A Performance and Quality Comparison of Conventional and Deep Learning Image Super-Resolution Techniques

Srinivas Kaza
Massachusetts Institute of Technology
77 Massachusetts Ave, Cambridge, MA 02139
kaza@mit.edu

Nikhil Murthy
Massachusetts Institute of Technology
77 Massachusetts Ave, Cambridge, MA 02139
nmurthy@mit.edu

Abstract

We evaluate two recent single-image super-resolution (SISR) approaches on a basis of performance and image quality. The first technique is the Single-Image Super-Resolution Using a Generative Adversarial Network (SRGAN) approach by Ledig et al. [12], a deep learning method leveraging generative adversarial networks (GANs). The second technique is Google's recent Rapid and Accurate Image Super Resolution (RAISR) approach by Romano et al. [14], a "hashing-based" machine-learning approach. Our motivation behind comparing these two approaches is to evaluate qualitative and quantitative differences between deep-learning and more conventional approaches to super-resolution. For SRGAN, we investigated the addition of class information as a condition for both the generator and discriminator, which improved performance. With respect to RAISR, we focused on several implementation modifications that made the algorithm more GPU-amenable, in an effort to improve performance without sacrificing image quality. Ultimately, we conclude that deep learning approaches to SISR work best with large scaling factors and when there is an abundance of training data, while RAISR is more effective for smaller scaling factors on larger images with smaller training datasets.

1. Introduction

In recent years, the problem of Single Image Super-Resolution (SISR) has received significant interest in machine learning community. The vast majority of the approaches proposed have been deep-learning based. In fact, 30 out of the 31 teams competing in NTIRE 2018 Challenge SISR [17] incorporated a deep learning approach. We want to evaluate both a deep learning and conventional machine learning approach to the SISR problem on a basis of performance and image quality. We also attempted a handful of experiments which aimed to improve upon the practical efficacy of these methods.

SRGAN, a deep-learning approach to SISR, uses the framework of adversarial training to generate super-resolution images. A generator network, G_{θ_G}, takes as input low-resolution images, I^{LR} with dimensions $W \times H \times C$, and outputs a "fake" high-resolution image $G_{\theta_G}(I^{LR})$ with dimensions $rW \times rH \times C$, where r is the scaling factor. A discriminator network D_{θ_D} is tasked with differentiating between the true high resolution images I^{SR} and the generated images $G_{\theta_G}(I^{LR})$. Following Goodfellow et al. [5], the generator and discriminator networks can be optimized by solving the following min-max problem:

$$\min_{\theta_G} \max_{\theta_D} \mathbb{E}_{I^{HR}\ p_{train}(I^{HR})}[\log D_{\theta_D}(I^{HR})] + \mathbb{E}_{I^{LR}\ p_G(I^{LR})}[\log(1 - D_{\theta_D}(G_{\theta_G}(I^{LR})))]$$

The GAN framework and training procedure encourages the reconstructed high-resolution images to move towards regions of the search space with high probability of containing photo-realistic images. An adversarial loss function pushes a solution to the natural image manifold using the discriminator network, while a content loss function encourages perceptual similarity instead of similarity in pixel space. SRGAN set the new state of the art on public benchmarking datasets for the problem of SISR.

The non-deep-learning approach was Google's RAISR technique. Most deep neural nets work by applying a series of filters to an image, followed by an activation function, followed by another series of filters. Instead of applying a fixed set of filters to every patch within an image, RAISR applies a "hash" function to each patch, the output of which (i.e the "hash key") is used to select the correct filter to apply. During the learning phase, the algorithm hashes many patches in the training set, and learns a mapping from low-resolution (LR) to high-resolution (HR) patches for every bucket in the hash table. During the inference phase, one merely needs to run the hashing function on a given patch, look up the correct filter in the hash table, and apply it on the patch.

