

## Supplementary Material: Sliced Wasserstein Distance for Learning Gaussian Mixture Models

Soheil Kolouri  
HRL Laboratories, LLC  
skolouri@hrl.com

Gustavo K. Rohde  
University of Virginia  
gustavo@virginia.edu

Heiko Hoffmann  
HRL Laboratories, LLC  
hhoffmann@hrl.com

### 7. Supplementary material

#### 7.1. Maximum log-likelihood and KL-divergence

The KL-divergence between  $I_x$  and  $I_y$  is defined as:

$$KL(I_x, I_y) = \int_{\mathbb{R}^d} I_y(\rho) \log\left(\frac{I_y(\rho)}{I_x(\rho)}\right) d\rho$$

For the maximum log-likelihood and in the limit and as the number of samples grows to infinity,  $N \rightarrow \infty$ , we have:

$$\begin{aligned} \lim_{N \rightarrow \infty} \operatorname{argmax}_{\mu_k, \Sigma_k, \alpha_k} \frac{1}{N} \sum_{n=1}^N \log(I_x(y_n)) &= \\ \operatorname{argmax}_{\mu_k, \Sigma_k, \alpha_k} \int_{\mathbb{R}^d} I_y(\rho) \log(I_x(\rho)) d\rho &= \\ \operatorname{argmin}_{\mu_k, \Sigma_k, \alpha_k} - \int_{\mathbb{R}^d} I_y(\rho) \log(I_x(\rho)) d\rho &= \\ \operatorname{argmin}_{\mu_k, \Sigma_k, \alpha_k} \int_{\mathbb{R}^d} I_y(\rho) \log(I_y(\rho)) d\rho - \\ \int_{\mathbb{R}^d} I_y(\rho) \log(I_x(\rho)) d\rho &= \\ \operatorname{argmin}_{\mu_k, \Sigma_k, \alpha_k} \int_{\mathbb{R}^d} I_y(\rho) \log\left(\frac{I_y(\rho)}{I_x(\rho)}\right) d\rho &= \\ \operatorname{argmin}_{\mu_k, \Sigma_k, \alpha_k} KL(I_x, I_y) \end{aligned}$$

#### 7.2. RMSProp update equations

SGD often suffers from oscillatory behavior across the slopes of a ravine while only making incremental progress towards the optimal point. Various momentum based methods have been introduced to adaptively change the learning rate of SGD and dampen this oscillatory behavior. In our work, we used RMSProp, introduced by Tieleman and Hinton [2], which is a momentum based technique for SGD. Let  $\alpha$  be the learning rate,  $\gamma \in (0, 1)$  be the decay parameter, and  $\kappa$  be the momentum parameter. The update equation for a GMM parameter, here  $\mu_k$ , is then calculated from:

$$\begin{cases} m_k^{(i)} = \gamma m_k^{(i-1)} + (1 - \gamma) \frac{\partial SW_p^p(I_x, I_y)}{\partial \mu_k} \\ g_k^{(i)} = \gamma g_k^{(i-1)} + (1 - \gamma) \left( \frac{\partial SW_p^p(I_x, I_y)}{\partial \mu_k} \right)^2 \\ v_k^{(i)} = \kappa v_k^{(i-1)} - \frac{\alpha}{\sqrt{g_k^{(i)} - (m_k^{(i)})^2 + \epsilon}} \frac{\partial SW_p^p(I_x, I_y)}{\partial \mu_k} \\ \mu_k^{(i)} = \mu_k^{(i-1)} + v_k^{(i)} \end{cases} \quad (1)$$

Where  $m_k$  and  $g_k$  are the first and second moments (uncentered variance) of  $\partial SW_p^p(I_x, I_y) / \partial \mu_k$ , respectively. Similar update equations hold for  $\Sigma_k$  and  $\alpha_k$ .

#### 7.3. Number of required slices

Regarding the growing number of required projections/slices, following the mini-batch learning literature, we first point out that the proposed algorithm is guaranteed to converge to a local optima for any fixed number of random projections,  $L$  (of course at different convergence rates). Here, we devise an experiment to show the convergence behavior of our algorithm for different number of random projections,  $L$ , and at different dimensions,  $d$ . To do so, we learn GMMs for data generated from mixture-models at different dimensions,  $d$ , and show the number of iterations to convergence for different  $L$ s (See Figure 7). Specifically, we generated a mixture of three Gaussian distributions with covariances equal to identity in  $d$ -dimensions where  $d \in \{2, 3, \dots, 10\}$ . At each  $d$ , we learn a GMM using our SWGMM technique with  $L = 1$ ,  $L = 5$ , and  $L = 10$  number of projections and repeat the experiment 10 times. The mean and variance of the log number of iterations needed for convergence are depicted in Figure 7.

