1.a. Image representation

a. Network architecture and input – Three alternatives where considered:
   Full blown prior-learning with hypernetworks – as you've understood from my mail, this was my initial direction. This directions had two main reasons:
   i. I've misunderstood the memorization task as one of generalization in the usual learning context (e.g. producing acceptable results over un-seen data)
   ii. The subject of hypernetworks was new for me. It seemed super cool. And nice example was supplied…

b. Memorization with input and architecture change – Since the basic task remained the same (memorize), the main difference that is required:
   i. A way to convey the network the connection between inputs (x,y) and specific images (e.g. first image, second image, etc) so it would be able to differentiate between inputs of different images– this was obtained via addition of channels to encode the image number (e.g. 1-100) in the memorized set.
      My first 'go-to' encoding was to transfer the serial number of the image from decimal to binary encoding. Considered it would be easier to learn.
      This was the implemented encoding.
      Later gave it some more thought, and understood a simpler solution exist (see below)
   ii. Network that is capable to memorize more data.
      As a first naïve attempt I've simply set the hidden layer size higher then in the given 1 image example. My assumption was that in 'worse case' scenario if each sub-layer is dataset_size X size_for_single_image – it can always learn some sub-partitioning of the network per-image (e.g each image "triggers" a sub-net).
      Off course this interpretation is a cruel simplification of the process, and the network can learn to map and represent common patterns of the images (e.g. repeating colors, edges etc), combining them differently conditioned on the input encoding.
      For this reason, I've experimented with several (only a few) multiplication of the basic layer size (used in the example), ending up with nice (subjective) performance for hidden layer X10 the size of the original SIREN in the example to represent the 100 images. It was relatively fast compared to X50 (since the layers are fully connected, the number of weights grows quite fast with each multiplication) and results where on par.

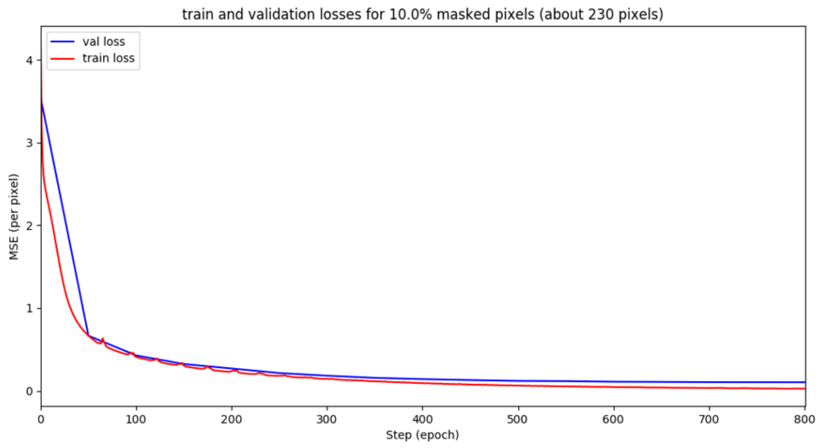
c. Memorization with input_ver_2 and architecture change – after implementing the first solution (which required encoding from decimal to binary and padding…) I've came to think that much more elegant solution exist.
   Instead of encoding each image with binary, I could've simply extend the grid creation process… adding another grid dimension, and mapping between it's sampled range and the number of the image. Then adding it to each 2D coordinate.
   Easy to learn bounded value. Extended to any size of data base (as the 2D grid dimensions with different resolutions). Much simpler… should of thought about it