RAISR does not compare favorably to modern deep learning approaches on a basis of image quality. Given that the algorithm only applies a single filter to an image, it cannot compete with a deep network that applies thousands of different filters. However, the simplicity of the hashing and filtering approach used in RAISR ensures that its run-time is several orders of magnitude lower than most modern deep learning approaches. This property makes RAISR a suitable option for upsampling large images with small, integer scale factors.

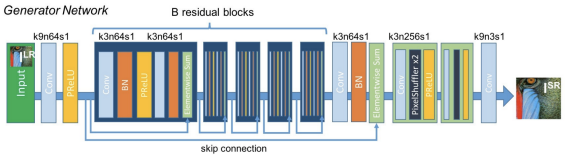
2. Approach: SRGAN

The SRGAN super-resolution algorithm and our experiments will be described in this section. We will first describe the SRGAN architecture and our implementation, and will then explain SRGAN-labels, our proposed model that incorporates class information to better generate high-resolution images.

Our implementation of SRGAN and SRGAN-labels is written in PyTorch and optimized for multi-GPU training.

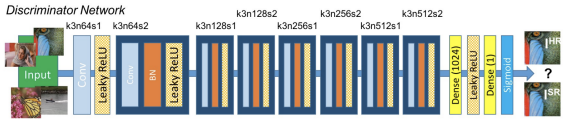
2.1. Architecture: SRGAN

SRGAN is made up of a Generator Network G and a Discriminator Network D. The Generator consists of B residual blocks with an identical layout (as inspired by Johnson et al. [9]), with each block containing two convolutional layers with 3×3 kernels and 64 feature maps followed by batch-normalization layers [8] and ParametricReLU [7] as the activation function. As proposed by Shi et al. [15], to increase resolution of the input image, two sub-pixel convolutional layers are used:



Architecture of Generator Network (from [12])

The Discriminator Network is used to discriminate real high-resolution images (I^{HR}) from generated high-resolution images ($G(I^{LR})$). The network contains eight convolutional layers, each with an increasing number of 3×3 filters, similar to the VGG network [16]. LeakyReLU ($\alpha = 0.2$) is used throughout along with strided convolutions to reduce the image size (when features are doubled). A final sigmoid activation function produces a probability of the input image being generated or real:



Architecture of Discriminator Network (from [12])

2.2. Architecture: SRGAN-labels

SRGAN-labels uses the SRGAN architecture but adds class information as another condition for both the generator and discriminator.

An input low-resolution image I^{LR} to the Generator G is usually of dimension $W \times H \times 3$, where each image has 3 channels: Red, Green and Blue. We propose adding another channel M_G of dimension $W \times H$ to each low-resolution image that contains label information. Specifically, let there be k classes in a given labeled dataset. For each image I_i^{LR} with class c_i, we set all values in M_G to 0 except for the pixel at position $(c_i//W) \times (c_i \mod W)$, which we set to 1. Similarly, for the Discriminator D, we add a similar channel M_D which is of size $rW \times rH$, where r is the scaling factor, and is defined analogously as M_G.

Inspired by Mirza and Osindero [13], the added label information acts as a condition for both the generator and discriminator, which can aid the generation process while also allowing the discriminator to discretize its search space.

2.3. Learning

To evaluate the performance of the Generator Network, we use the following perceptual loss function, which is a weighted sum of a content loss (l_X^{SR}) and an adversarial loss (l_{Gen}^{SR}):

$$l^{SR} = l_X^{SR} + 10^{-3} l_{Gen}^{SR}$$

where the content loss is based on VGG feature maps [12]:

$$l_{VGG/i,j}^{SR} = \frac{1}{W_{i,j} H_{i,j}} \sum_{x=1}^{W_{i,j}} \sum_{y=1}^{H_{i,j}} (\phi_{i,j}(I^{HR})_{x,y} - \phi_{i,j}(G_{\theta_G}(I^{LR}))_{x,y})^2$$

and the adversarial loss is defined based on the probabilities of the discriminator over all training examples:

$$l_{Gen}^{SR} = \sum_{n=1}^{N} -\log D_{\theta_D}(G_{\theta_G}(I^{LR}))$$

3. Approach: RAISR

The RAISR super-resolution algorithm, as well as some of our experiments, will be described in this section. To begin with, we describe the learning process used in RAISR, and how it interfaces with the rest of the method. We will then describe the hashing algorithm as well as the filtering process. We did not implement Google's post-processing operations that avoid oversharpening, although that could be an interesting direction for future work.

