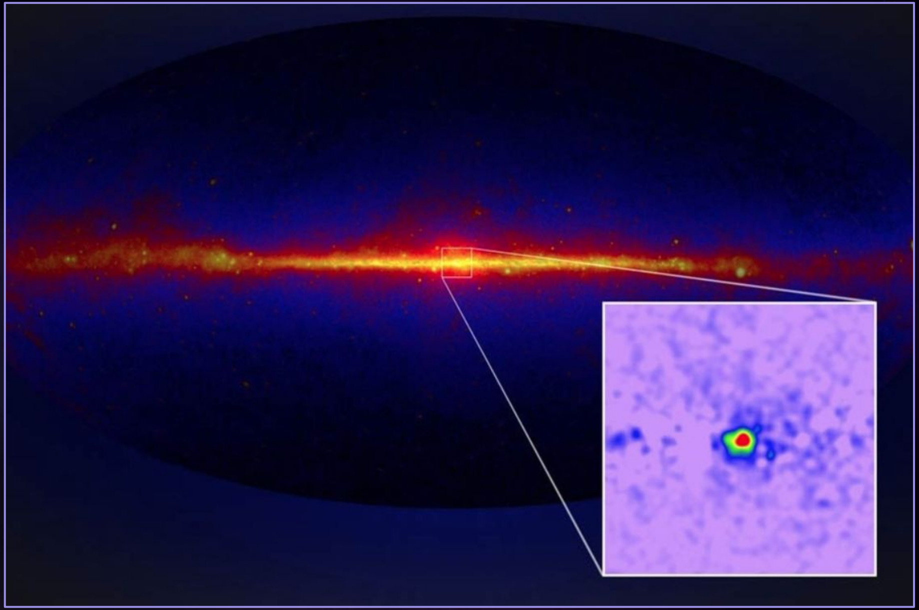
Applying ML on simulations of the Milky Way

