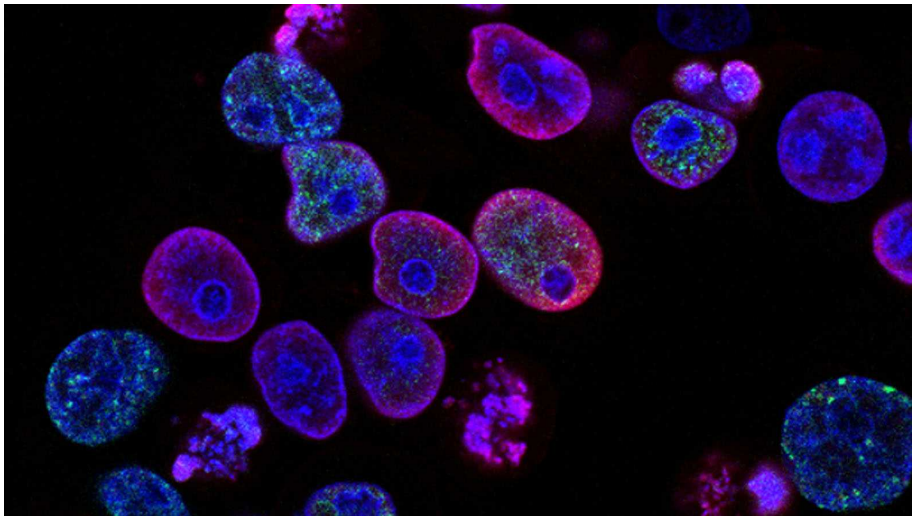
DAVID BAO FU, ISABELLA GAGNER, JAIRO ALFONSO

The Adipocyte Cell Imaging Challenge

The NordAxon Code Monkeys

November 15, 2020



1 HIGH LEVEL DESCRIPTION OF SUBMITTED SOLUTION

In this report, the NordAxon Code Monkeys proudly present our solution to the Adipocyte Imaging Challenge hosted by Astra Zeneca in collaboration with AI Sweden! We would like to thank the organizers for the opportunity to be part of this competition, and our fellow colleagues at NordAxon for their unwaivering support during these weeks.

1.1 HIGH LEVEL SOLUTION DESCRIPTION

In our application, we proposed both simple U-Net++ and Pix2Pix GAN architectures as solutions to this image-image translation problem. Due to the time constraints of the competition, we have decided to stick with the Pix2Pix GAN architecture, as they converged quicker during training! We want to be clear that it is possible that the U-Net architecture on its own could solve the problem, possibly even better given the right conditions, but as they were slower in training we had to focus on the GANs.

For each target output, we have created one GAN model respectively. This means that depending on the target the user wishes to predict, a different model will be used. Figure 1.1 shows the final high-level solution. We also created a segmentation model to be able to predict masks for the nuclei target. This model is explained more in detail under 2.1.

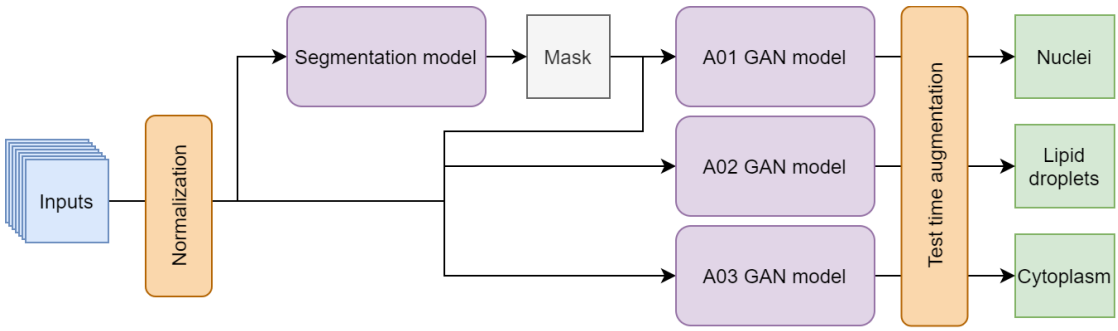


Figure 1.1: Flow chart of our high-level solution at test time.

