DOME: Supplementary Material

March 2022

1 Network Architecture and Training Details

1.1 Segmentation Networks Implementation Details

All three of our image-conditioned object segmentation networks are inspired by the U-Net [9] architecture. The *Concatenation* network is an encoder-decoder with a U-Net structure (which we will refer to as the *U-Net encoder-decoder* for simplicity), where the bottleneck and the live images are simply concatenated channel-wise before being processed by the network. The *Tiling* network uses the U-Net encoder-decoder to process the live image, and at each feature level of the encoder (including the input), a 256-dimensional conditioning vector is tiled (meaning repeated at each pixel), and concatenated channel-wise to the feature map. This 256-dimensional conditioning vector is the output of a *conditioning network* that has the bottleneck image as its input. In the *FiLM* network the U-Net encoder-decoder receives the live image as its input. Then, its decoder feature maps are modulated by FiLM layers. These FilM layers use modulating parameters that are outputs of a *conditioning network* with multiple heads that uses the bottleneck image as its input. All our models share a lot of their architecture, as is illustrated in Fig. 1. As such, in the following we describe details of our U-Net encoder-decoder and the conditioning network architectures (for Tiling and FiLM) that are at the core of our models.

1.1.1 U-Net Encoder-Decoder Implementation Details

Our U-Net encoder-decoder architecture follows the U-Net structure [9], meaning that the encoder down-samples the image to feature maps of lower and lower resolutions using convolutional layers, and the decoder up-samples those features back to the original resolution. At each resolution level, the encoder features are concatentated back to the decoder features.

Our encoder consists of a succession of convolutional layers, with (1) $[64, 128, 256, 512]$ filters, (2) kernel size (3×3), (3) stride 2, (4) Instance Normalisation (*InstanceNorm*) [13], (5) ReLU activations [7], (6) Dropout [10] with probability 0.25, and (7) padding such as to half the image resolution at each layer.

Our decoder consists of a series of bilinear upsampling layers that double the resolution of the feature maps followed by convolutional layers that process

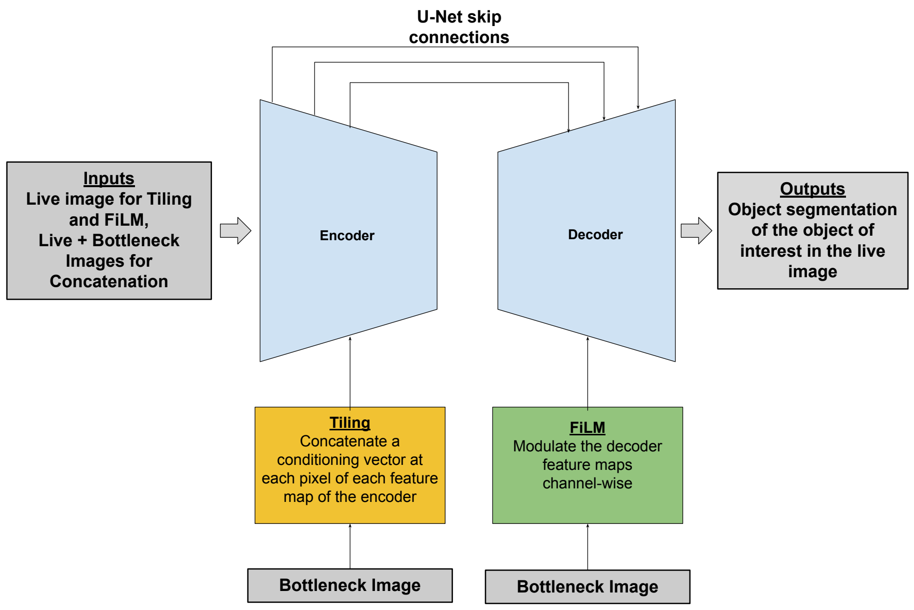
U-Net skip connections
Inputs
Live image for Tiling and FiLM,
Live + Bottleneck Images for Concatenation
Encoder
Decoder
Outputs
Object segmentation of the object of interest in the live image
Tiling
Concatenate a conditioning vector at each pixel of each feature map of the encoder
FiLM
Modulate the decoder feature maps channel-wise
Bottleneck Image
Bottleneck Image

Figure 1: Illustration of our Segmentation network architectures.

them. Similarly to U-Net [9], feature maps of the decoder are concatenated with feature maps of the encoder of the same resolution. The convolutional layers that are used after up-sampling have (1) $[512, 256, 128, 128]$ filters, (2) kernel size (3×3), (3) stride 1, (4) Instance Normalisation ($InstanceNorm$) [13], (5) ReLU activations [7], (6) Dropout [10] with probability 0.25, and (7) padding such as to keep the resolution the same. At the full resolution, we also apply 3 post-up-sampling convolutions with 64 filters, kernel size of 3, stride 1, ReLU activations, InstanceNorm, Dropout of 0.25, and padding to keep the resolution unchanged. Finally, we apply a 1×1 convolution that outputs a single channel, which is passed through a Sigmoid [6] activation to give the final output.