Clotilde, Kathrine, Frederikke and Sophia

AML final project 14.06.2023

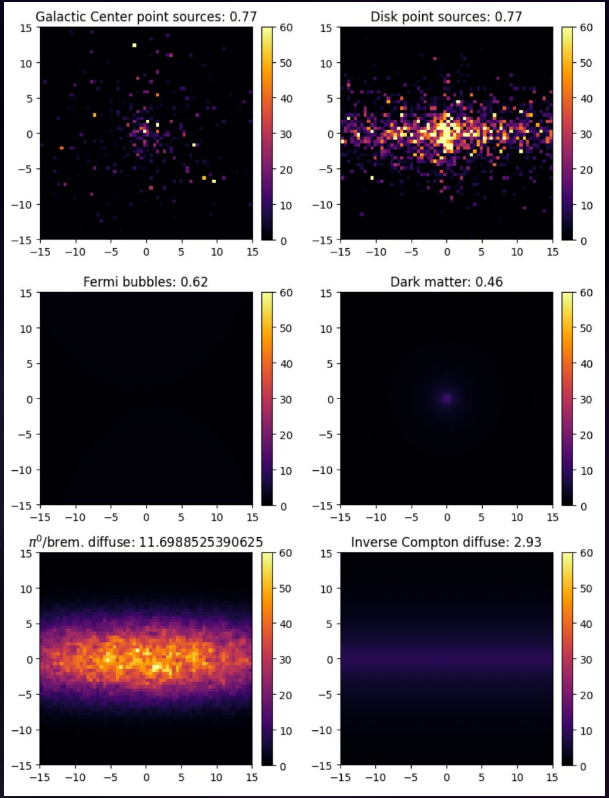
Introduction

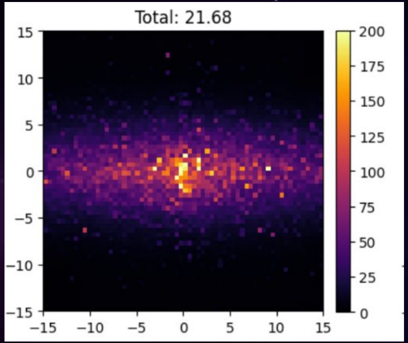


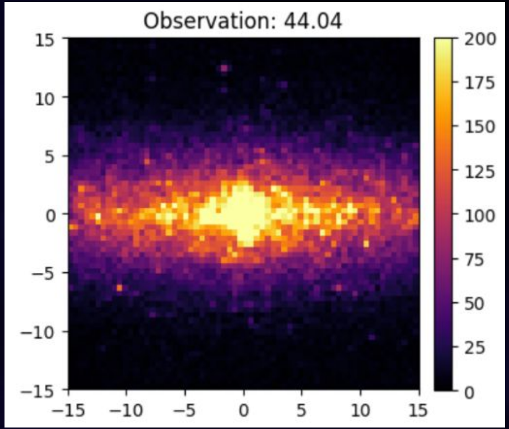
Simulation: Infer dark matter from 2D images of galactic center

