Realtime image adaptation: Randomized-to-Canonical Adaptation Networks for visual recognition

Jochen Jacobs
Technical University Munich
Munich, Germany
jochen.jacobs@tum.de

Sohrab Tawana
Technical University Munich
Munich, Germany
sohrab.tawana@tum.de

Frederik Olberg
Technical University Munich
Munich, Germany
olberg@in.tum.de

***Abstract*—The main objective of this project is to implement a randomized-to-canonical adaptation network (RCAN) to convert images of a scene with randomized textures to a canonical representation. The given scene consists of a table with a robot arm and a cylinder. The resulting images can be used to train a reinforcement learning algorithm. The project consists of two parts: Firstly, we generate training data by taking images of the randomized and canonical scenes. Afterwards we design and train a RCAN network. That RCAN network gets wrapped into a ROS node. The ROS node takes the visual input of scene and converts it into a canonical representation in real-time. Our results speak for using a RCAN for visual recognition to bridge the gap between training in simulation and application in the real world.**

I. Introduction

The initial motivation for this project is the need of a lot of training data to learn specific tasks for deep reinforcement learning models. This learning of specific tasks is usually done in simulation because the training data is limited or cumbersome to acquire in the real world. On the one hand learning in the simulation offers the advantage of theoretically unlimited training data. On the other hand there is the disadvantage of overfitting the trained models to the simulation data, making the simulation model rarely applicable to the real world. To prevent that from happening the most distinct properties like colors, textures, positions and lighting of the simulation are randomized, in order to make the models more robust and keep their performance high when they are transferred to the real world. However, training a reinforcement learning model directly on the randomized simulation is challenging and it leads to long training times or poor performance. To circumvent that the entire problem was split up into two tasks.

We are the first group of this two-group project, therefore we tackle the upstream task of building a Neural Network (NN) infrastructure that takes visual input from a randomized scene and transforms it into a canonical representation. This canonical representation can then be taken as a basis for the reinforcement training of the robot arm in simulation to perform a task e.g., knocking a small cylinder off a table. The intention of this approach is to help overcome the gap between simulated and real-world images, because input from a real scene would appear to the NN to be another variation of a randomized scene. In our case we use a Generative Adversarial Network (GAN) [1] as our NN setup.

Next we briefly go over the related literature, followed by our approach to the problem. Our solution consists of multiple steps like definition of the canonical representation, building a script for environment and robot arm randomization, generating training data, creating and improving the GAN architecture and finally our results and conclusion with ways to improve our results.

II. Related Work

Domain randomization has first been introduced by [2]. They argue that using domain randomization can help overcome the reality gap where a system trained on artificial data is not able to generalize to images from the real world. Domain randomization can be used to make the artificial data as diverse as possible so that "the real world may appear to the model as just another variation".

As the large amount of variability in the data makes it harder for a reinforcement learning algorithm to learn, S. James et al. [3] propose to use a two-stage learning approach. Here first the images of the random domain are converted into a canonical representation that is then used to train the reinforcement learning algorithm.

In this project we implement the first stage of this approach. Here S. James et al. [3] introduce Randomized-to-Canonical Adaptation Networks (RCANs). RCANs are a type of image-conditioned generative adversarial networks (cGANs)[4]. In cGANs, a typical U-Net architecture [5] is extended by a discriminator. The discriminator is typically used in addition to a standard L1 loss function and acts as an additional loss for the generator. The discriminator is fed with the generated data and the original input image and is tasked to determine if the output is the result of the U-Net or a real image of the output domain. This follows the standard GAN architecture[1] where the discriminator is tasked with differentiating between generated (fake) and real images. However, a standard GAN architecture does not use an L1 loss function. Because cGANs do have an L1 loss function the discriminator network can