For our FiLM model, the features are also passed through a *FiLM block* at each level of the decoder. Similarly to [8], our FiLM block consists of a residual block with two convolutions of kernel size 3, stride 1 and InstanceNorm, and a FiLM layer, as is depicted in Fig. 2. The FiLM layer modulates its input feature maps channel-wise through an affine transformation. So, for the channel c of feature map X, X_c, we get that the FilM layer outputs $X_c \times \gamma_c + \beta_c$, where γ_c and β_c are the c'th dimensions of the parameter vectors γ and β. For full details on FiLM, we refer the reader to [8]. We describe how we obtain the β and γ parameter vectors required by the FiLM layer in Section 1.1.2.

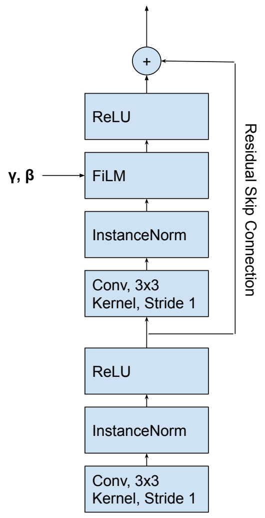

Figure 2: Our FiLM block implementation, inspired by [8].

1.1.2 Conditioning Network Implementation Details

Our conditioning network processes the bottleneck image in the Tiling and FiLM architectures. It has a convolutional neural network (CNN) architecture that is composed of a *CNN encoder* and one or more *CNN heads*. For the Tiling model, we use a CNN encoder followed by a single CNN head that is used in order to generate a 256-dimensional conditioning vector. For the FiLM model, we use a CNN encoder followed by multiple CNN heads, one for each of the feature maps that FiLM layers modulate.

The CNN encoder is made of a series of convolutional layers with (1) $[64, 128, 256, 256, 128]$ filters, (2) kernel size (3×3), (3) stride 2, (4) Instance Normalisation (*InstanceNorm*) [13], (5) ReLU activations [7], (6) Dropout [10] with probability 0.25, and (7) padding such as to half the image resolution at each layer. The output of the CNN encoder is then flattened, and passed to one of more CNN heads.

A CNN head is made of a succession of fully connected layers with (1) $[512, 256, 256, 256]$ units, (2) Instance Normalisation (*InstanceNorm*) [13], (3) ReLU activations [7] in all but the last layer, and (4) Dropout [10] with probability 0.25. For the FiLM architecture for each head there is also an additional layer that changes the dimensionality of the output as appropriate to modulate corresponding decoder feature map.

1.1.3 Training Details

To train our models, we use the $\alpha-$balanced variant of the Focal Loss [5]. We use $\gamma = 2$ for its focusing parameter, and set the weighting factor α using the class frequencies in our training data. We refer the reader to [5] for full details on the focal loss. For training, we use a batch size of 32, the Adam optimiser, a learning rate of 5×10^{-4}, and a held-out validation set that consists of 5% of our training data. We also have a learning rate scheduler that reduces the learning rate by a factor of 0.75 each time the performance of our models on the held-out validation set stagnates.

1.2 Learned Visual Servoing Network

1.2.1 Network Architecture

Our visual serving network is tasked with taking in the segmented live and bottleneck images in order to output e_{xy}, e_s and e_r. In practice, we implemented this with 3 independent Siamese CNNs [16], each outputting e_{xy}, e_s, and e_r, respectively. All three take in the same bottleneck and live images, and their architecture only differs in the number of outputs they have: 2 for predicting e_{xy}, 1 for predicting e_s, and 2 for predicting e_r. We note that the angle e_r requires 2 outputs because we use a sine-cosine encoding of the actual angle.