before I've implemented 😊

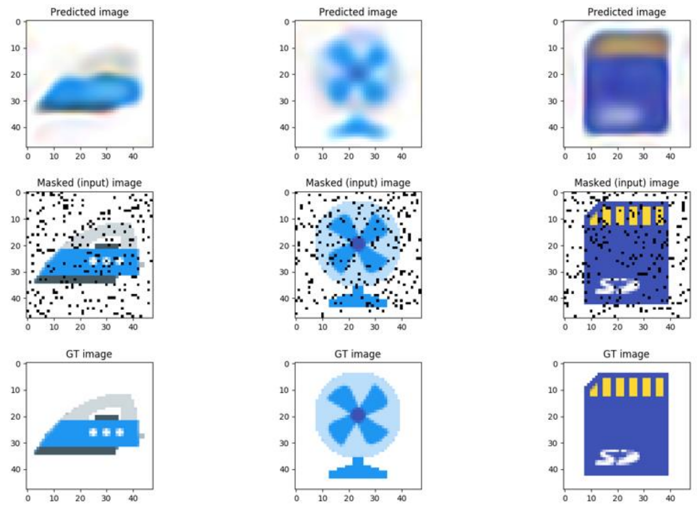
II. Generalization –

a. The first 'generalization' that came to mind, was that of common learning task – e.g. generalization to unseen examples. In our case, since for every input grid of an image we have specific output (so every pixel coordinate / rgb output is a learning example), we can create a test set by masking-out some portion of the image pixels, using them for validation purpose. In this specific case 10% of the pixels of each image where masked during training, and used for validation.
Those pixels should be fixed for each given image and the relevant predicted values during training should not reflect on the training loss (e.g. masked)
b. Another possible interpretation in the context of memorization might be the simple error between the represented dataset and the produced one. In some sense we expect the net to accurately represent/memorize the full data set, so any error can be viewed as generalization error of sort. I must admit this interpretation feels a bit less natural to me, and I need to convince myself this is a legitimate use of the term generalization.

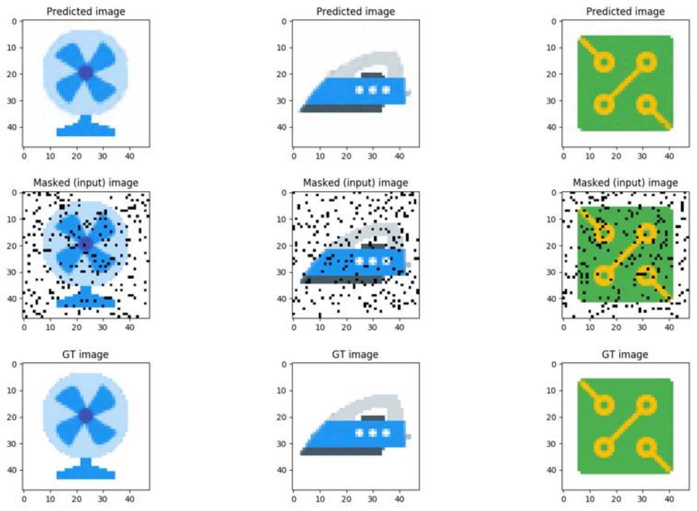
III. Networks generalization during training
Train/val loss:



Example image from step 50:



Example image from step 750



2.a. Up sampling demonstration

I. Up-sampling image from 48x48 to 256x256 with net trained on 48X48 images
Rows from bottom to top: Low res images (training set), High res image (for comparison), network prediction (up-sampled result).

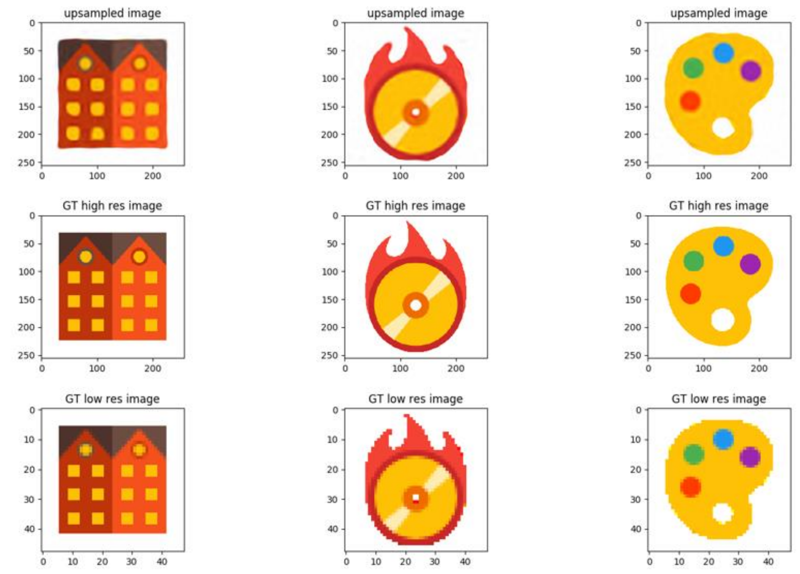


Fig 1 – Demonstration of up-sampling

2.b. Interpolation between selected pairs of images

I. **Similarity measure** –
In this task we'd like to (eventually) interpolate between pairs of image representations / embeddings.
Essentially, the "joint representation" network follows the simplified version of SIREN (with the expectance of hidden layers size), it is a basic predictor (and not auto-encoder for example). Hence, the activations of the last hidden unit were used as the embedding.
To model the embedding vector, several similarity measures can be considered.
Among them are Euclidean distance (or other distance metrics variations), dot product between

the vectors and cosine similarity (naturally more similarity metrics exist. For this task the 3 mentioned where considered)
When choosing an appropriate measure, several considerations where taken into account:

- *Activation sparsity* – the chosen architecture simply enlarges the networks memorization capacity via hidden layer size. Since each of the input signals for a given image could've been represented by a smaller network (as was shown in the original SIREN example), it's quite possible that for any given input only some part of the network (or in our case hidden layer) would be active.
- *Encoding effective length* – a somewhat similar subject. Some of the inputs might require less 'bits' (or activations) then others to be properly represented. As intuition we can consider an image with more frequent color change/ more edges, vs. relatively homogenous image.
- *Desired similarity result* - we would like to measure similarity between two high-dimensional vectors, while emphasizing importance of activations in similar places and over the similarity of "total energy" / amplitude of a given comparison.
  We'd like two high-dimensional vectors to point in a similar direction (non-zero components in similar places, with similar sign), with less importance given to the exact amplitude.

The similarity measure chosen for the task at hand was the cosine similarity measure.
For two vectors x, y the similarity value can be defined as the angle between the two vectors.
More formally $similarity = \cos(\theta) = \frac{x \cdot y}{\|x\|\|y\|}$.
The output of the similarity measure vary in the range of [-1,1] with 1 being similar and -1 being dissimilar (Vectors pointing to opposite directions)
Vectors with resembling values in similar places would have higher similarity score.
Since we normalize by the norms of the vectors, we attend to the challenge of comparing representation of different length properly (avoiding measure "skewness" for images with very sparse of very dense representation). Another "nice" related property is that it enables us to avoid situation in which similarity spikes due to (relatively) high values in some small subspace of the high-dimensional vector.

* Caveat

1. Usually this similarity measure is applied to signals in the interval of [0,1], rather then [-1,1] as in our case. Tried this comparison with and without proper range transformation without significant difference in result.

II. **Selection of images for interpolation** -
For this task two aspects where considered:
- The "interesting" characteristics of the interpolation (or travel along the some manifold of the embedding space) can be better observed when interpolating between examples which are far apart in the embedding space.
- Interpolating between examples which are adjacent in the embedding space can be prettier (😊) and serve as a sanity check for the similarity measure.
Both examples presented below.

When choosing the dissimilar embedding some “cheating” took place. Since some specific images can be far from many others in the embedding space (which was the case as least for this specific choice of embedding and similarity measure), some additional effort was done to present a variety of image pairs with different images in each pair (a sort of naïve max suppression)