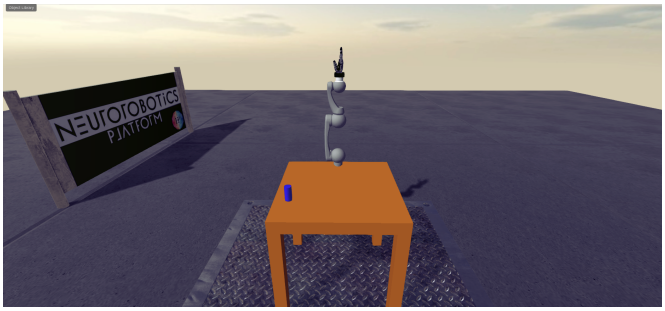


Fig. 1. Pre-Built Gazebo experiment

be simpler. Specifically, as the L1 loss already handles low frequency accuracy the discriminator only needs to check for differences in high frequency sharpness. Therefore, the discriminator of the Convolutional Neural Network can be implemented as a "PatchGAN" where only small patches of the images are analyzed. This discriminator models the image as a Markov random field and assumes independence between pixels separated by more than a patch [4].

The approach of S. James et al. [3] also uses additional outputs of the RCAN as an auxiliary task. Auxiliary tasks [6] can be used as an additional level of regularization. In our case we decided not to use additional outputs for our implementation as our application was simple enough that this regularization was not needed.

Our task consists of two main components: Generation of Training Data for the NN and the transformation in the NN of the image itself.

III. Training Data Generation

Next we describe the steps needed to create training images for our NN. As a robotics simulation platform we use the Neurobotics Platform (NRP) [7]. It uses the Gazebo simulation engine[1] of the ROS framework[2].

A. Canonical Representation

We started off by defining the canonical representation with group two of this project to make sure our final product is concurrent with their needs. We were provided with a pre-built Gazebo experiment from a previous group project. It includes a table, a small cylinder on the table and a robot arm attached to the top of the table as well as a NRP banner in the background. It can be seen in Fig. 1. We modified that scene according to our agreement on the canonical representation. We removed the banner, changed the background and floor color to black, changed the table's color to white, and set distinguishable colors for every joint of the robot arm. We also added a virtual camera to the scene to get an image from a predefined perspective. Examples of this canonical representation can be seen in Fig. 2 (b). That definition of the canonical representation of the scene is being used. That state of the NRP experiment was shared via a repository with group two of the project.

[1]Gazebo: http://gazebosim.org/
[2]ROS: https://www.ros.org/

B. Environment Randomization

To be able to build our RCAN that turns images of randomized scenes into our agreed upon canonical representation we needed to generate a big number of training image pairs. Each pair consists of one image of the canonical representation and one image with randomized textures and lighting.

As a first step the parts and degree of randomization of the scene must be specified. The higher the degree and number of items that are randomized, the higher the robustness in the transfer to the real world. As a trade off the high degree of randomization increases the difficulty of training the NN, to get reliable results, significantly. Therefore, the degree of randomization and number of items to be randomized must be chosen with multiple factors in mind. We agreed upon leaving the size and position of the table as they are, as those parameters would most likely not change in a real-world scenario as well. The randomized objects include the table, cylinder, floor and every individual module of the robot arm. The pool for our random textures is the standard textures offered by the Gazebo plugin in the NRP. Every single part of the scene is randomized separately. Also the color of the light source changes in the real world, so the color of the light source needs to be randomized too.

In addition the size and position of the cylinder are randomized as well, as the object to be manipulated by the robot arm would most likely be different in a real-world scenario too. Although the positional randomization of the cylinder is limited to the surface of the table and the size is constrained to not get too big or small for group two's task of knocking it off the table. After some experimentation, a scaling factor in the range between 0.5 and 1.5 was found to be a good range for the randomization of the size of the cylinder.

To be able to take pairs of images of the same scene – once with randomized textures and once in the canonical representation – we implemented a function to turn the parameters of the scene back into the canonical representation. That enables us to automate the image pair generation, without having to manually interfere.