With the Siamese CNN architecture, the segmented live and bottleck images are first processed using convolutional encoders with shared weights. The resulting features are then flattened and concatented, before being processed by an series of dense layers to obtain the final output [16]. In our models, the convolutional encoder consists of a series of convolutional layers with (1) $[256, 256, 256, 256, 256]$ filters, (2) kernel size (3×3), (3) stride 2, (4) Instance Normalisation (*InstanceNorm*) [13], (5) ReLU activations [7], (6) Dropout [10] with probability 0.25, and (7) padding such as to half the image resolution at each layer. The dense layers that process the concatenated image features have (1) $[512, 256, 256, 128, o]$ units, with o being then number of outputs discussed above, (2) Instance Normalisation (*InstanceNorm*) [13], (3) ReLU activations [7], and (4) Dropout [10] with probability 0.25. We note that the final layer giving the output has no activation function, and is hence simply a dense linear layer. This architecture is illustrated in Fig. 3

1.2.2 Training Details

To train our models, we use the ground truth segmentation masks to create the inputs, and use the mean squared error on our prediction labels (see Section 2.1) as our loss. We use a batch size of 16, the Adam optimiser, a learning rate of 5×10^{-4}, and a held-out validation set that consists of 5% of our training data. We also have a learning rate scheduler that reduces the learning rate by a factor of 0.75 if the performance of our models on the held-out validation set stagnates.

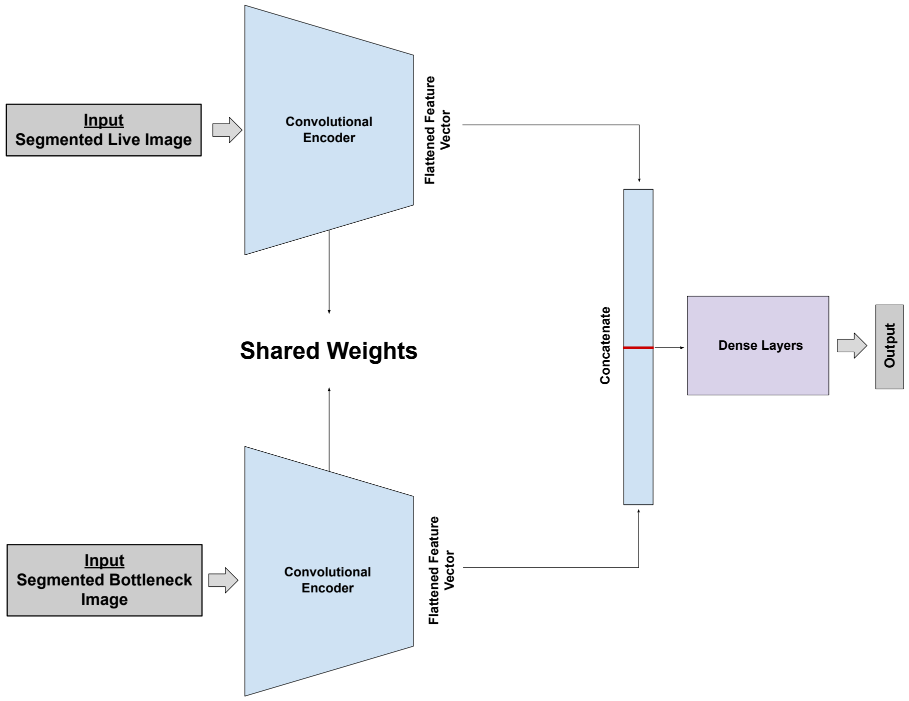

Figure 3: Illustration of our Siamese CNN architecture. Output is one of $\{e_{xy}, e_r, e_s\}$.

2 Dataset Generation Details

2.1 Dataset Generation Process

We use Blender [3] and Blenderproc [4] in order to create a simulation environment for generating our data. Our environment is made of a *room* with 6 planes as walls. The *floor* is where we place our objects, and the *ceiling* is made to emit light which serves as ambient light to our environment.

In order to create our data, a random object from either Shapenet [2] or ModelNet40 [15] is first placed at centre of the floor. A camera position is then uniformly sampled in the ranges $\{[-0.05, 0.05], [-0.05, 0.05], [0.12, 0.25]\}$ such that the camera is looking straight downwards with a random orientation around the vertical axis, and a bottleneck image is rendered from the camera. This bottleneck image is considered valid if the object is fully visible on it, and we repeat the camera sampling process until we get a valid bottleneck image. When a valid bottleneck image is obtained, it is stored along with a segmentation mask of the object (the bottleneck segmentation) and the process continues.