This means that in total, we have 4 models - one for predicting nuclei masks, and then one for each target image. When training these models, the data flow of course looked slightly different. A training time flow chart can be seen in Figure 5.1, further down the report.

During the competition, we trained one model at a time. The main steps for training the different models were as follows:

1. Segmentation model:
    a) Normalize the input brightfield images
    b) Create nuclei masks using maskSLIC (see 2.1)
    c) Train the U-Net segmentation model with all seven input images (see 2.1.1)
2. A_1 GAN model:
    a) Normalize the input brightfield images
    b) Load the corresponding mask as input
    c) Train the A_1 GAN model with augmentations (see 4.1.2)

3. A_2 and A_3 GAN models:
    a) Normalize the input brightfield images
    b) Train the GAN model with augmentations (see 4.1.2)

These steps and models will be explained more in detail further down the document.

1.2 VARIABLE NAMES

In this report, the different inputs and targets may be described using aliases. These aliases are described in Table 1.1 below.

Table 1.1: Table of the input and target names and their corresponding aliases.

Name	**Alias**
Target 1: Nuclei	A_1
Target 2: Lipid Droplets	A_2
Target 3: Cytoplasm	A_3
Input Brightfield Images	Z_1-Z_7

1.3 TABLE OF SECTIONS AND CORRESPONDING CODE

In Table 1.2 below, we have gathered the sections and main parts of the pipeline with pointers to the code, to facilitate for the reader.

Table 1.2: Table of the sections in this report and the corresponding code segments in the code repository.

Section	**Location in code**
Feature engineering	
Masking	`src/features/maskslic.py`
	`src/models/unets.py`
	`src/train_segmentation.py`
Data processing	
Normalization (training)	`src/prepare_training_data.py`
Normalization (inference)	`src/prepare_inference_data.py`
Augmentation	`src/data/augmentations.py`
Model architecture	
U-Net implementations	`src/models/unets.py`
Pix2Pix training	`src/train_pix2pix.py`
Discriminator implementation	`src/models/networks.py`
Loss function	`src/utils/losses.py`
Post processing	
Histogram matching	`notebooks/Local/10_Histogram matching.ipynb`
	`src/utils/postprocessing.py`
End-end inference pipeline	`notebooks/End-to-end pipeline.ipynb`

2 FEATURE ENGINEERING

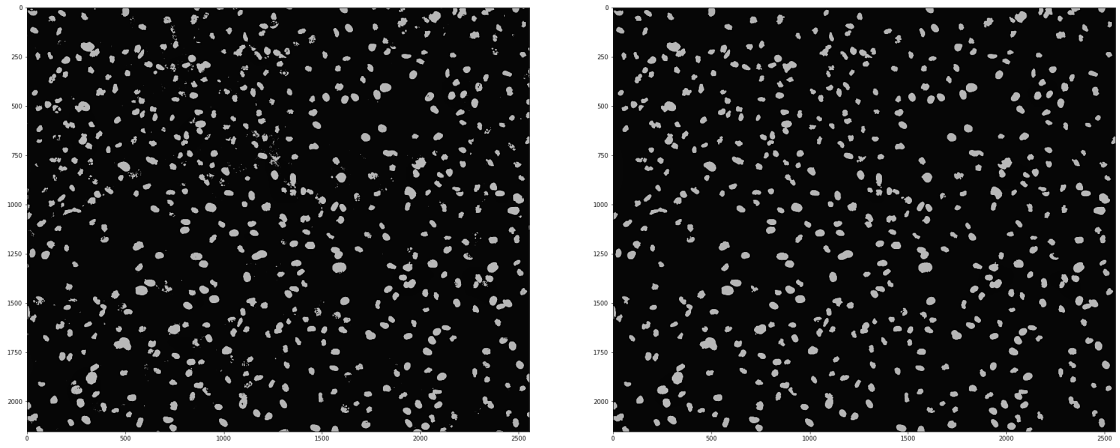
In this section, we will present the chosen method for feature engineering. In our pipeline, this method was masking one target image - the nuclei.

