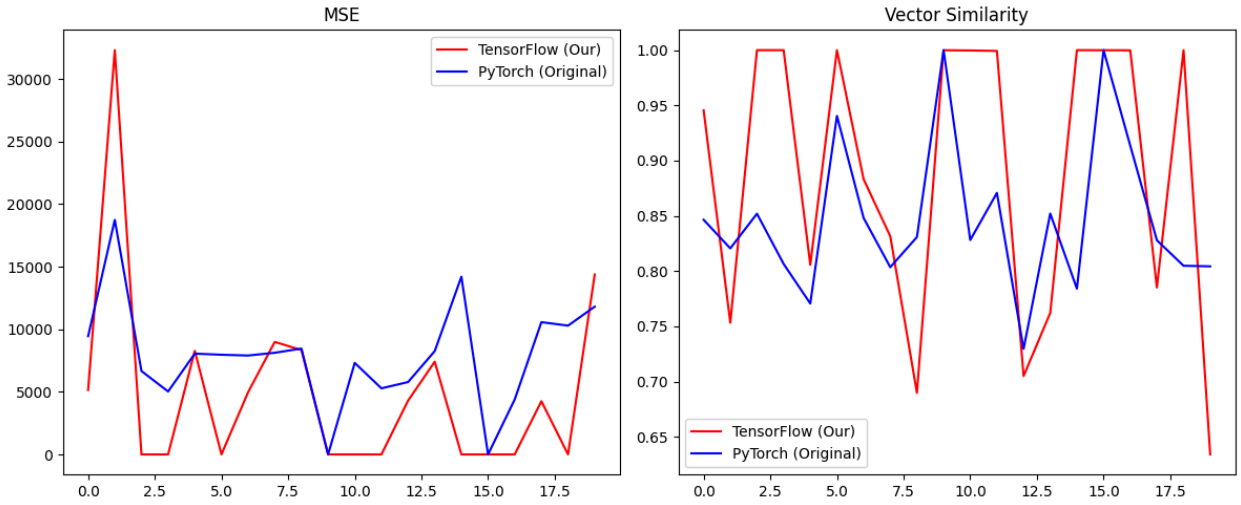

Figure 9: plots_20gray_1e5_mask

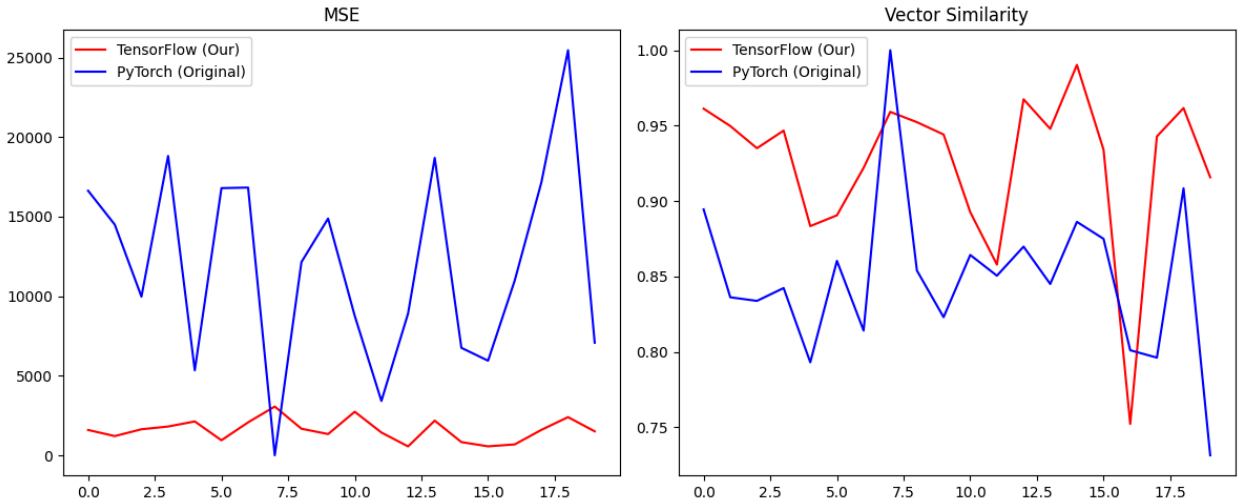

Figure 10: plots_20gray_1e4_noise

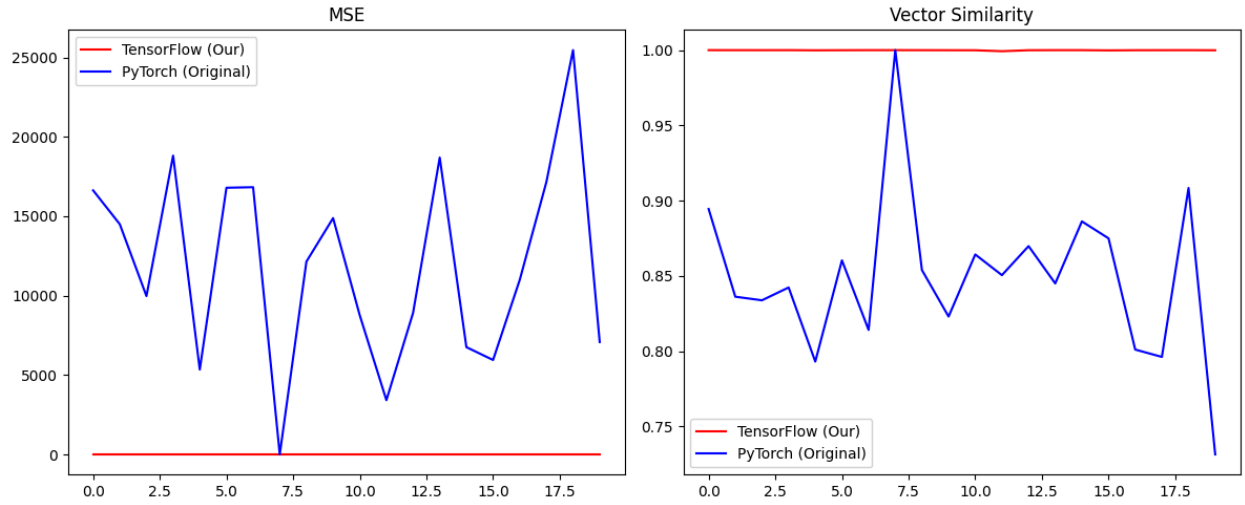

Figure 11: plots_20_gray_1e5_noise

Challenges

The main challenge of the task remains on how to properly associate features. The original implementation have used dot product to calculate the feature similarity. However, this did not work out as expected in the setting that image pixel values are directly used as features. We improved the performance of the model in the “Image Remembering” task by using distance (l1 norm) for calculating association scores. We observed that the distance could result in unstable image patterns which suggest that we shall look more into the mathematical founding of our method and encourage us to also find better methods for association score calculation.

Another potential way to go is use feature extraction and represent the images as features instead of raw feature values. By properly learning and extracting image features using CNN (or other networks) and apply normalization techniques, we expect similar or better results. However, we didn’t go in this direction because we haven’t figured out how to properly work through the training process without overfitting. Notice that in order to properly extract the features, the model needs to learn how to extract based on some training set. However, considering that our task is to retrieve using corrupted images based on original images and also compare the reconstructed images with original images for loss, it’s hard to avoid that the feature extraction network overfit on the training dataset images. Furthermore, what we want to explore in the first place is long term memory. So we made an assumption that the images are just seen once and then need to be remembered. This is another big challenge too. We need to actually remember a large number of images (the experiments we have done are too small in this sense) implicitly and efficiently, and be able to recall the right one(s) with partial corresponding keys. How to store efficiently, in terms of time and storage space, is one of the most important challenges considered by the authors.

Reflection

1. **How do you feel your project ultimately turned out? How did you do relative to your base/target/stretch goals?**

   The project turned out to be more challenging than we had anticipated. We had expected the original pytorch would handle the Image Remembering task smoothly and successfully until we found that it cannot even remember the same image if it had remembered a white image using their model. That means even we have successfully implemented the original model, we still cannot even achieve our base goal (achieve good accuracy on small percentage of the dataset).