During the implementation of the environment randomization, we encountered a bug in the `SetVisualProperties` service of the NRP. To solve this issue, we implemented our own gazebo plugin. More information about the issue and the plugin can be found in appendix A.

C. Robot arm randomization

Furthermore, the position of the robot arm is randomized. Every single joint of the robot arm can be turned using the gazebo simulation. Thus, randomizing every single joints angle value leaves the robot arm in a random position which unfortunately can end up in a wedged position. In order to prevent this issue we simply compare the position of each individual joint to other known objects positions, e.g. the height of the table. This solution does not avoid every bad

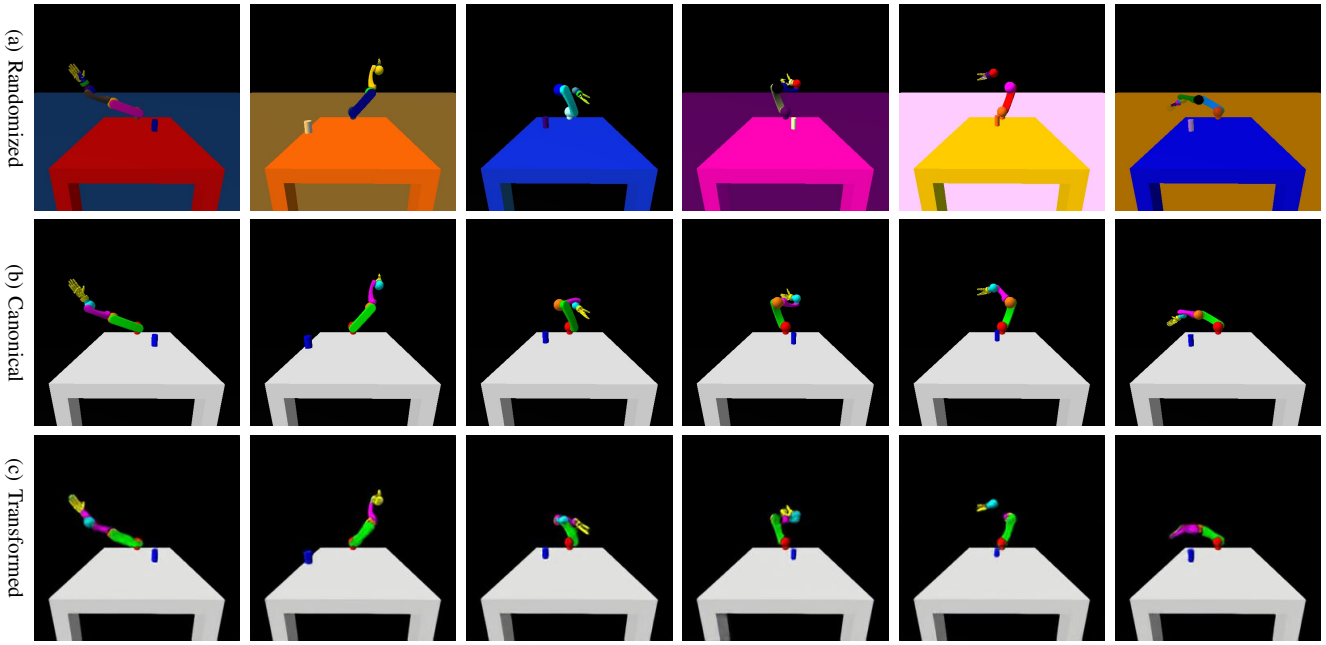


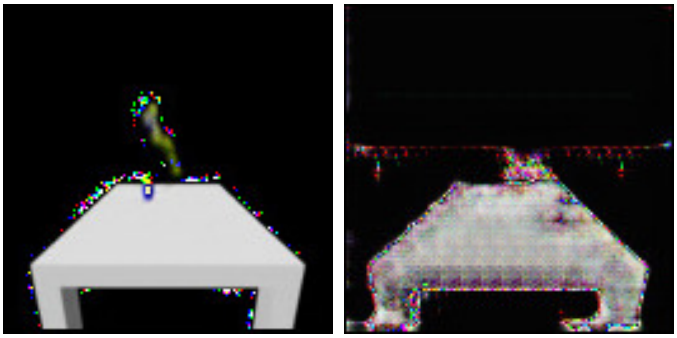
Fig. 2. Example of the training data and resulting output of the generator. (a) Images of the scene with randomized textures applied; (b) The same scene with the canonical textures; (c) the output of the generator with the randomized images as input.