2.1 MASKING

When computing our baseline for the application to this competition, we found that there were some problems when trying to predict target A_1, the nuclei. The model had difficulties finding the nuclei in the the input images. Due to this, we decided to compute masks for the A_1 images.

Starting with using only a threshold value to compute the masks, we found that the masks turned out noisy (see Figure 2.1). To improve the masking, we decided to use the *maskSLIC* method, as described by Irving [2016]. This method is available in the `scikit-image` library, and we used a threshold of $2 \cdot \mu_{image}$, where μ_{image} is the mean of the image that is to be masked.

Using these masks, we trained a segmentation model, as mentioned in section 1.1. The model is described in section 2.1.1 below. This segmentation model will, given the 7 brightfield inputs, predict a mask that will be used downstream in the A01 GAN model. This is done to avoid target leakage - we need a model that can run only given the brightfield images and not be dependant on the actual nuclei images for the masks.



(a) Threshold mask.

(b) maskSLIC mask.

Figure 2.1: Comparison of a simple threshold mask and the maskSLIC implementation. Some noise is visible in the threshold mask, that is not present in the maskSLIC mask.

2.1.1 Segmentation model

The segmentation model for masking for the A_1 targets is a U-Net model, with the ResNet152 as backbone. The U-Net model is presented in a paper by Ronneberger et al. [2015] and the ResNet152 is presented by He et al. [2015]. In Figure 2.2, a simple overview of the architecture is shown. We chose the U-Net model for this task as it has been previously successful in biomed-

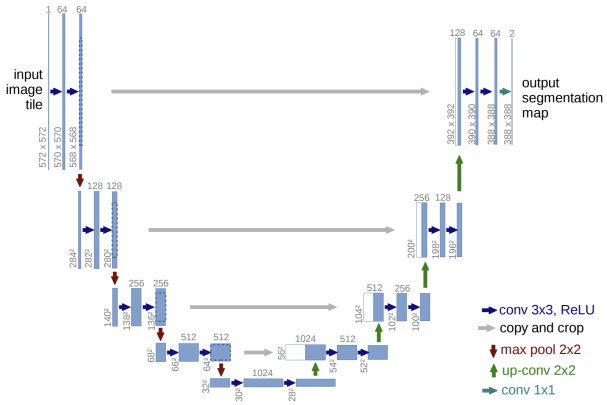


Figure 2.2: Overview of the UNet architecture from Ronneberger et al. [2015].

ical image segmentation tasks [Ronneberger et al., 2015], and we decided to pair it with the ResNet152 backbone because it has proven successful in similar tasks (see the winners in the 2018 Kaggle competition found at `https://www.kaggle.com/c/data-science-bowl-2018/overview`), and we wanted a backbone with skip-connections to avoid vanishing gradient problems.

When implementing this model with the ResNet backbone, we used the library called `segmentation_models.pytorch` [Yakubovskiy, 2020]. This facilitated the implementation as it could be used out-of-the-box, with only a few parameters to be chosen.

2.2 Feature engineering parameters

In the table below we present the parameters chosen for creating the masks using maskSLIC, and the parameters for training the segmentation U-Net model.

Table 2.1: A table of the data processing parameters.

General parameters	**values**
maskSLIC threshold	$2\mu_{image}$
Segmentation model parameters	**values**
kernel size	(3,3)
initial learning rate	$1e-3$
batch size	32
epochs	400

3 Data processing

In this section, we will present our data processing steps. The steps taken were normalisation of the input images and augmentations implementations, as described in their respective sections below.