We wrote an implementation of both the training and the inference stages of the algorithm in Rust. We also were working on a GPU implementation of the inference step in CUDA that was abandoned for a currently-incomplete OpenGL compute shader implementation.

3.1. Learning

RAISR attempts to learn a mapping from a set of low-resolution patches to a set of corresponding high-resolution patches. It does so by first applying a "cheap" upsampling method to the low-resolution patches. Google's implementation used a bicubic filter, but we had no difficulties replicating similar results with a bilinear filter instead. The importance of the initial upsampling step is to properly handle the low-frequency parts of the image; for these components, the choice of upsampling technique is not particularly important assuming it does not oversharpen. The RAISR paper then uses the following form to "learn" a filter to apply to a given patch:

$$\sum_{i=0}^{L} ||A_i h - b_i||_2^2$$

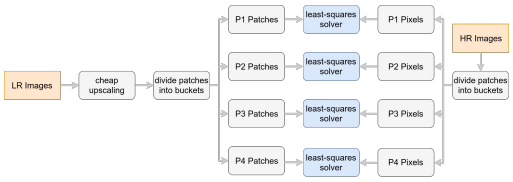
.

where A_i is a $[MN\ d^2]$ matrix storing patches at every location in the image, b_i is the center HR pixel at each patch location, h is the filter to be learned, M and N are the width and height of the image, and d is the width of the filter/patch (note that the total filter/patch size is d^2). In the original paper, d is 11, and we noticed the best results with this filter size. The authors of RAISR note that you can multiply both sides of the equation by A^T. If we define $Q = A^T A$ and $V = A^T b$, then we obtain:

$$||Qh - V||_2^2$$

.

Note that Q is only $[d^2\ d^2]$ rather than $[MN\ d^2]$, and likewise V is much smaller than b. This is a useful property, and we also benefit from the fact that we can build up Q incrementally. We solve this matrix equation easily with a conjugate gradient least squares solver.



Architecture of RAISR's learning process

3.2. Hashing

RAISR uses a "hashing" algorithm to sort patches into bins. The hash key has four primary components – angle, strength, coherence, and pixel type. The last parameter is merely a parameter that changes by offset into the underlying interpolation technique. The authors of the RAISR paper noted that the cheap interpolation is not shift invariant, and given that we are attempting to apply a single filter to the output of this interpolation technique, we need to incorporate the offset into the hash key to compensate for the aliasing.

The first three parameters can be computed from a patch of the image gradient centered around the pixel in question. In the original implementation, the gradient size was (conveniently) chosen to be $9x9$, although the technique used to compute the gradient was not specified. We had no issues using a Sobel filter. The paper then describes a simple method to weight the gradient and construct a useful closed form. They unravel the gradients into a matrix that contains two column vectors for each gradient – G, and then compute

$$G^T W G$$

where W is a diagonal Gaussian weighting matrix. A 2x2 SVD is then performed. The angle parameter is the angle of the eigenvector corresponding to the largest singular value. The strength is the larger singular value, and the coherence comes from both eigenvalues.

3.3. Inference

At this point, one merely needs to apply the filter from the correct hash bucket learned during the learning stage. See the Filterbank Compression section for more details about how this stage was implemented.

4. Implementation Challenges

There were many implementation pitfalls that we came across while implementing both of these algorithms.

4.1. SRGAN: Training Time

As with many deep learning models, training and hyperparameter search time can often be a limiting constraint. Considering the large dataset sizes, the number of parameters and the added complexity of adversarial training, training and testing both SRGAN and the SRGAN-labels variant proved to be no exception.