case, but reliably ensures that the arm does not disappear under the table. Consequently, more of our produced training data is valuable for group two to work on.

For our ROS-implemented camera to be able to take photos, the simulation must run continuously. An irreconcilable problem that arose from this, is to script the animation of the individual joints. For this reason we deviated from the standard way of controlling each joint individually to a generalized approach of setting every property regarding the robot at once by using the `SetModelConfiguration` service. The only drawback of this approach is that the changed physical environment means that the position data must be sent a second time before the second image is made in the canonical representation. However, since other objects are also randomized or set for this purpose, it does not cause any additional expenditure of time.

D. Training Data Generation script

The environment randomization and robot arm randomization were combined in a training data generation script. This script subscribes to the virtual camera topic in the Gazebo scene to be able to get the published images. At first it randomizes the robot arm position, applies the canonical textures, and takes a virtual photo with the virtual camera. Then it randomizes the scene with the robot arm in the same position and takes a second virtual photo and saves those paired into a predefined directory. The camera angle and light source position stay non-randomized in this whole process. As a result, our script can generate about 10,000 pairs of images in approximately 5 hours. A sample of the training set can be seen in Fig. 2 (a) and (b). That time is constrained by the speed of the NRP simulation running locally on our computers.



(a) L1 Loss (b) L1 + Discriminator Loss

Fig. 3. Early tries of training the neural network. (a) only using L1 Loss to train generator; (b) after adding a GAN discriminator. Initially the GAN discriminator reduced the quality of the results.

The option of running multiple instances of the NRP on one computer at the same time using docker was neglected due to the sufficient size of the training data as well as the additional effort it would have taken. This extension would mean that it would also have to be ensured that seeds and the resulting images differ.

IV. Neural Network Training

The second part of the project consists of the implementation and training of the RCAN. First a dataset loader had to be implemented.

A. Dataset Loader

As a library for our RCAN implementation, we use PyTorch. To be able to train a RCAN we need a Data Loader.