#### 7.4. Experimental Details

Here we provide the detailed implementation and architecture of the adversarial autoencoders we used in our paper. Figure 9 shows the detailed architectures of the deep adversarial autoencoder for MNIST and CelebA datasets. The architecture of the deep binary classifiers used for scoring

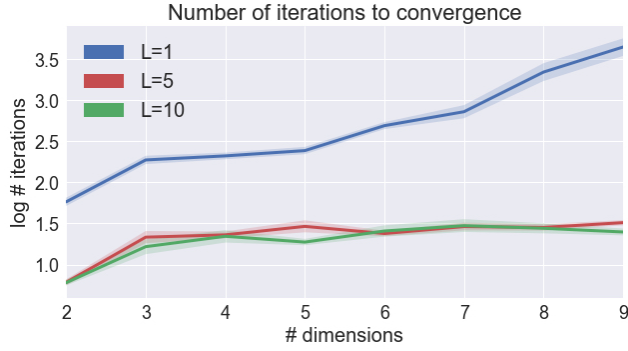


Figure 7. Log number of iterations to convergence for different number of random projections,  $L$ , and at different dimensions,  $d$ .

the fitness of the GMMs are shown in Figure 8. We used Keras [1] for implementation of our experiments.

For the loss function of the autoencoder we used the *mean absolute error* between the input image and the decoded image together with the adversarial loss of the decoded image (equally weighted). The loss functions for training the adversarial networks and the binary classifiers were chosen to be cross entropy. Finally, we used RMSProp [2] as the default optimizer for all the models, and trained the models over 100 Epochs, with batch size of 250.

### 7.5. CelebA Generated Images

Figure 10 shows all the GMM components learned by EM and our SWM formulation.

## References

- [1] F. Chollet et al. Keras. <https://github.com/fchollet/keras>, 2015. 2
- [2] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012. 1, 2

## MNIST Dataset

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 28, 28, 1)	0
conv2d_1 (Conv2D)	(None, 28, 28, 8)	80
leaky_re_lu_1 (LeakyReLU)	(None, 28, 28, 8)	0
conv2d_2 (Conv2D)	(None, 28, 28, 8)	584
leaky_re_lu_2 (LeakyReLU)	(None, 28, 28, 8)	0
max_pooling2d_1 (MaxPooling2)	(None, 14, 14, 8)	0
conv2d_3 (Conv2D)	(None, 14, 14, 16)	1168
leaky_re_lu_3 (LeakyReLU)	(None, 14, 14, 16)	0
conv2d_4 (Conv2D)	(None, 14, 14, 16)	2320
leaky_re_lu_4 (LeakyReLU)	(None, 14, 14, 16)	0
max_pooling2d_2 (MaxPooling2)	(None, 7, 7, 16)	0
conv2d_5 (Conv2D)	(None, 7, 7, 32)	4640
leaky_re_lu_5 (LeakyReLU)	(None, 7, 7, 32)	0
conv2d_6 (Conv2D)	(None, 7, 7, 32)	9248
leaky_re_lu_6 (LeakyReLU)	(None, 7, 7, 32)	0
max_pooling2d_3 (MaxPooling2)	(None, 4, 4, 32)	0
flatten_1 (Flatten)	(None, 512)	0
dense_1 (Dense)	(None, 2)	1026
Total params: 19,066		
Trainable params: 19,066		
Non-trainable params: 0		

Adversarial Network

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 28, 28, 1)	0
conv2d_7 (Conv2D)	(None, 28, 28, 8)	80
leaky_re_lu_7 (LeakyReLU)	(None, 28, 28, 8)	0
conv2d_8 (Conv2D)	(None, 28, 28, 8)	584
leaky_re_lu_8 (LeakyReLU)	(None, 28, 28, 8)	0
average_pooling2d_1 (Average)	(None, 14, 14, 8)	0
conv2d_9 (Conv2D)	(None, 14, 14, 16)	1168
leaky_re_lu_9 (LeakyReLU)	(None, 14, 14, 16)	0
conv2d_10 (Conv2D)	(None, 14, 14, 16)	2320
leaky_re_lu_10 (LeakyReLU)	(None, 14, 14, 16)	0
average_pooling2d_2 (Average)	(None, 7, 7, 16)	0
conv2d_11 (Conv2D)	(None, 7, 7, 32)	4640
leaky_re_lu_11 (LeakyReLU)	(None, 7, 7, 32)	0
conv2d_12 (Conv2D)	(None, 7, 7, 32)	9248
leaky_re_lu_12 (LeakyReLU)	(None, 7, 7, 32)	0
average_pooling2d_3 (Average)	(None, 4, 4, 32)	0
flatten_2 (Flatten)	(None, 512)	0
dense_2 (Dense)	(None, 128)	65664
Total params: 83,704		
Trainable params: 83,704		
Non-trainable params: 0		