To combat the very large training time initially, we added CUDA support in our SRGAN implementation, which considerably sped up training. Additionally, we added multi-GPU support by parallelizing computations across multiple GPUs over the batch dimension by using PyTorch's DataParallel module, along with its MPI-like primitives. The combination of these two improvements helped ameliorate many of the challenges we were having with respect to training time.

However, to further aid training time, we decided to start by training our models on CIFAR10 [10], a labeled image dataset that consists of 60000 32×32 color images in 10 classes, with 6000 images per class. By training Generator Networks to scale 16×16 images to generated 32×32 sized

images, the problem was much more manageable, allowing us to get results much faster.

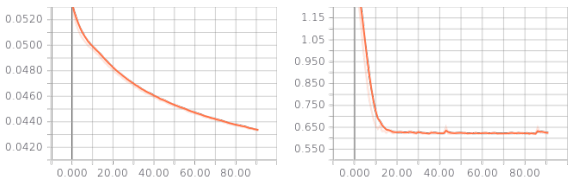
4.2. SRGAN: Early Stop in Learning

One challenge that we frequently ran into when training our Generative Adversarial Networks was a stop in learning early on in training, as evidenced by the Discriminator Network loss approaching 0.

The first technique employed to address the early stopping in learning of both networks was to add a pre-training stage in which the Generator Network was trained solely on the Mean-Squared Error (MSE) loss. Because the Discriminator Network's loss was approaching 0 very early on in training, we noticed that the Discriminator was very easily able to differentiate between the real and fake distributions of high-resolution images. Thus, the Generator was being outpaced by the Discriminator when adversarial training, and needed a head-start to compete. Therefore, this pre-training stage, which occurred for only two epochs, allowed the Generator to at least generate semi-realistic images before adding the Discriminator to the training process. Empirically, the pre-training made a large difference in preventing early stopping in learning.

In addition, we also noticed that the hard labels of the Discriminator Network, 1 if the input image is real and 0 if the input image is generated, prevented gradient flow through the network and also hampered training. Instead, we replaced these hard labels with soft labels that added a small amount of randomness, where a random number between $(0, 0.3)$ is used as the soft label if the input image is real and random number between $(0.7, 1)$ is used if the input image is generated.

The loss curves for both the Generator and Discriminator Networks is shown below, after addressing the issues mentioned above:



Generator (left) and Discriminator (right) Loss Curves trained on CIFAR10 (SRGAN)

4.3. SRGAN: Hyperparameter Search

Finally, we went through an extensive process of hyperparameter search in order to fully optimize our models and maximize performance. Some of the hyperparameters that were tested include: generator pre-training learning rate, generator learning rate, discriminator learning rate, number of residual blocks in the generator, activation function (sigmoid, swish, ReLU, ParametricReLU), number of epochs for pre-training, number of epochs for training and batch size.

4.4. RAISR: Filterbank Storage

RAISR has a unique number of performance challenges during the filtering stage due to the scatter-gather nature of the algorithm. The deep learning approaches apply the same filter for every pixel. RAISR uses a different pixel for each pixel, and the filter to use is determined at runtime. Thus, RAISR accesses the filterbank in a fairly random pattern. However, the question arises – how does one store and access the filters?

We found Intel Nervana's reverse-engineered SGEMM writeup [1] insightful, as well as other SGEMM implementations [11]. Our original CUDA implementation simply stored and fetched the filters in global memory. Performance was poor, because the device was unable to coalesce the reads into the filterbank. Constant memory is not an option for this application for several reasons – not only is the filterbank slightly larger than the total size of constant memory (even after size optimizations), the random access reads will all be serialized. Texture memory is another possibility, and given the size of the texture/L1 cache on modern GPU architectures, it is possible the majority of the filterbank will remain resident in the cache during the entirety of the filtering step. Alternatively, we could swap out chunks of the filterbank in phases to minimize thrashing. Our OpenGL implementation stores the filterbank in a buffer texture, which seems like a promising start. Notably, Google's followup paper to RAISR, BLADE [4], claims that the filterbank was accessed as an RGBA texture (on Android).