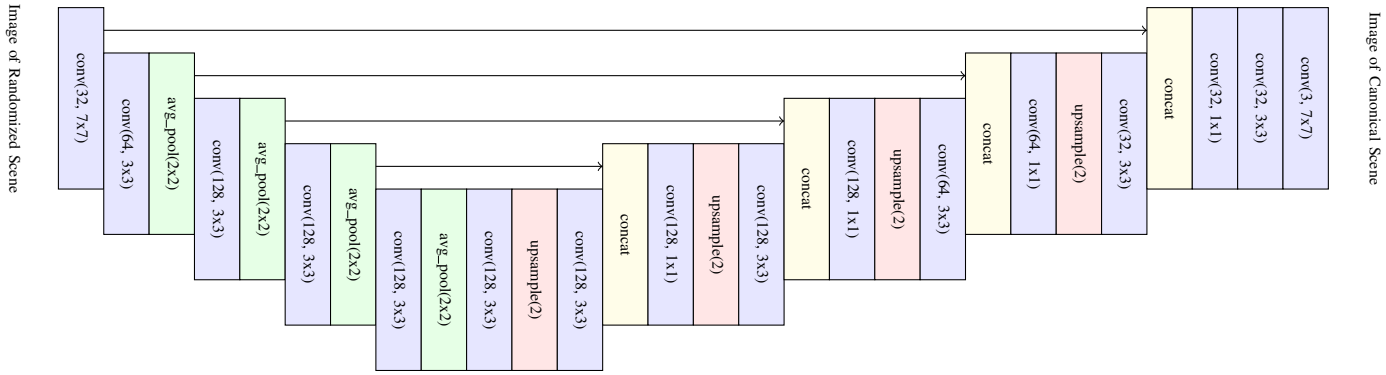


Fig. 4. Network architecture of the generator. Convolutions are referenced as *conv(out channels, kernel size)*; Average Pooling is referenced as *avg_pool(kernel size)*; Upsampling is referenced as *upsample(scale factor)*. All convolutions are same convolutions (i.e. the padding is chosen such that input and output image size is the same). Every convectional layer except the final one is followed by a batch norm layer and a leaky ReLU activation with $a = 0.2$. Concat layers combine the channels of both inputs as separate channels of the output.

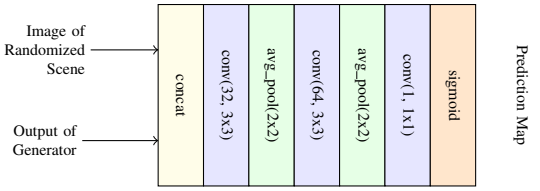


Fig. 5. Network architecture of the discriminator. All convolutions are same convolutions (i.e. the padding is chosen such that input and output image size is the same). Every convectional layer except the final one is followed by a batch norm layer and a leaky ReLU activation with $a = 0.2$. Concat layers combine the channels of both inputs as separate channels of the output.

We implemented one which takes a CSV file with the pairs of images linked to each other via the file name, a directory of all the image pairs and an optional feature of transforming the image pairs.

B. GAN to RCAN

A Generative Adversarial Network consists of a generative model or Generator G and a discriminative model or Discriminator D. The exact process is described in detail in the original paper by Goodfellow et al [1]. We studied various pre-built GAN architectures and concluded to build our own Generator and Discriminator. The pre-built solutions were not fitting for our case.

We first experimented with only training a simple U-Net architecture and an L1 loss function without the discriminator. We achieved decent results as shown in Fig. 3 (a). Our first attempts of adding the discriminator did not yield the desired outcome and instead decreased the quality of the results as shown in Fig. 3 (b). While the general architecture of GANs makes it hard to determine what exactly is causing the issues, we applied several changes: We implemented a PatchGAN discriminator as described in section II, the weighting between the L1 loss and the discriminator loss was adjusted, and gradient clipping was added. Further experimentation was done on the depth of the U-Net and channel count. Finally, the resolution of the input and output images has been increased.

C. Final Architecture of our RCAN

The final architecture of our generator network can be seen in Fig. 4: The generator uses a four-layer U-Net architecture. Here every convolutional layer is followed by batch normalization and a leaky ReLU activation with a slope of 0.2. First, a 7x7 convection with 32 channels is used, followed by the U-Net block. Each U-Net block starts with a 3x3 convolution and down-sampling using a 2x2 average pooling. Then the next U-Net block is used recursively, followed by upsampling with a scale factor of two and a 3x3 convolution. Afterwards, the input of the U-Net block is concatenated to the output of the last convolution (skip-connection). To combine the channels of the concatenated inputs a 1x1 convolution is used. This process is done four times recursively with a channel count of 64, 128, 128, 128 respectively in each block. The innermost block uses a 3x3 convolution with 128 channels as the inner block. After the application of the U-Net blocks, a 3x3 convolution with 32 channels and a final 7x7 convolution with three channels is used. The final convolution does not use batch normalization nor a activation function.

Our final discriminator can be seen in Fig. 5: The output of the generator is concatenated with the input image. Here we use a 3x3 convolution with 32 layers followed by down-sampling, a 3x3 convolution with 64 layers, down-sampling again and a final 1x1 convolution with one channel (being used). The first and second convolutional layers are then followed by batch normalization and a ReLU activation with a slope of 0.2. After the final convolutional layer, a sigmoid activation function takes the end of our architecture.