3.1 NORMALISATION

When analysing the input data $Z_1 - Z_7$ and the value distributions over the different magnifications, we got the following numbers. It is quite easy to distinguish a difference between the magnifications. We are aware that the distributions of the values are not normally distributed (as can be seen clearly in Figure 3.1), but have decided to approximate them as such to simplify the normalisation process.

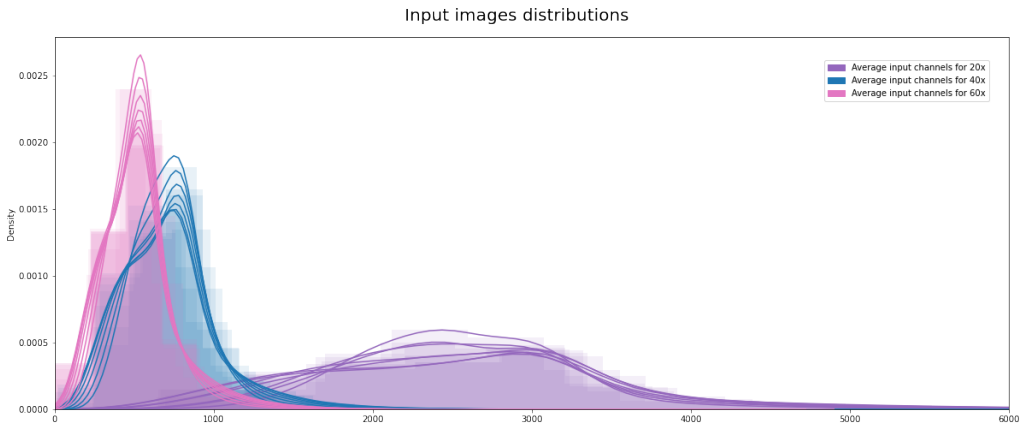


Figure 3.1: A visualisation of the general value distributions of the seven inputs. Each line in the plot corresponds to an average value distribution for a Z_x target in its specific magnification.

Table 3.1: Mean and standard deviation of the images, per magnification.

Magnification m	**Mean** μ_m	**Standard deviation** σ_m
20x	2594	966
40x	713	282
60x	524	228

As we have assumed normal distributions, we used the following equation for normalising the input images:

$$X_m^i = \frac{X_m^i - \mu_m}{\sigma_m}, \tag{3.1}$$

where X_m^i is input matrix number i of magnification m, μ_m is the mean value of the magnification and σ_m is the standard deviation of the magnification (i.e. corresponding to the values in Table 3.1).

3.2 AUGMENTATIONS

To simplify the implementation of different augmentations, we decided to use the `Albumentations` package presented by A. Buslaev and Kalinin [2018]. Given that our task is framed as an image-to-image translation problem, and `Albumentations` is developed for classification or segmentation, we had to stack the target with the inputs $Z_1 - Z_7$ to act as one input image for the augmentations. We did this to make sure that the inputs, targets and masks were augmented in the exact same way. When the target corresponded to A_1, the mask as described in section 2.1 above was used.

For a clean implementation, we decided to use a wrapper function for the stacking and unstacking of the inputs and targets. In Figure 3.2 we have visualised the stacking for `Albumentations`.

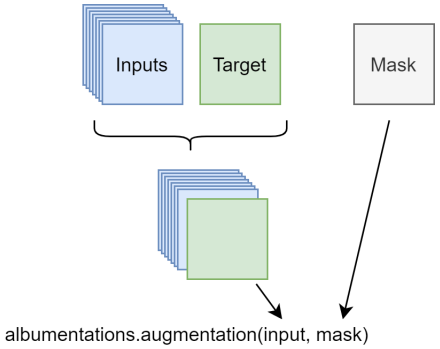


Figure 3.2: A simple visualisation of how we stack the images to be able to use the Albumentations library.

The augmentations that we used were the following:

1. Random Resize and Crop,
2. Vertical Flip,
3. Horizontal Flip,
4. Rotate,
5. Transpose.

In our application, we proposed the usage of more augmentations. These are implemented and available in the repo, but due to the time constraints we chose to only work with the affine transforms that we deemed were safe to use.