a. Dissimilar

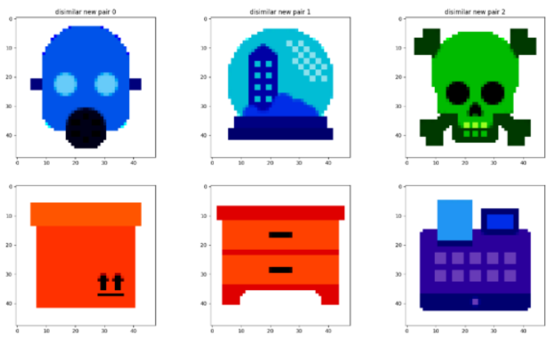


Fig 3 – Images with dissimilar embedding

b. Similar

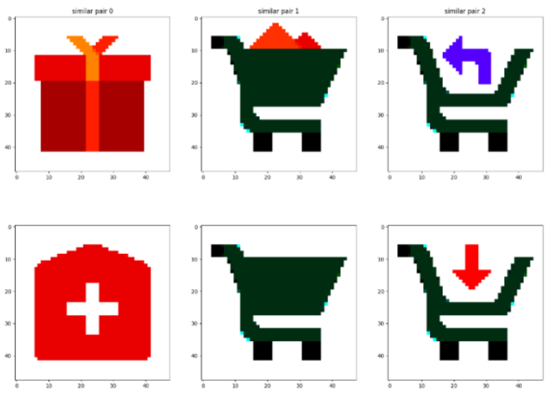


Fig 4 – Images with similar embedding

III. **Demonstrate interpolation**

a. Dissimilar

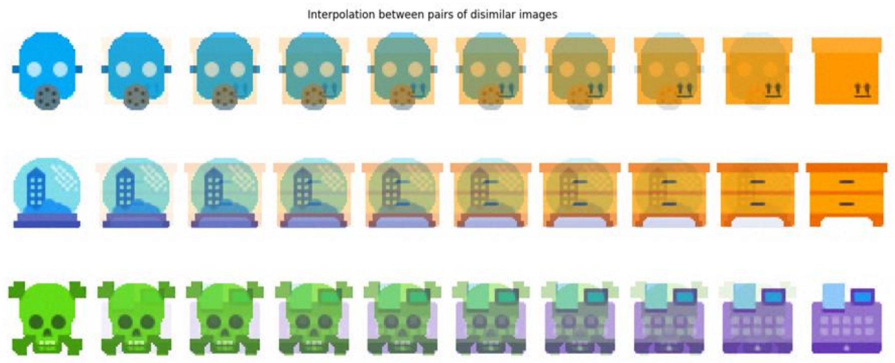


Fig 5 – Interpolating over dissimilar images

b. Similar

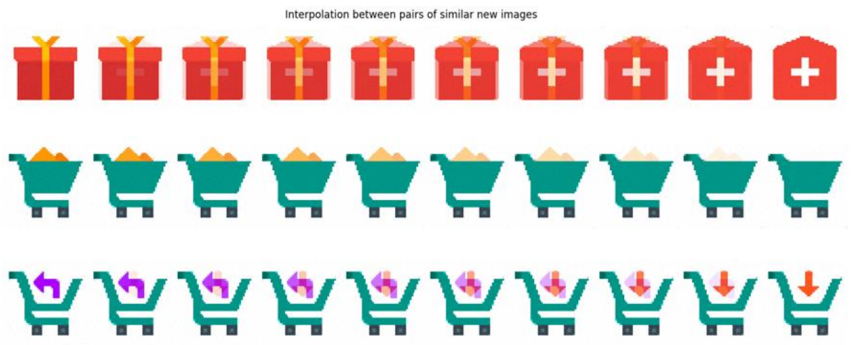
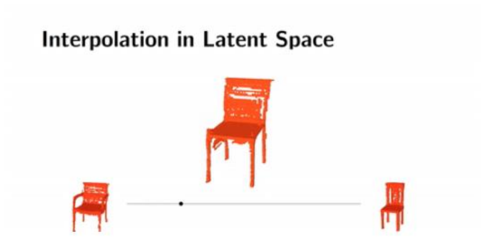


Fig 6 – Interpolating over Similar images

IV. **Interpolation shortcoming**

When traversing the embedding space we would like to see some “smooth” (what is smooth? I’ll try to give some intuition later) transition from one image to another, where new values (“sampled” between the two representations) can potentially produce some new and interesting results (not exactly one image but also not the other. Rather some representation of an area of the embedding space manifold learned/approximated via common features from different examples.

One example that I’ve had in mind when approaching the task is this one:



Interpolation in Latent Space. Source: Hackernoon Latent Space Visualization.

Fig 7 – Interpolating in latent space

Source

In our case, the interpolation looks more like gradual alpha blending like between two images. Each intermediate step simply looks like a combination of the two with some transparency factor.

Off course this could be a direct result of my choice of embedding/representation of the last hidden layer activation (which seemed like a great idea until I saw those results).

But setting this a side, one can hypothesize the following about our ‘blending’ effect:

- ‘Poor’ representation of the imbedding space – Since no adaptation specific adaptation of either the network architecture or the training data was done for this task, we remain with the basic SIREN model with enlarged hidden layer size. This was great for memorization, however might be less then perfect for embedding learning.
  We learn a high dimensional embedding space with relatively little data. The larger our embedding space and the less data we have, the harder is to learn a “compact” embedding with many of it’s manifold areas “covered” during learning.
  So in our case, large non-compact embedding space with little data can potentially map the different data points to different/far areas in the embedding space with little information regarding the transition from one area to another. In other words the representation of the different images are far away, and the transition in not smooth (consider a very rough signal quantization).
- Since we use the last hidden layer as our embedding – we don’t leave much room for interpretation. We have one linear layer to provide results for all our different combinations (sampling) this by itself will have some tendency to show the interpolation

as sort of linear interpolation from one image to another (which might be a simpler and better explanation, and makes me wonder if the choice of the last hidden layer was smart...).