The generator is trained with a loss that is calculated as $(L_{\text{discriminator}} + 10 \cdot L_{L1})/11$ where L_{L1} refers to the L1-Loss between the output image and the target and $L_{\text{discriminator}}$ refers to the Binary Cross Entropy loss between the output of the discriminatory and a target prediction map of "all true". The

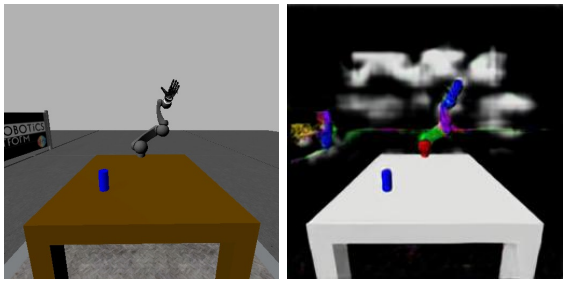


Fig. 6. Applying the final neural net to the original scene reveals some problems with our dataset.

discriminator is trained with Binary Cross Entropy loss on pairs of real data with a target prediction map of "all true" and pairs of data generated by the generator with a target prediction map of "all fake". Both generator and discriminator are trained using the Adam optimizer with a learning rate of 0.0002 and beta of 0.5.

D. Training Process

We trained our first epochs on our local computers which have no Graphical Processing Units (GPUs) that can be used for parallelization by Nvidia's CUDA library. Training the RCAN on our computers without GPUs was very time intensive and slowed the potential improvement of the architectures down by a lot. Thus, we started using Google Colab[3] as they offer free GPU resources including support for native PyTorch libraries. In the meantime, we approached the instructors for a server solution to make the training of our RCAN architecture less time-consuming.

Another measure that could help improve producing more training data in less time was the possibility to run the simulation of NRP in parallel using an already existing Docker project. However, the lack of computing power and synchronisation issues prevented us from taking a closer look at realizing this approach, as the potential gain in time seemed small from our point of view. Therefore, we focused our time more on the key features of the project since our accumulated training data set at this point in time seemed sufficient.

V. Results

Training our RCAN architecture on the Google Colab resources enabled us to complete more training epochs in less time and improve our RCAN quicker. The results can be seen in Fig. 2 (c). Overall the result are very close to the target images. However, the network struggles to determine the order of the separate elements of the robot arm. As a different order of links implies a significantly different robot pose this issue would cause problems when the output is passed to the reinforcement learning algorithm. This could be solved by providing the RCAN or reinforcement learning with additional information about the robot pose, as this information is usually available.

[3]Google Colab: https://colab.research.google.com

Converting the default version of the Gazebo experiment in Fig. 1 into the canonical representation works reasonably well. In Fig. 6 it can be seen, that the background and banner are problematic for our RCAN architecture, due to them not being in the data, with which the RCAN was trained. We assume that including objects like the banner and different backgrounds into the training data would improve those parts of the canonical representation of the default Gazebo experiment and deliver comparable results like those in Fig. 2 (c).

After getting satisfying results consistently, we wrapped our RCAN into a ROS Node. That node takes the visual input of the virtual camera on the NRP and turns it into the canonical representation in real-time. That representation can be seen in the video output of the virtual camera.

VI. Conclusion

Developing a RCAN that turns a simulated randomized scene into its canonical representation is very feasible and can help a lot in bridging the gap between training in the simulation and application in the real world. Although our results can be improved in many ways. The resolution of the output canonical representation can be increased by a factor of two or four.

To make the RCAN more robust multiple slight changes in the camera position can be taken into the randomized parameters of the training data. Camera angles in the real world will most likely not be at the exact same spot and it is the likeliest source of problems for the real-world transition. Furthermore, the generated training data can be augmented by transforming the training images. Transforming images is used to artificially augment the training data pairs through e.g. mirroring them on a vertical or horizontal axis in the middle of the image, without having to generate those with our training data generation method. That simultaneously lowers the level of overfitting the RCAN to the simulated data.