All the augmentations are available via the `Albumentations` library. As the augmentations were applied during training and with every augmentation having a probability of $P = 0.5$ of being applied at all, it is hard to estimate the total number of training images.

3.3 DATA PROCESSING PARAMETERS

In the table below, Table 3.2, we present the different data processing parameters used when preparing the data.

Table 3.2: A table of the data processing parameters.

Parameter	**Value**
Augmentation probabilities P	0.5
Normalising parameters	see Table 3.1 and Equation 3.1
Resize augmentation size	(256,256)
Rotate limit angle	9

4 Model Architecture

In this section we will start by motivating our model choice, followed by an explanation of the architecture and its details and parameters. For pointers to code sections, we refer to Table 1.2 and for a high-level visualisation of the models we refer to Figure 1.1.

4.1 Image-to-Image Translation

Image-to-Image Translation is a class of computer vision and graphics problems where the goal is to learn the mapping between an input image and an output image. We use a general purpose 'pixel to pixel' Generative Adversarial Network (GAN) network described in Isola et al. [2018]. This choice is due to issues faced with traditional CNN networks that produce blurry results unless the loss function is carefully tailored for a task. The pix2pix model extends the vanilla GAN network and creates a conditional generative model suitable for image-to-image translation that we can condition on our input images.

4.1.1 The Generative Adversarial Network

In Figure 4.1, we show a simple flow chart of the vanilla GAN overall structure. Shortly described, the GAN architecture is two models competing against one another. The generator takes an input, generates an output and feeds the output to the discriminator model. The discriminator model will be fed with true target images and generated ones, and will try to distinguish which input is generated and which input is the true target. Ideally, the GAN will be fully trained when the discriminator no longer can distinguish between true and generated images. This means that the generator model will be very good at generating targets given an input.

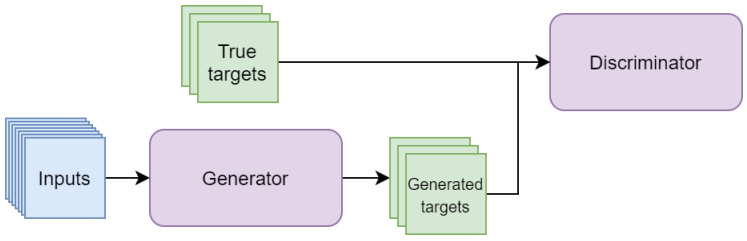


Figure 4.1: An overview of a general GAN architecture.

4.1.2 Our Pix2Pix inspired models

The Pix2Pix model in Isola et al. [2018] is a type of conditional GAN where the generation of the output image is conditional on an input image, in this case, preprocessed brightfield images.

The generator models in our implementation are UNetResnet152 models, i.e. UNet models with ResNet152 as a backbone. This means that the generator in the GAN is the same architecture as the segmentation model described in section 2.1.1, with only a minor detail - kernel size for upsampling in the UNet - separating the implementations. In the Pix2Pix paper, they describe their discriminator as implementing an architecture they call PatchGAN. We have not implemented that, but have used a vanilla CNN as discriminator. Therefore, our solution is not a pure breed Pix2Pix model, but rather a simplified version.

All three GAN models are more or less the same, with the number of input channels to the generator UNetResNet152 being main difference - for A_1, this will be 8 channels due to the mask, and for the other two models the value is 7. This is visible in Table 4.1 below.

4.2 Model Parameters

In Table 4.1 we have gathered the model parameters that we have chosen for the training of these models. Because of the time constraints, yet again, we did not have the time to implement any extensive hyperparameter search.

Table 4.1: A table of the parameters of the models.

