1 Deep FRAME Model

To fix notation, let $\mathbf{I}(x)$ be an image defined on the square (or rectangular) image domain $\mathcal{D}$, where $x = (x_1, x_2)$ indexes the coordinates of pixels. We can treat $\mathbf{I}(x)$ as a two-dimensional function defined on $\mathcal{D}$. We can also treat $\mathbf{I}$ as a vector if we fix an ordering for the pixels.

For a filter F, let $F * \mathbf{I}$ denote the filtered image or feature map, and let $[F * \mathbf{I}](x)$ denote the filter response or feature at position x.

A deep FRAME model is a composition of multiple layers of linear filtering and element-wise non-linear transformation as expressed by the following recursive formula:

$$[F_k^{(l)} * \mathbf{I}](x) = h\left(\sum_{i=1}^{N_{l-1}} \sum_{y \in \mathcal{S}_l} w_{i,y}^{(l,k)} [F_i^{(l-1)} * \mathbf{I}](x+y) + b_{l,k}\right) \tag{1}$$

where $l \in \{1, 2, ..., \mathcal{L}\}$ indexes the layer. $\{F_k^{(l)}, k = 1, ..., N_l\}$ are the filters at layer l, and $\{F_i^{(l-1)}, i = 1, ..., N_{l-1}\}$ are the filters at layer $l-1$. k and i are used to index filters at layers l and $l-1$ respectively, and N_l and N_{l-1} are the numbers of filters at layers l and $l-1$ respectively. The filters are locally supported, so the range of y is within a local support $\mathcal{S}_l$ (such as a 7×7 image patch). At the bottom layer, $[F_k^{(0)} * \mathbf{I}](x) = \mathbf{I}_k(x)$, where $k \in \{R, G, B\}$ indexes the three color channels. Sub-sampling may be implemented so that in $[F_k^{(l)} * \mathbf{I}](x)$, $x \in \mathcal{D}_l \subset \mathcal{D}$.

We take $h(r) = \max(r, 0)$, the rectified linear unit (re-lu), that is commonly adopted in modern ConvNet. We define the following random field model as the deep FRAME model:

$$p(\mathbf{I}; w) = \frac{1}{Z(w)} \exp\left[\sum_{k=1}^{K} \sum_{x \in \mathcal{D}_L} [F_k^{(L)} * \mathbf{I}](x)\right] q(\mathbf{I}), \tag{2}$$

where b_k is the bias term, $w = (w_k, b_k, k = 1, ..., K)$, and $h(r) = \max(r, 0)$. $q(\mathbf{I})$ is the Gaussian white noise model.

Model (2) corresponds to the exponential tilting model with scoring function

$$f(\mathbf{I}; w) = \sum_{k=1}^{K} \sum_{x \in \mathcal{D}_L} [F_k^{(L)} * \mathbf{I}](x). \tag{3}$$

The learning of w from training images $\{\mathbf{I}_m, m = 1, ..., M\}$ can be accomplished by maximum likelihood. Let $L(w) = \sum_{m=1}^{M} \log p(\mathbf{I}; w)/M$,

$$\frac{\partial L(w)}{\partial w} = \frac{1}{M} \sum_{m=1}^{M} \frac{\partial}{\partial w} f(\mathbf{I}_m; w) - \mathrm{E}_w\left[\frac{\partial}{\partial w} f(\mathbf{I}; w)\right]. \tag{4}$$

The expectation can be approximated by Monte Carlo samples. One can sample from $p(\mathbf{I}; w)$ in (2) by the Langevin dynamics:

$$\mathbf{I}_{\tau+1} = \mathbf{I}_\tau - \frac{\epsilon^2}{2}\left[\mathbf{I}_\tau - \frac{\partial}{\partial \mathbf{I}} f(\mathbf{I}; w)\right] + \epsilon Z_\tau, \tag{5}$$

where τ denotes the time step, ϵ denotes the step size, assumed to be sufficiently small, $Z_\tau \sim \mathrm{N}(0, \mathbf{1})$.

We can build up the model layer by layer. Given the filters at layers below, the top layer weight and bias parameters can be learned according to

$$\begin{aligned}\frac{\partial L(w)}{\partial w_{i,y}^{(L,k)}} =& \frac{1}{M}\sum_{m=1}^{M}\sum_{x\in\mathcal{D}_L} \delta_{k,x}^{(L)}(\mathbf{I}_m; w)[F_i^{(L-1)} * \mathbf{I}_m](x+y) \\ &- \frac{1}{\tilde{M}}\sum_{m=1}^{\tilde{M}}\sum_{x\in\mathcal{D}_L} \delta_{k,x}^{(L)}(\tilde{\mathbf{I}}_m; w)[F_i^{(L-1)} * \tilde{\mathbf{I}}_m](x+y),\end{aligned} \tag{6}$$

and

$$\frac{\partial L(w)}{\partial b_{L,k}} = \frac{1}{M}\sum_{m=1}^{M}\sum_{x\in\mathcal{D}_L} \delta_{k,x}^{(L)}(\mathbf{I}_m; w) - \frac{1}{\tilde{M}}\sum_{m=1}^{\tilde{M}}\sum_{x\in\mathcal{D}_L} \delta_{k,x}^{(L)}(\tilde{\mathbf{I}}_m; w). \tag{7}$$

where $\mathbf{I}_m$ are M observed images, and $\tilde{\mathbf{I}}_m$ are $\tilde{M}$ synthesized images sampled from the model.