For further robustness many more different texture, environment and light source changes can be included in the randomization scripts while generating the training data. In addition, it would be useful to randomize the camera position itself to cover more cases of the real world.

However, every added randomization can increase the difficulty of reaching a RCAN that converges and delivers satisfying results. A fair trade-off or balance has to be struck, to optimize the RCAN for the sim-to-real adaptation and at the same time keep the training of the RCAN feasible. Something that is also to be kept in mind is to avoid the problem of overfitting by modelling the training data too well. What is certain is that more computing power and time could allow for a more complex RCAN architecture. Gradually changing the architecture and retesting could also yield new insights.

References

[1] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Nets," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, Eds.,

vol. 27, no. January. Curran Associates, Inc., 2014, pp. 2672–2680. [Online]. Available: https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf

[2] 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," *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 2017-Septe, pp. 23–30, 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8202133/

[3] S. James, P. Wohlhart, M. Kalakrishnan, D. Kalashnikov, A. Irpan, J. Ibarz, S. Levine, R. Hadsell, and K. Bousmalis, "Sim-To-Real via Sim-To-Sim: Data-Efficient Robotic Grasping via Randomized-To-Canonical Adaptation Networks," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 2019-June. IEEE, 2019, pp. 12 619–12 629. [Online]. Available: https://ieeexplore.ieee.org/document/8954361/

[4] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-Image Translation with Conditional Adversarial Networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 2017-Janua. IEEE, 2017, pp. 5967–5976. [Online]. Available: http://ieeexplore.ieee.org/document/8100115/

[5] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9351. Springer Verlag, 2015, pp. 234–241. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-24574-4_28

[6] L. Liebel and M. Körner, "Auxiliary tasks in multi-task learning," 2018. [Online]. Available: http://arxiv.org/abs/1805.06334

[7] E. Falotico, L. Vannucci, A. Ambrosano, U. Albanese, S. Ulbrich, J. C. Vasquez Tieck, G. Hinkel, J. Kaiser, I. Peric, O. Denninger, N. Cauli, M. Kirtay, A. Roennau, G. Klinker, A. Von Arnim, L. Guyot, D. Peppicelli, P. Martínez-Cañada, E. Ros, P. Maier, S. Weber, M. Huber, D. Plecher, F. Röhrbein, S. Deser, A. Roitberg, P. van der Smagt, R. Dillman, P. Levi, C. Laschi, A. C. Knoll, and M.-O. Gewaltig, "Connecting Artificial Brains to Robots in a Comprehensive Simulation Framework: The Neurorobotics Platform," *Frontiers in Neurorobotics*, vol. 11, p. 2, 2017. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnbot.2017.00002

[8] Laboratory of Computer Vision & Robotics - VeRLab, "Gazebo ROS Model Color," Belo Horizonte, Brazil, 2018. [Online]. Available: https://github.com/verlab/gazebo_ros_model_color

APPENDIX

A. Gazebo Plugin to change materials

For the randomized colors we needed to apply different colors to each element of the robot arm. The Robot arm is a single model with multiple links. Originally, we used the `SetVisualProperties` ROS service that is provided by the NRP. This service needs to be called for each link we want to change. However, if it is called for different links in the same model in quick succession only the change from the last call is applied permanently. Changes from earlier calls are reverted by the next call. This is caused by a race condition in the implementation of that service. We reported the bug in the NRP Forum[4] but it has not been fixed yet.

To get around this issue, we created our own Gazebo plugin that directly changes the material of each link. It is based on a similar plugin that can only set the material color [8]. We have modified it such that it instead changes the material itself. The modified plugin can be found on GitHub at https://github.com/jacobsjo/gazebo_ros_set_material.

We added this plugin to every link of the robot. The plugin then advertises a separate service for changing the link material of each link.

[4]see https://forum.humanbrainproject.eu/t/1449