Model	**Parameter**	**Value**
Generator A01	Number of channels as input	8
	Kernel size downsampling	(3,3)
	Kernel size upsampling	(5,5)
Discriminator A01	Number of channels as input	8
	Kernel size	(3,3)
Generator A02, A03	Number of channels as input	7
	Kernel size downsampling	(3,3)
	Kernel size upsampling	(5,5)
Discriminator A02, A03	Number of channels as input	7
	Kernel size	(3,3)

4.3 Model Size

As our UNets are based on the ResNet-152 convolutional backbone, the generator models trained in the Pix2Pix pipeline takes around 300MB. The segmentation model takes around 770 MB. The sizes and number of trainable parameters are summarized in Table 4.2.

Table 4.2: A table of information about the models.

Model	**No trainable params**	**Model size**
Generator, A_1	67 173 521	300 MB
Generator, A_2, A_3	83 188 881	300 MB
Segmentation model, A_1	67 169 425	770 MB (contains metadata)

4.4 Model Output

As the Pix2Pix model is fully convolutional, the output size will depend on the input size. As there is one model per target output, the models only output one image with one channel each. This means that an input with shape $(256, 256, X)$ will generate an image of size $(256, 256)$.

The benefit of using a fully convolutional model is that once it is fully trained, no additional test time modifications need to be done. This means that it is ready to run - feed the respective model with the input images and it shall output the target.

5 Model Training

During the competition, we have trained one model at a time on the A100 GPU provided. Due to time constraints, no sophisticated hyperparameter tunings were implemented and tested.

The neccessary steps to reproduce the training pipeline are presented under the section 7 below.

5.1 Training Scheme Overview

For the A1 target, we need to train 2 models. The first is a segmentation model which can be trained using `train_segmentation.py`. The image-to-image translation model is then trained using the `train_pix2pix.py` script with the extra `--mask-input` flag which implements the Pix2Pix cGAN training scheme as described above.

For the A2 and A3 target, you run the `train_pix2pix.py` script to train a the image-to-image translation model for each target using the Pix2Pix cGAN training scheme.

The optimization method used is PyTorch Adam optimizer, and the default values (PyTorch) were used.

5.2 Pre-Training

As mentioned before, we used the ResNet152 as a backbone for our UNet models. This is explained in the paper by He et al. [2015]. Shortly explained, the ResNet152 model is a convolutional, pre-trained model that is 152 layers deep. It has been trained on the ImageNet dataset, reaching as low as 3.57% error on the ImageNet test set [He et al., 2015].