Encoder

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 128)	0
dense_3 (Dense)	(None, 512)	66048
reshape_1 (Reshape)	(None, 4, 4, 32)	0
conv2d_13 (Conv2D)	(None, 4, 4, 32)	9248
leaky_re_lu_13 (LeakyReLU)	(None, 4, 4, 32)	0
conv2d_14 (Conv2D)	(None, 4, 4, 32)	9248
leaky_re_lu_14 (LeakyReLU)	(None, 4, 4, 32)	0
up_sampling2d_1 (UpSampling2)	(None, 8, 8, 32)	0
conv2d_15 (Conv2D)	(None, 8, 8, 32)	9248
leaky_re_lu_15 (LeakyReLU)	(None, 8, 8, 32)	0
conv2d_16 (Conv2D)	(None, 8, 8, 32)	9248
leaky_re_lu_16 (LeakyReLU)	(None, 8, 8, 32)	0
up_sampling2d_2 (UpSampling2)	(None, 16, 16, 32)	0
conv2d_17 (Conv2D)	(None, 14, 14, 16)	4624
leaky_re_lu_17 (LeakyReLU)	(None, 14, 14, 16)	0
conv2d_18 (Conv2D)	(None, 14, 14, 16)	2320
leaky_re_lu_18 (LeakyReLU)	(None, 14, 14, 16)	0
up_sampling2d_3 (UpSampling2)	(None, 28, 28, 16)	0
conv2d_19 (Conv2D)	(None, 28, 28, 1)	145
Total params: 110,129		
Trainable params: 110,129		
Non-trainable params: 0		

Decoder

## CelebA Dataset

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 64, 64, 3)	0
conv2d_1 (Conv2D)	(None, 64, 64, 16)	448
leaky_re_lu_1 (LeakyReLU)	(None, 64, 64, 16)	0
conv2d_2 (Conv2D)	(None, 64, 64, 16)	2320
leaky_re_lu_2 (LeakyReLU)	(None, 64, 64, 16)	0
max_pooling2d_1 (MaxPooling2)	(None, 32, 32, 16)	0
conv2d_3 (Conv2D)	(None, 32, 32, 32)	4640
leaky_re_lu_3 (LeakyReLU)	(None, 32, 32, 32)	0
conv2d_4 (Conv2D)	(None, 32, 32, 32)	9248
leaky_re_lu_4 (LeakyReLU)	(None, 32, 32, 32)	0
max_pooling2d_2 (MaxPooling2)	(None, 16, 16, 32)	0
conv2d_5 (Conv2D)	(None, 16, 16, 64)	18496
leaky_re_lu_5 (LeakyReLU)	(None, 16, 16, 64)	0
conv2d_6 (Conv2D)	(None, 16, 16, 64)	36928
leaky_re_lu_6 (LeakyReLU)	(None, 16, 16, 64)	0
max_pooling2d_3 (MaxPooling2)	(None, 8, 8, 64)	0
conv2d_7 (Conv2D)	(None, 8, 8, 64)	36928
leaky_re_lu_7 (LeakyReLU)	(None, 8, 8, 64)	0
conv2d_8 (Conv2D)	(None, 8, 8, 64)	36928
leaky_re_lu_8 (LeakyReLU)	(None, 8, 8, 64)	0
max_pooling2d_4 (MaxPooling2)	(None, 4, 4, 64)	0
flatten_1 (Flatten)	(None, 1024)	0
dense_1 (Dense)	(None, 2)	2050
Total params: 147,986		
Trainable params: 147,986		
Non-trainable params: 0		