Second, we take four live images. In order to do so, we randomly place between 0 and 25 distractor objects in the room, and render images from 4 new camera poses. These poses are obtained by translating and rotating the camera

	Randomised Simulation Aspects
Light Parameters	ambient light strength, point lightsource number and location, point lightsource strength, point lightsource colour
Colour and texture parameters	surface properties (base colour, roughness, metallic, specular, anisotropic, sheen, clearcoat), random texture assignment from low-fidelity images

Table 1: Simulation aspects randomised during our domain randomisation procedure. We note that the object surface properties listed correspond to specific Blender Principled BSDF Shader parameters [3, 4].

relative to its bottleneck pose. The translation is sampled uniformly in the ranges $[-0.011, 0.011]$,$[-0.011, 0.011]$, and $[-0.06, 0.26]$, and the rotation in the range $[-90°, 90°]$ around the vertical axis, such that the object is at least partially in view in the live image. For each of these poses we render the live image, and the object (as opposed to distractor) segmentation mask. We note that for the learned visual servoing datasets we also store the labels for e_{xy}, e_s, and e_r. e_{xy} is obtained by projecting the centre of the object's bounding box in the bottleneck and live images, and then finding the pixel space difference between the two projections. e_s is the ratio between the total number of segmented pixels in the bottleneck segmentation and the total number of segmented pixels in the live image. e_r is the relative rotation around the vertical axis between the bottleneck and the live camera poses.

Finally, the whole process is repeated until the desired dataset size is obtained. Our final segmentation network dataset consisted of 160000 datapoints and our final servoing network dataset consisted of 80000 datapoints. A datapoint for the segmentation network consisted of a tuple $(I_{live}, I_{bot}, S_{live}, S_{bot})$, where I_{live} is the live image, I_{bot} is the bottleneck image, S_{live} is the live segmentation and S_{bot} is the bottleneck segmentation. A datapoint for the servoing network consisted of a tuple $(I^{seg}_{live}, I^{seg}_{bot}, e_{xy}, e_s, e_r)$, where $I^{seg}_{bot} = I_{bot} \otimes S_{bot}$, and $I^{seg}_{live} = I_{live} \otimes S_{live}$.

2.2 Domain Randomisation

In order to overcome the reality gap [12, 1], we randomise several visual aspects of our simulator when rendering images, namely light parameters and colour/texture parameters, as listed in table 1. We note that for the colour and texture parameter randomisation, we randomly choose to either randomise surface properties, or to assign random images as textures, which was also the subject of one of the ablation studies in our paper. We finally note that, when generating data, we only randomise the colour and texture parameters once per object loaded in the simulation, but randomise light parameters also in-between rendering the bottleneck image and the live images.

3 Further Experimental Details

3.1 Experimental Setup Implementation Details

In our experiments, the gains for the servoing network are set to $\{0.07cm/s$, $0.07cm/s$, $0.015cm/s$, $0.4rad/s\}$ for $\{g_x, g_y, g_s, g_r\}$, unless otherwise specified. As such, for our gains ablation, our gains become $\{0.21cm/s, 0.21cm/s, 0.045cm/s, 1.6rad/s\}$ for $gains\times 3$, $\{0.35cm/s, 0.35cm/s, 0.075cm/s, 2.0rad/s\}$ for $gains\times 5$, and $\{0.49cm/s$, $0.49cm/s$, $0.105cm/s$, $2.8rad/s\}$ for $gains \times 7$. We set our maximum control rate to $30Hz$, although in practice during visual servoing the execution frequency may become slower due to the time needed for computations between each iteration. For our learning-based visual servoing controller, we threshold the output of the visual servoing network in order to detect when the live and bottleneck images are aligned with those thresholds set to 1.28pixels for e_{xy}, 0.01 for e_s and $1.5°$ for e_r. If all these outputs are below their respective thresholds, then we consider that the live and bottleneck images have been successfully aligned.

3.2 Segmentation Quality Dependence Evaluation Details

For our segmentation quality dependence evaluation experiment, we used Perlin noise and extra segmentation artifacts in order to make noisy segmentation masks. Our perlin noise segmentation warping procedure is inspired by [11], and [14]. The Perlin noise is generated as follows. First, we choose a random octav value from $\{2, 3, \text{and } 4\}$. Then we sample uniformly the lacunarity in the range $[2.0, 4.0]$, and the frequency uniformly in the range $[0.005, 0.1]$. Using these parameters we generate a vector field $\mathbf{F}$, where $\mathbf{F}_i$ is a 2D vector representing the direction where to warp pixel i. To get the warped segmentation map, we simply take the value of each pixel position i and place it at pixel position j such that $j = \mathbf{F}_i \times w \times i$, with w uniformly sampled in the range $[1, 12]$. For the extra segmentation artefacts we used in our extra 1,2 and 3 noise settings we proceeded as a follows. We first randomly sampled 1,2 or 3 pixels in the image space, and created squares of size between 4 and 12 pixels (randomly chosen) around those pixels. We then warped those artefacts using our Perlin noise procedure and finally added them to the original mask. Examples of noisy segmentations for each of our noise settings can be seen in Fig. 4.