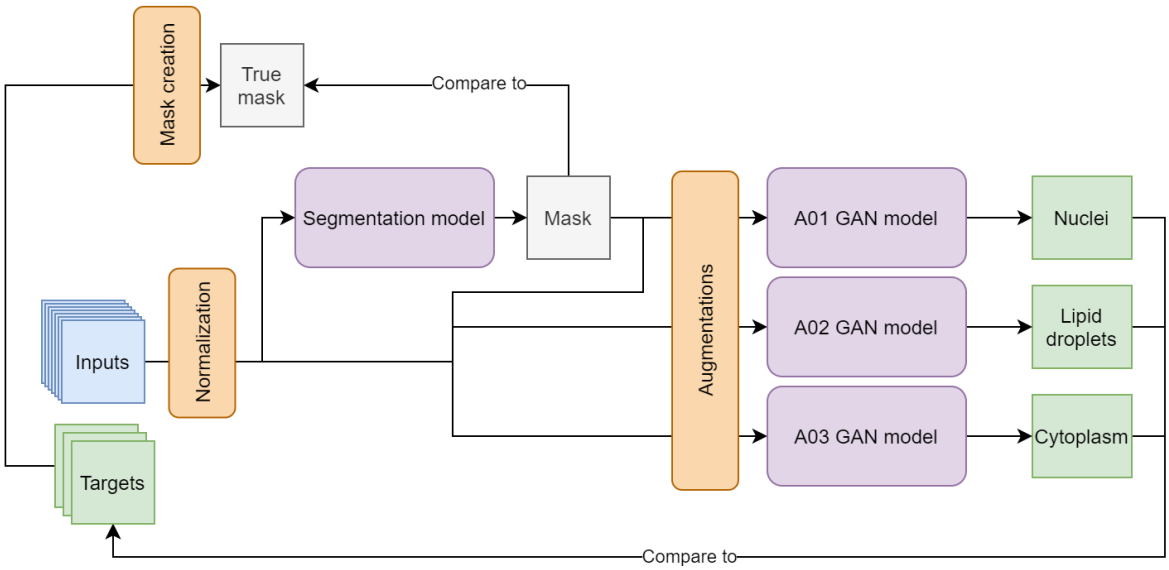


Figure 5.1: Flow chart of how our high-level solution during training time.

The way this is added to the UNet model, is that the encoder part of the UNet architecture consists of the ResNet152 model. Essentially what this does, is allow the UNet to be already trained on extracting features in our data before we even have started the training. This means that the model will be better faster, as it will not need to learn to find and extract features from scratch.

As described in 2.1.1, we used the library `segmentation_models.pytorch` by Yakubovskiy [2020] for a simple implementation of this architecture.

5.3 Loss Function

The generator loss function used was the mean absolute error (MAE), spectral loss and the addition of the discriminator loss.

The discriminator loss function was the mean squared error (similar as in a Least Squared GAN).

The segmentation loss function was the binary cross entropy. We planned to compare it with the soft dice loss but did not due to time constraints.

5.4 Training parameters

In the Table 5.1, we are listing the parameters that were tuned for training. As mentioned before, no sophisticated hyperparameter search could be conducted.

Table 5.1: A table of the parameters of the models.

Model	Training parameter	Value
Generator A01	Initial learning rate	1e-3
	Optimizer	Adam, default parameters
	Learning rate scheduler	Lambda Learning rate policy
Discriminator A01	Initial learning rate	1e-5
	Optimizer	Adam, default parameters
	Learning rate scheduler	Lambda Learning rate policy
Generator A02, A03	Initial learning rate	1e-3
	Optimizer	Adam, default parameters
	Learning rate scheduler	Lambda Learning rate policy
Discriminator A02, A03	Initial learning rate	1e-5
	Optimizer	Adam, default parameters
	Learning rate scheduler	Lambda Learning rate policy

5.5 Consistency in training results

We never managed to get a stable training using the Pix2Pix GAN training scheme. The reason is that the discriminator learns much faster than the generator models and the gradient vanishes. In other words, the generator never learns to fool the discriminator. Different approaches were tried in decreasing the discriminator performance and increasing the performance of the generator. Such as, tuning the learning rate, the learning rate policy, and updating the generator many times before updating the discriminator once (and more).

It would be interesting to see how much of an improvement the model could make, if we had managed to tune the hyperparameters such that the Pix2Pix GAN training becomes stable. Which might be possible given more time.

In short, the results were pretty consistent given that the discriminator loss reaches close to zero as the generator loss converges to a value as well. But given that if the generator manages to fool the discriminator, we might expect different results.

5.6 Training time

Each model took around 4-8 h to train, depending on when the validation loss converged and stopped improving. In other words, we have 4 models which took around 24 h to train in total. The hardware used was the NVIDIA A100 40 GB RAM.

6 Analysis of Model Performance/Output

During the competition, we have focused on finding loss functions that improve the general task of image-image translation and the problems that arise with this type of problem (high-frequency loss, intensity and others), and thus we never implemented or adjusted our loss function to better accomodate the specific Cellprofiler pipeline. When evaluating the results

from our models, we have instead used mean absolute error pixel-wise over the images, which is part of the evaluation pipeline.

6.1 Post processing

When we had our models fully trained, we still had problems with the intensities in the generated images. Because of this, we decided to implement histogram matching to the generated outputs. The data flow for doing this is visualised in Figure 6.1 below.