   Therefore, we decided to change the model architecture. We improved over the original implementation by using distance (l1 norm) for association score calculation. The experiments suggested that our change is successful and promising. However, since the changing architecture is unexpected and need extra careful consideration, we have very limited time left for finishing other things of the workflow.

   That said, we have perfectly improved the original model to achieve the base goal, and experimented some amount of quality and quantity of corrupted images to achieve the target goal. But we need more time to do the stretch goals.

2. **Did your model work out the way you expected it to?**

   Our initial implementation, which was the same model as the pytorch HopfieldLayer worked as expected (i.e. reconstructed the same images as the pytorch model did). However, the original pytorch model didn't work as expected (i.e. it cannot properly remember images). We changed the model architecture and that led to properly reconstructed images most of the time and worked just as expected.

3. **How did your approach change over time? What kind of pivots did you make, if any? Would you have done differently if you could do your project over again?**

   Yes, originally we expected to do image reconstruction by reimplementing the same pytorch model in tensorflow and focus more on the evaluation of the quality of images. But as we finished the implementation, we found out the original model doesn't work (so did ours). So we revised the model architecture – use distance (l1 norm) to calculate the association scores instead of dot product – and that works pretty well on the task.

   If we could have started the project again, we would have first experimented with the original model to make sure the model works as expected on our desired task before going into the implementation details.

4. **What do you think you can further improve on if you had more time?**

   If we had more time, we can research more about promising methods to calculate the association scores between images, try using a different workflow involving training and testing for the task (to see the model performance with weights), and evaluate the correlation between images in some way to measure the quality of stored images.

5. **What are your biggest takeaways from this project/what did you learn?**

   There would be many takeaways but the biggest way would be to properly structure the workflow before going into details. If we had found out that the original model doesn't work as they said on this task, we would have more time to solve the problem instead of knowing that in the last second.