3.a+c. Design of improved solution and explain why this should produce refined solution:

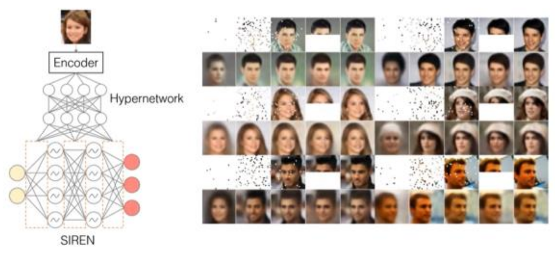
Following up on the assumption of ‘poor representation of embedding space’, and use of last hidden layer (followed by a linear one), an improved solution would relate to several key points:

1. Compact embedding space, not at the end of the architecture – in other words a sort of encoder-decoder. This will attend to both aspects.
   First, by choosing a bottleneck layer which is smaller then both previous/following hidden layers we force a more compact space representation (compared to the current one).
   Second, since its followed by several stacked non-linear hidden layers, a linear interpolation/sampling between vectors in the embedding space will undergo additional non-linear process with high representation capability, hopefully resulting in a “less linear” interpolation between the two image.

2. Additional data – Attending to the “mapping” of the (compact) embedding space, we can additional data either via augmentation (in color space and image space) or by presenting the net with additional images from other data sets which we consider relevant to our task (if we stay with multi-image embedding, we should start with images of “similar” characteristics (e.g. png images of simple forms), and consider the effect of more images to memorize on the architecture of the net.

3. Generalization beyond memorization – As presented in Sitzmann et al. work, it is possible to learn a “prior of the space of function parametrized by SIREN” (see figure below). In this setting we follow the idea presented in “hypernetworks”. Here instead of implicitly learn the parameters of the SIREN network we have an additional hypernetwork to generate the weights of the specific SIREN network conditioned on the input (so instead of one large very redundant network that memorize all of the images, we learn how generate SIREN params of the “same” compact network so that the generate parameters would represent the image relevant for the inputs.

Learning priors over the space of SIREN functions



In our case this means advancing with the prior idea of encoder-decoder a step further, using the encoder prediction as the weights of our SIREN network.
Why is that better? We move our embedding space even higher up the ladder.
The authors show the generalization capability of such approach (learning to predict the image from sparse pixels). Intuitively I'd hypothesize this generalization (or ability to "learn the essence" of the image) contributes to the smoothness compactness of the embedding space.
This is obviously more of a 'hand waving'. But I would try it to see how it works

Commented [TF1]:

3.b Implementation details

1. Use of encoder-decoder architecture –

- Change the basic SIREN architecture to have a bottleneck.
- Use 5 hidden layers with the middle hidden layer smaller (some controllable parameters, start with 10 times smaller) then previous and following layers.
- Use the small hidden layer as the embedding / latent space.
- Consider required changes due to larger architecture:
  - Number of epochs, learning rate
  - Size of "big" hidden layers (can it be smaller now? Play with hyperparameter)

2. Additional data

- Add augmented data
  - Flip, rotations, crops
  - There's a list of color augmentation (brightness, contrast, saturation, hue).
    However I'm not confident they are relevant to our "simple" thumb images.
    I need to consider if this is relevant and if so which. Wouldn't start here

- Additional dataset – consider some additional png data sets (many available. Finding one that is simple enough, as in given data would be a bit more challenging but probably possible)

3. Hypernetwork

Following the trainings examples by Sitzmann et al. use some conv encoder and define its output as the network weights of the SIREN, utilizing torchmeta HyperNetwrok module to represent it.

As mentioned, this was my initial (too complicated) direction, before I’ve fully understood the multi-image embedding task.
In this case the output of the convolutional encoder (e.g. all SIREN params) would serve as the latent space.
It’s a vague explanation (not exactly implementation details).
Following the example to implement it should be quite possible. However additional aspects such as the size of data needed to train the conv encoder (and the whole net), and possible changes in its architecture (making it smaller to accommodate data size) must be considered. Nevertheless, it’s interesting to try.