2 Experiment

For each input image, design a set of filters, and learn a Deep FRAME model. We define the Julesz ensemble as the set of images that reproduce the observed sufficient statistics over those designed filters. You use Langevin dynamics to draw samples from the model. Figure 2 shows an example of synthesis. The synthesis starts from a zero image and the sampling stops when it matches all the sufficient statistics. To reduce the computational complexity, all images are resized into the size of 224 × 224 pixels.

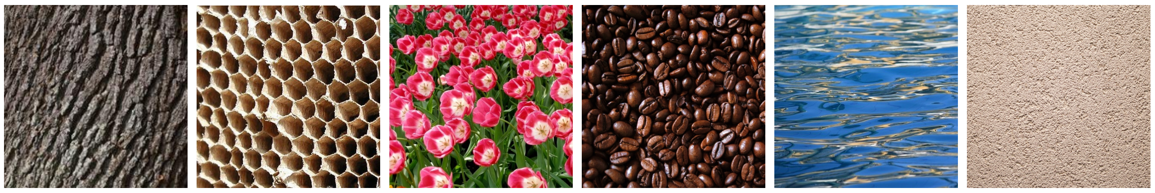

Figure 1: Six training images.

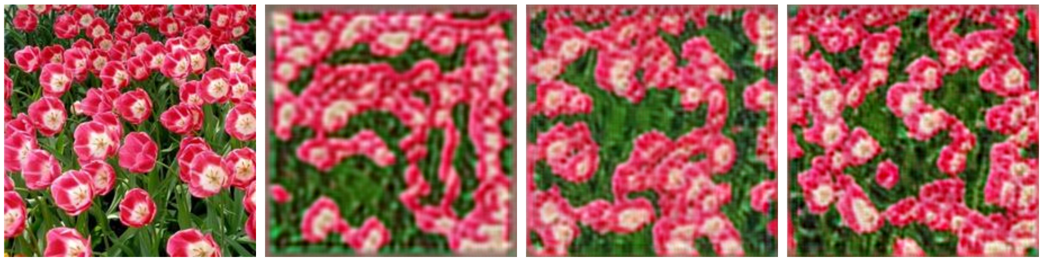

Figure 2: Synthesis example. The first image is the training image, and from left to right, the rest images are synthesized images using one layer to three layers respectively.

3 Code

The code locates at `.\release\`, and training images locates at `.\Image\`. The synthesized results will be saved at `.\release\synthesiedImage\`, and the model will be saved at `.\release\working\`.

1. `experiment_learn_frame.m` is the main entry function, which gives an example of learning and synthesizing images from the model. **Modification:** You need to write your own main function to do experiment on all six images.

2. `add_bottom_filters.m` is the function to design the structure of Deep FRAME model. **Modification:** You design your own structure of the model. The example setting may not generate vivid results. You can use more filters and larger sub-sampling size for each filter. You can also try a model with more layers to generate more realistic images. Specifically, you can modify the following parameters:

a) `num_in`: Number of filters in the previous layer.

b) `num_out`: Number of filters in the current layer.

c) `stride`: Sub-sampling size for each filter. For instance, when stride $= 2$, the convolution for each filter is computed every two pixels.

d) `filter_sz`: Filter size in pixel, and we use square filters. Note that the filter size is the size defined on feature map $F * \mathbf{I}$. Here is one example to compute the filter size in the second layer. Suppose in the first layer, `filter_sz = 5, stride = 2`, and in the second layer `filter_sz = 3`, then the filter size in the second layer can be computed by $(3-1) \times 2 + 5 = 9$

3. `train_model_generative.m` is the function to learn model and synthesize images.

4. `visualize_filters.m` is the function to visualize filters in the first layer. **Modification:** Write you own function to visualize filters in the first layer. The learned filters in the first layer is saved in `net.layers{1}.weights{1}`.

To run the code, you need firstly to run `Setup.m` to configure and compile the code. You can either use CPU or GPU version. To use GPU, set `config.gpus = [1]` in `frame_config.m`, otherwise, set `config.gpus = []`.

4 Submission

There are two tasks,

1. Show the synthesized images from each layer.
2. Display the filters in the first layer using your `visualize_filters.m`.