Our goal: Apply various ML methods on the images

Introduction: Data







64 x 64

Simulator made by Adam Coogan, Montréal University

Introduction: Methods

Part a

Parameter estimation using **CNNs**

Part b

Semantic segmentation using **U-net**

Part c

Denoising images using **autoencoding**

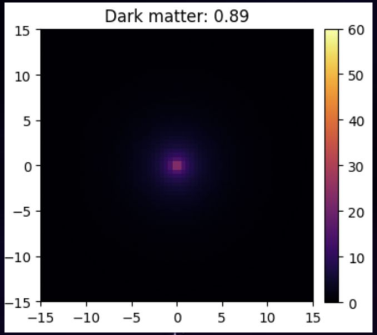
Part a: **Method**

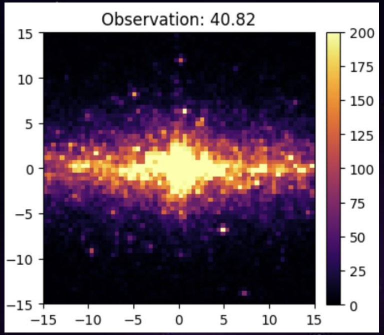
Single output CNN

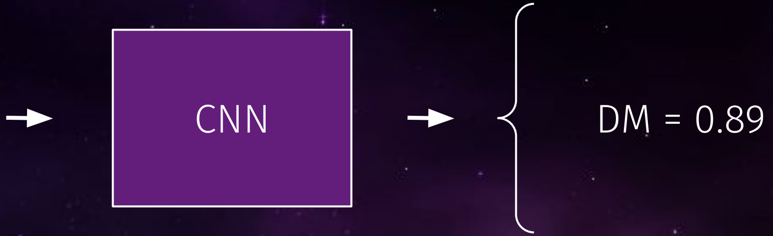
Parameter

Input

Output



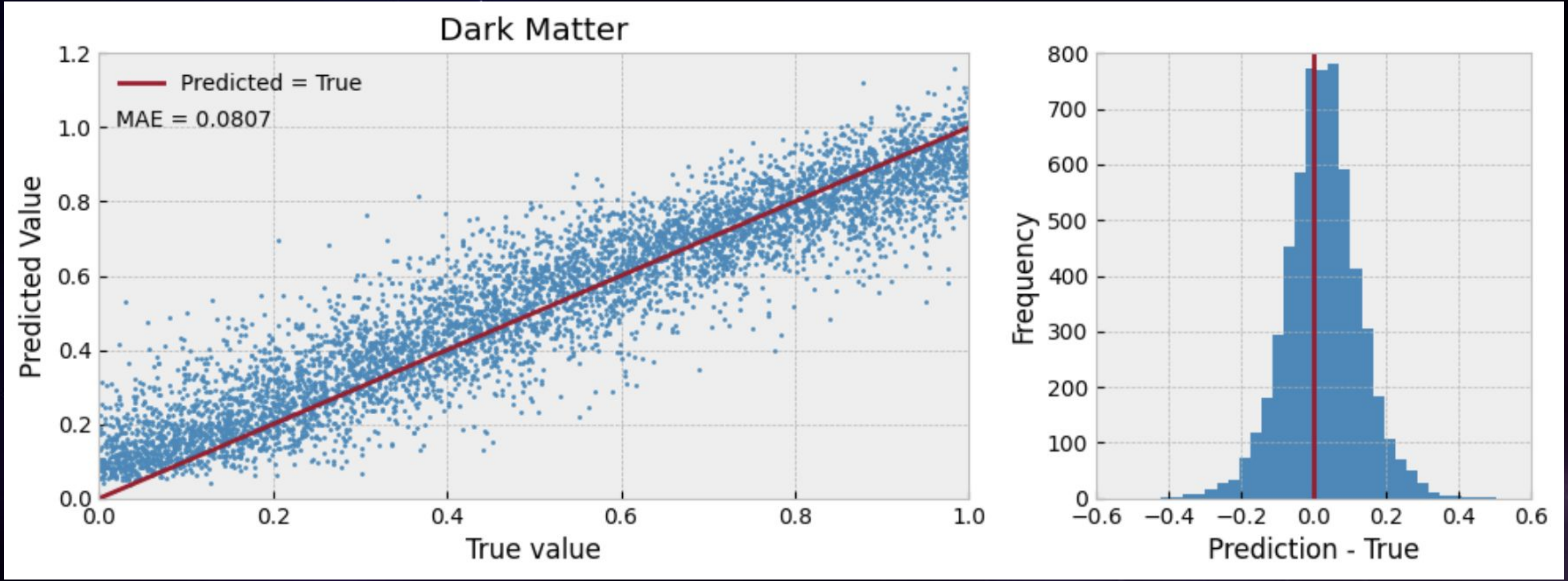




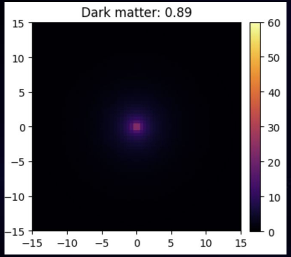
Goal: Estimate the amount of dark matter with a CNN