Shared memory is yet another option for filterbank storage. With some care to avoid alignment issues, one could load large chunks of the filterbank, stored in constant (or global) memory, into shared memory. This step would be performed across the entirety of the block. One might wish to use global memory for this purpose due to relatively small size (˜8k) of the multi-processor constant memory cache. Then each invocation would index into shared memory rather than global memory, and benefit from the much better shared-memory latency. There are several problems with this approach. To begin with, it is not immediately obvious that the performance improvement from using shared memory would offset the penalty of a large block-wide load. However, one could contend that cuBLAS and cuDNN load model parameters in a similar fashion, as Intel Nervana's SGEMM writeup would suggest [1]. The more pressing issue is the problem of shared memory bank conflicts.

Most algorithms that merely need to access sequential elements of shared memory need not worry about bank conflicts, because subsequent entries of shared memory are located on different memory banks. However, RAISR accesses random offsets into the filterbank. We can estimate

the number of bank conflicts that we are likely to encounter by calculating the expected value of bank collisions. In essence, we wish to compute the number of invocations within a warp that happen to access the same bank. This problem is just a small modification of the famous birthday problem – rather than just calculating the number of people required to have a likely birthday collision, we need to calculate the expected number of birthday collisions overall. Given n lookups and m banks –

$$E = n - m(1 - \frac{m-1}{m}^{n})$$

.

In our case, the expected number of collisions overall evaluates to ˜11.586. A more telling statistic is the expected *maximum* number of collisions for any given entry in a hash table. The entry in the table with the most collisions (i.e the most accessed bank) will introduce the bottleneck. This value comes out to ˜3.539. A 4-way bank conflict is not ideal, but also not unreasonable.

4.5. RAISR: Filterbank Compression

We had several ideas regarding filterbank compression. Model compression is not a new concept in machine learning [6], and we decided to try several experiments of our own. Our implementation of RAISR has 24 angle buckets, 3 strength buckets, 3 coherence buckets (these were the suggested hyperparameters in the original RAISR paper). For a 2x upsample are also 4 more buckets for the pixel type. Each filter is 11x11 and the filter datatype is float32. Thus, the overall size of the filterbank is 418KB.

Our first, and most successful, experiment was to store an additional texture containing a set of minimum and maximum bounds computed on a per-filter basis and then quantize the filters to uint8. This change trades off the compute resources required to rescale the filter with a 73% memory bandwidth saving. We noticed no statistically significant change in SSIM as a result of implementing this optimization.

Another idea was to represent the filters in the frequency domain and then use a quantization matrix for compression. This idea was vaguely inspired by JPEG; however, the run-length encoding and Huffman steps are omitted to avoid latency issues. We can even modify the learning stage of the algorithm accordingly.

$$\min ||Q \left\lfloor S \odot D^T h D \right\rfloor - V||_2^2$$

where D is the DCT matrix and S is the (scaled) quantization matrix. The spectral coefficients would need 11 bits to represent, but most of them would require only a couple of bits after quantization. There was some initial excitement about this approach because it had the potential to halve the filterbank size. Additionally, the DFT of the filters seemed to suggest that they had few high frequency components. Unfortunately even minor amounts of quantization introduced unacceptable amounts of noise into results from transfer learning, and artificial amounts of smoothing into the learning process.

5. Results

We evaluated SRGAN, SRGAN-labels and RAISR on the CIFAR10 dataset [10]. SRGAN and SRGAN-labels were trained on Amazon AWS on an NVIDIA K80 GPU. The following SSIMs were achieved for each model:

Model:	SSIM
SRGAN	0.9474
SRGAN-labels	**0.9508**
RAISR	0.8609

Visual results for SISR on CIFAR10 as an illustration of each method's performance are shown below (it is worth noting that RAISR was not trained on CIFAR10, and thus theses results are not completely representative of RAISR's potential performance):

We also evaluated RAISR on the validation data in track 1 of the DIV2K dataset [3], and achieved an SSIM on 0.8601 using the Pytorch SSIM evaluation tool used for the previous section. Unfortunately, we did not have an opportunity to train RAISR on the DIV2K train dataset, so these results are not completely representative of RAISR's potential performance.