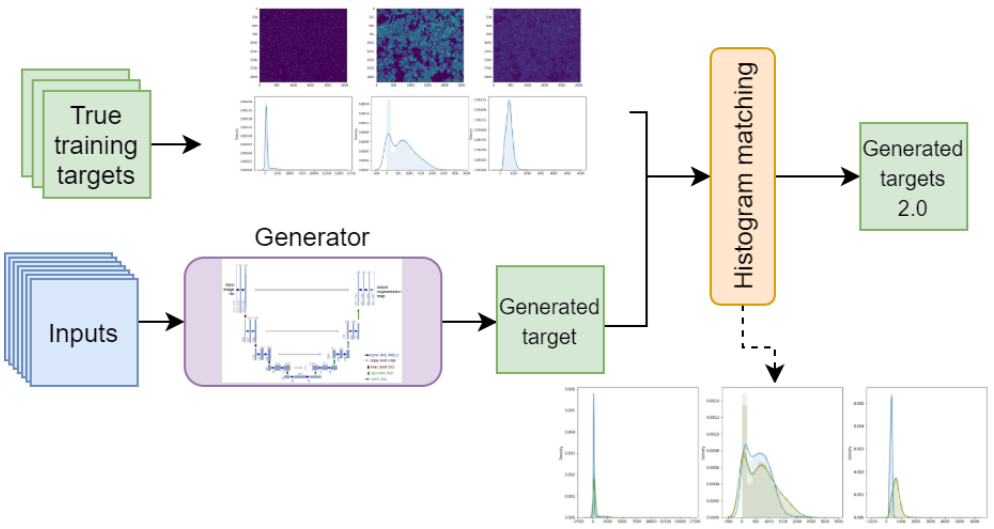


Figure 6.1: Flow for histogram matching.

For each magnification and target, a pixel distribution is computed. For each generated image, the histogram is then matched to the specific magnification and target that the generated image corresponds to.

When adding histogram matching to our pipeline, we found that the MAE decreased by factors up to 15.

7 Model Execution End-to-End

To run inference with our models, we have prepared a notebook under `AZHackathon/notebooks/End-to-endpipeline.ipynb` where we detail the data preparation and inference given the raw data. If you wish to run a model with inference, simply add the `--verbose` argument to the `predict.py` calls in the notebook!

For a training pipeline, we would suggest that you follow the following steps:

1. Start by processing your raw input data by running the script `prepare_training_data.py`. At the bottom of the script, you can define the paths to your raw data and where you want to store the training data. Please make sure that you follow how the already defined paths are structured - there will be a middle step folder created.

2. Follow this with training your first model! You will start with training the segmentation model for the nuclei masks by running the `train_segmentation.py` script. Masks were created with the maskSLIC algorithm in the previous step, and now you can use these for the training. If you did not change the paths in the previous script (except for the one to your raw data), you should not have to change anything in this script. Otherwise, you may have to change the paths.
3. Now you have the GAN models left! Here you have a multitude of parameters that you can tune directly from the command line. You run the script `train_pix2pix.py` with arguments that fit your data. Please note that the A_1 model will need 8 channels as input, whereas the other two targets only need 7. This is because of the masks from step 2.
4. This is of course followed by predicting on your models. Use the `predict.py` script. If you wish to run a model with inference, simply add the `--verbose` argument to the `predict.py` script!

REFERENCES

E. Khvedchenya V. I. Iglovikov A. Buslaev, A. Parinov and A. A. Kalinin. Albumentations: fast and flexible image augmentations. *ArXiv e-prints*, 2018.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL `http://arxiv.org/abs/1512.03385`.

Benjamin Irving. SLIC in a defined mask with applications to medical imaging. *CoRR*, abs/1606.09518, 2016. URL `http://arxiv.org/abs/1606.09518`.

Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks, 2018.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015. URL `http://arxiv.org/abs/1505.04597`.

Pavel Yakubovskiy. Segmentation models pytorch. `https://github.com/qubvel/segmentation_models.pytorch`, 2020.