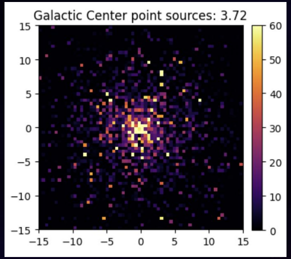
Part a: **Result**

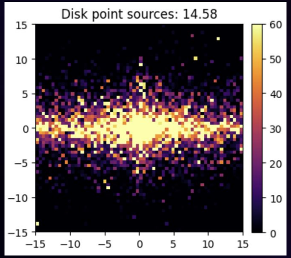
Single output CNN



Parameters

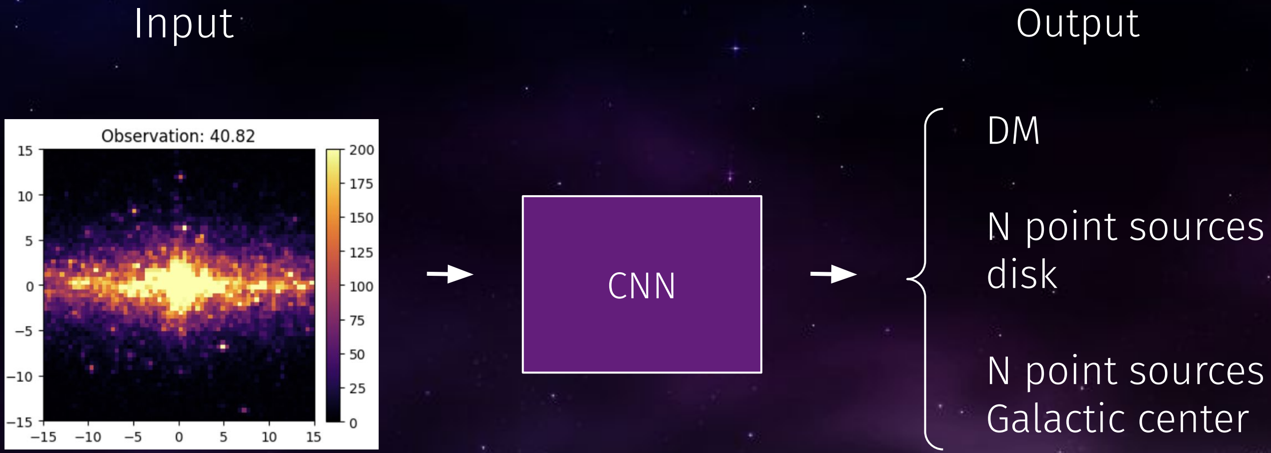






Part a: **Method**

Multiple output CNN



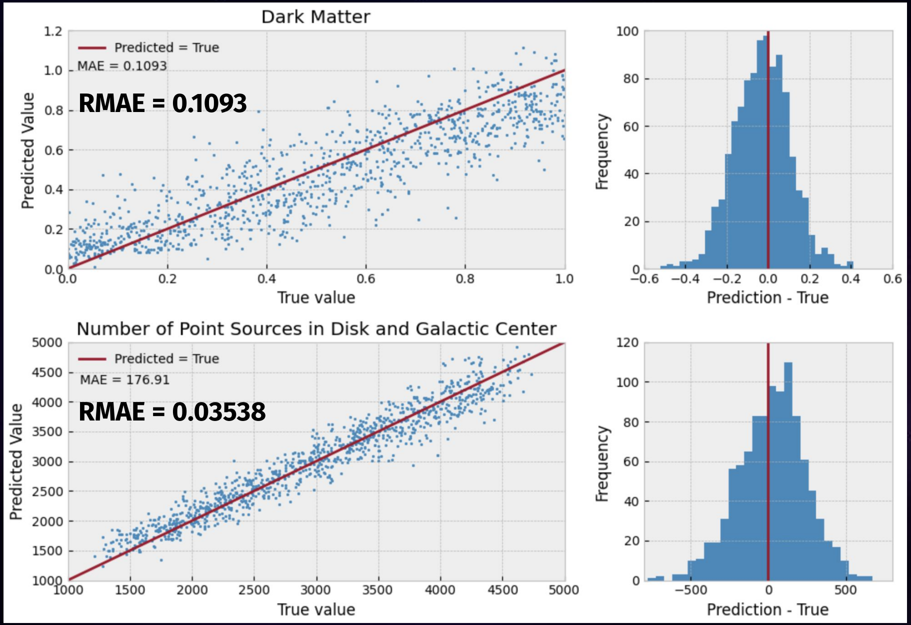
Goal: Estimate the amount of dark matter and the number of points sources with a CNN