Adversarial Network

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 64, 64, 3)	0
conv2d_9 (Conv2D)	(None, 64, 64, 16)	448
leaky_re_lu_9 (LeakyReLU)	(None, 64, 64, 16)	0
conv2d_10 (Conv2D)	(None, 64, 64, 16)	2320
leaky_re_lu_10 (LeakyReLU)	(None, 64, 64, 16)	0
average_pooling2d_1 (Average)	(None, 32, 32, 16)	0
conv2d_11 (Conv2D)	(None, 32, 32, 32)	4640
leaky_re_lu_11 (LeakyReLU)	(None, 32, 32, 32)	0
conv2d_12 (Conv2D)	(None, 32, 32, 32)	9248
leaky_re_lu_12 (LeakyReLU)	(None, 32, 32, 32)	0
average_pooling2d_2 (Average)	(None, 16, 16, 32)	0
conv2d_13 (Conv2D)	(None, 16, 16, 64)	18496
leaky_re_lu_13 (LeakyReLU)	(None, 16, 16, 64)	0
conv2d_14 (Conv2D)	(None, 16, 16, 64)	36928
leaky_re_lu_14 (LeakyReLU)	(None, 16, 16, 64)	0
average_pooling2d_3 (Average)	(None, 8, 8, 64)	0
conv2d_15 (Conv2D)	(None, 8, 8, 64)	36928
leaky_re_lu_15 (LeakyReLU)	(None, 8, 8, 64)	0
conv2d_16 (Conv2D)	(None, 8, 8, 64)	36928
leaky_re_lu_16 (LeakyReLU)	(None, 8, 8, 64)	0
average_pooling2d_4 (Average)	(None, 4, 4, 64)	0
flatten_2 (Flatten)	(None, 1024)	0
dense_2 (Dense)	(None, 256)	262400
Total params: 408,336		
Trainable params: 408,336		
Non-trainable params: 0		

Encoder

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 256)	0
dense_3 (Dense)	(None, 1024)	263168
leaky_re_lu_17 (LeakyReLU)	(None, 1024)	0
reshape_1 (Reshape)	(None, 4, 4, 64)	0
conv2d_17 (Conv2D)	(None, 4, 4, 64)	36928
leaky_re_lu_18 (LeakyReLU)	(None, 4, 4, 64)	0
conv2d_18 (Conv2D)	(None, 4, 4, 64)	36928
leaky_re_lu_19 (LeakyReLU)	(None, 4, 4, 64)	0
up_sampling2d_1 (UpSampling2)	(None, 8, 8, 64)	0
conv2d_19 (Conv2D)	(None, 8, 8, 64)	36928
leaky_re_lu_20 (LeakyReLU)	(None, 8, 8, 64)	0
conv2d_20 (Conv2D)	(None, 8, 8, 64)	36928
leaky_re_lu_21 (LeakyReLU)	(None, 8, 8, 64)	0
up_sampling2d_2 (UpSampling2)	(None, 16, 16, 64)	0
conv2d_21 (Conv2D)	(None, 16, 16, 32)	18464
leaky_re_lu_22 (LeakyReLU)	(None, 16, 16, 32)	0
conv2d_22 (Conv2D)	(None, 16, 16, 32)	9248
leaky_re_lu_23 (LeakyReLU)	(None, 16, 16, 32)	0
up_sampling2d_3 (UpSampling2)	(None, 32, 32, 32)	0
conv2d_23 (Conv2D)	(None, 32, 32, 16)	4624
leaky_re_lu_24 (LeakyReLU)	(None, 32, 32, 16)	0
conv2d_24 (Conv2D)	(None, 32, 32, 16)	2320
leaky_re_lu_25 (LeakyReLU)	(None, 32, 32, 16)	0
up_sampling2d_4 (UpSampling2)	(None, 64, 64, 16)	0
conv2d_25 (Conv2D)	(None, 64, 64, 3)	435
Total params: 445,971		
Trainable params: 445,971		
Non-trainable params: 0		

Decoder

Figure 8. Details of the convolutional autoencoders learned for the MNIST and CelebA face dataset



### Adversarial Networks for Quality Assessment

Layer (type)	Output Shape	Param #	Layer (type)	Output Shape	Param #	Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 128)	0	input_1 (InputLayer)	(None, 256)	0	input_1 (InputLayer)	(None, 2)	0
dense_1 (Dense)	(None, 256)	33024	dense_1 (Dense)	(None, 512)	131584	dense_1 (Dense)	(None, 8)	24
dense_2 (Dense)	(None, 512)	131584	dense_2 (Dense)	(None, 1024)	525312	dense_2 (Dense)	(None, 16)	144
dense_3 (Dense)	(None, 256)	131328	dense_3 (Dense)	(None, 512)	524800	dense_3 (Dense)	(None, 8)	136
dense_4 (Dense)	(None, 2)	514	dense_4 (Dense)	(None, 2)	1026	dense_4 (Dense)	(None, 2)	18
Total params: 296,450			Total params: 1,182,722			Total params: 322		
Trainable params: 296,450			Trainable params: 1,182,722			Trainable params: 322		
Non-trainable params: 0			Non-trainable params: 0			Non-trainable params: 0		
MNIST Dataset			CelebA Dataset			Ring-Square-Line Dataset		

Figure 9. Details of the deep binary classifiers used for scoring the fitness of GMMs.



(a)



(b)

Figure 10. GMM Samples Generated from the GMM learned from EM-GMM (a), and from SW-GMM (b). Each column depicts random samples from a single component of the GMM.