Regarding performance, we evaluated RAISR's hashing performance on a GTX 1070 in Table 1. Note that these benchmarks were taken using the older CUDA code, which had an inefficient filtering mechanism, rather than the newer OpenGL compute shader. These timings were computed by running the kernel 100 times in a row; with the appropriate CPU-GPU sync points before and after execution. We also collected the combined inference (i.e hashing and filtering) in Table 2.

	Time (ms)	Megapixels/second
3480x2160	5.15	1459.57
3200x1800	3.63	1585.78

Table 1. 2x hashing performance results

	Time (ms)	Megapixels/second
3480x2160	41.04	1831.58
3200x1800	36.30	1576.78

Table 2. 2x inference (hashing and filtering) performance results

As mentioned earlier, the filtering performance is currently quite poor. Some of the filterbank storage and compression techniques may improve these results.

6. Conclusion and Future Work

We considered the relative performance and accuracy of SRGAN and SRGAN-labels, a GAN architecture for super-resolution tasks, and RAISR on the CIFAR10 and DIV2K datasets. We conclude that the deep learning approaches work exceptionally well with high scaling factors and low-resolution input, where each pixel covers a large area in the represented space. The difference is less noticeable for larger images, or images where each pixel represents a small region in world space. For these images, less computationally-intensive approaches to single-image super-resolution work well.

If we had more time to continue research, we would implement several strategies for filterbank compression, including loading the filterbank into shared memory. Additionally, we would explore better incorporating label information into our deep learning model. Given the kinds of applications that benefit from image super-resolution, we believe that both deep learning and conventional learning approaches have a compelling use-case.

7. Individual Contributions: Srinivas

I worked on everything related to RAISR, including the performance and image quality evaluation. I also have an open source Rust implementation (CPU and GPU) at [2]. The old GPU implementation that was used for benchmarking here is not currently open source; I can email the source if necessary.

References

[1] Intel nervana maxas sgemm. `https://github.com/NervanaSystems/maxas/wiki/SGEMM`. Accessed: 2018-12-09.

[2] rusty-raisr. `https://github.com/animatedrng/rusty-raisr`. Accessed: 2018-12-12.

[3] E. Agustsson and R. Timofte. Ntire 2017 challenge on single image super-resolution: Dataset and study. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017.

[4] P. Getreuer, I. Garcia-Dorado, J. Isidoro, S. Choi, F. Ong, and P. Milanfar. Blade: Filter learning for general purpose computational photography. In *2018 IEEE International Conference on Computational Photography (ICCP)*, pages 1–11, May 2018.

[5] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[6] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015.

[7] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[8] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[9] J. Johnson, A. Alahi, and L. Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In B. Leibe, J. Matas, N. Sebe, and M. Welling, editors, *Computer Vision – ECCV 2016*, pages 694–711, Cham, 2016. Springer International Publishing.

[10] A. Krizhevsky and G. Hinton. Convolutional deep belief networks on cifar-10. *Unpublished manuscript*, 40(7), 2010.

[11] A. Lavin. maxdnn: An efficient convolution kernel for deep learning with maxwell gpus. *CoRR*, abs/1501.06633, 2015.

[12] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi. Photo-realistic single image super-resolution using a generative adversarial network. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 105–114, July 2017.

[13] M. Mirza and S. Osindero. Conditional generative adversarial nets. *CoRR*, abs/1411.1784, 2014.

[14] Y. Romano, J. Isidoro, and P. Milanfar. Raisr: Rapid and accurate image super resolution. *IEEE Transactions on Computational Imaging*, 3(1):110–125, March 2017.

[15] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. *CoRR*, abs/1609.05158, 2016.

[16] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[17] R. Timofte, S. Gu, J. Wu, and L. Van Gool. Ntire 2018 challenge on single image super-resolution: Methods and results. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2018.