Part a: **Result**

Multiple output CNN

Two outputs

DM & total number of point sources



Part a: **Result**

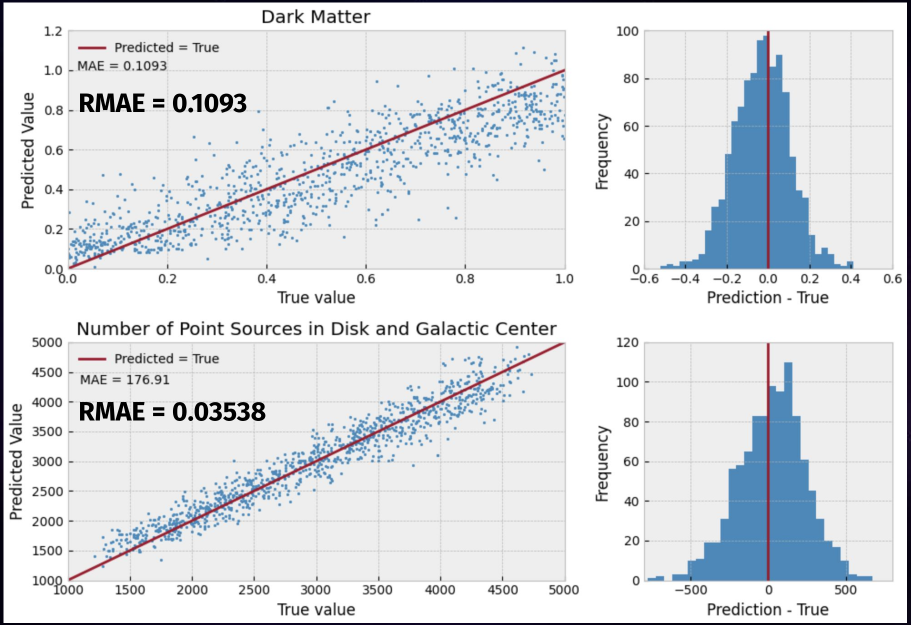
Multiple output CNN

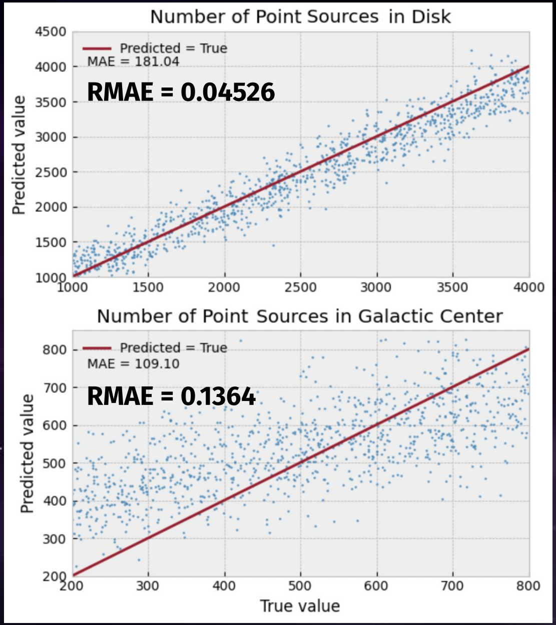
<u>Two outputs</u>

DM & total number of point sources

<u>Three outputs</u>

(DM) & separate estimates of point sources





Part a: **Result**

Multiple output CNN

Table with mean relative errors

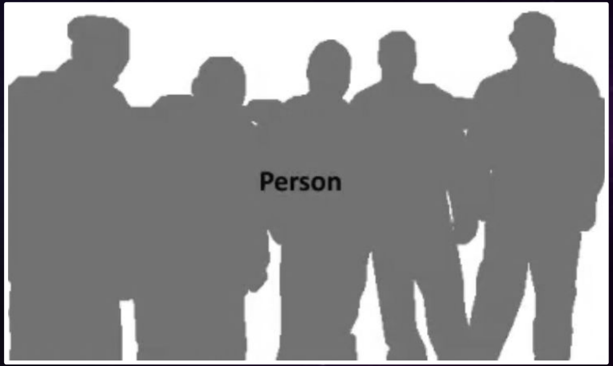
N outputs / parameter	DM	N PS total	N PS disk	N PS center
One	0.0807	-	-	-
Two	0.1093	0.0354	-	-
Three	0.1175	-	0.0453	0.1364

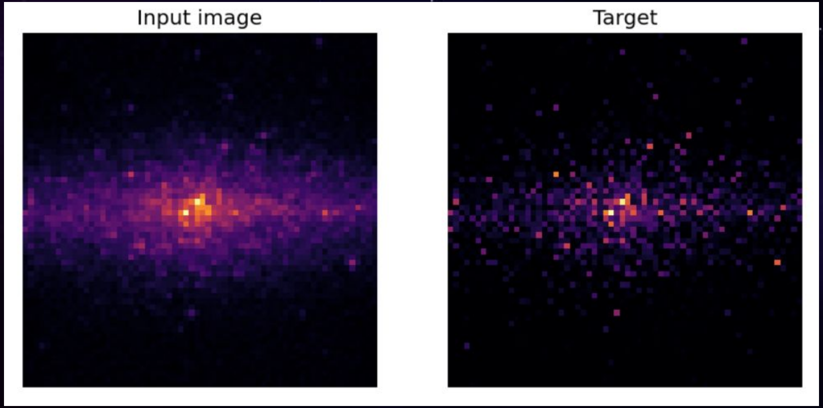
DM = dark matter, N = number, PS = point sources