3.3 Image Conditioned Object Segmentation Evaluation Details

We show 13 examples from our image conditioned object segmentation network test set in Fig. 5. These showcase the 13 objects used in order to create that test set, and feature the bottleneck image, the live image, and the hand-segmented ground truth segmentation.

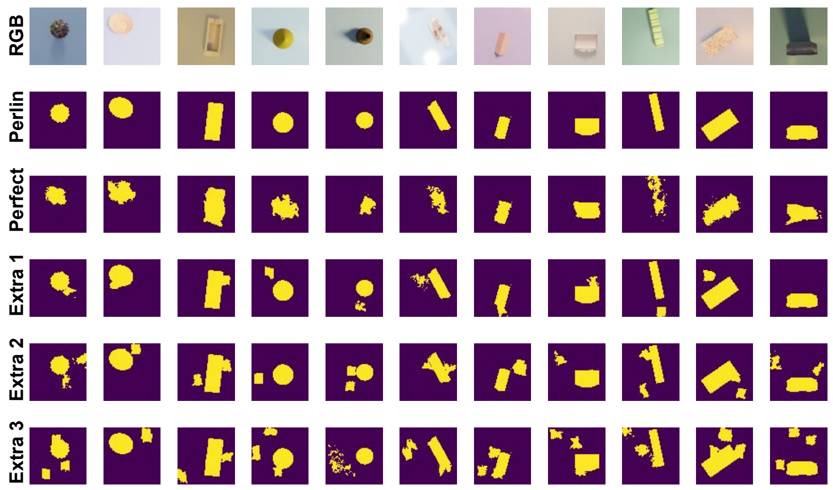

Figure 4: Examples from our segmentation noise settings for our segmentation quality dependence evaluation experiment.

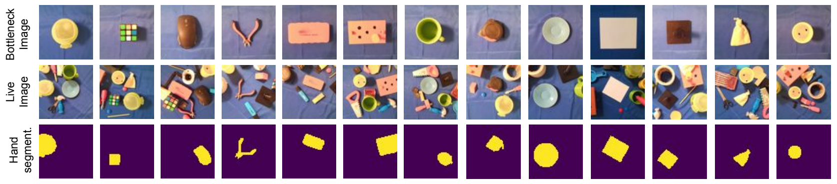

Figure 5: Examples from our image conditioned object segmentation network test. Top: Bottleneck images. Middle: Live images. Bottom: Hand-segmented ground truth segmentations.y

References

[1] R. Alghonaim and E. Johns. Benchmarking domain randomisation for visual sim-to-real transfer. In *IEEE International Conference on Robotics and Automation*, 2021.

[2] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, J. Xiao, L. Yi, and F. Yu. ShapeNet: An Information-Rich 3D Model Repository. Technical Report arXiv:1512.03012 [cs.GR], 2015.

[3] B. O. Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018.

[4] M. Denninger, M. Sundermeyer, D. Winkelbauer, Y. Zidan, D. Olefir, M. Elbadrawy, A. Lodhi, and H. Katam. Blenderproc. *arXiv preprint arXiv:1911.01911*, 2019.

[5] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. Focal Loss for Dense Object Detection. In *IEEE International Conference on Computer Vision (ICCV)*, 2017.

[6] A. A. Minai and R. D. Williams. On the derivatives of the sigmoid. *Neural Networks*, 1993.

[7] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *International Conference on Machine Learning*, 2010.

[8] E. Perez, F. Strub, H. de Vries, V. Dumoulin, and A. C. Courville. Film: Visual reasoning with a general conditioning layer. *CoRR*, abs/1709.07871, 2017.

[9] O. Ronneberger, P. Fischer, and T. Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *International Conference on Medical Image Computing and Computer Assisted Intervention (MICCAI)*, 2015.

[10] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 2014.

[11] S. Thalhammer, K. Park, T. Patten, M. Vincze, and W. Kropatsch. Sydd: Synthetic depth data randomization for object detection using domain-relevant background. *TUGraz OPEN Library*, 2019.

[12] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2017.

[13] D. Ulyanov, A. Vedaldi, and V. Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016.

[14] E. Valassakis, N. Di Palo, and E. Johns. Coarse-to-fine for sim-to-real: Sub-millimetre precision across wide task spaces. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021.

[15] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao. 3d shapenets: A deep representation for volumetric shapes. In *IEEE conference on computer vision and pattern recognition*, 2015.

[16] C. Yu, Z. Cai, H. Pham, and Q.-C. Pham. Siamese convolutional neural network for sub-millimeter-accurate camera pose estimation and visual servoing. *arXiv preprint arXiv:1903.04713*, 2019.