Part b: **Method**

Semantic Segmentation with U-net

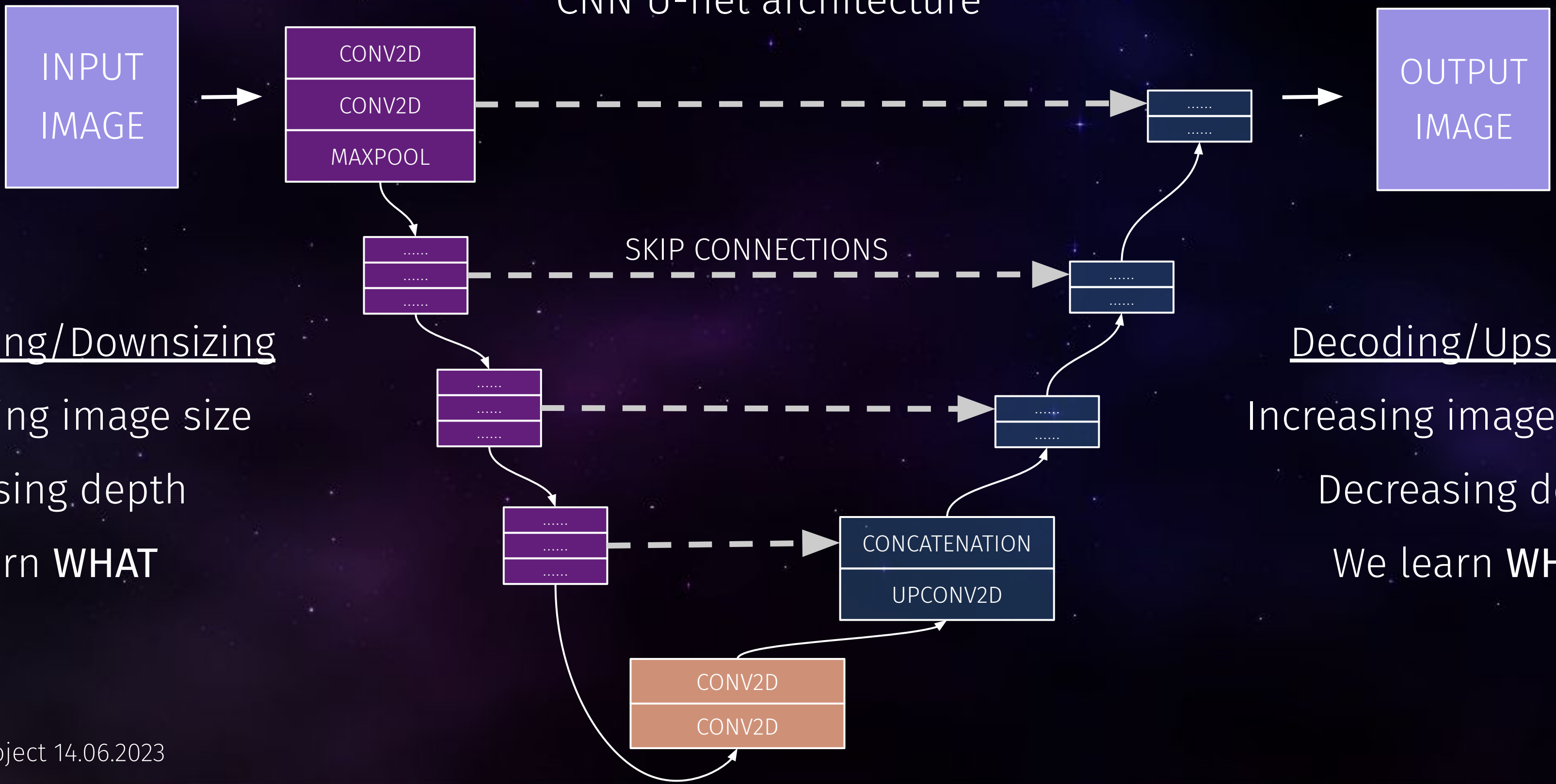
Goal: Detect point sources by categorizing each pixel in an image into a class (y/n)





Part b: **Method**

CNN U-net architecture



<u>Encoding/Downsizing</u>

Reducing image size

Increasing depth

We learn **WHAT**

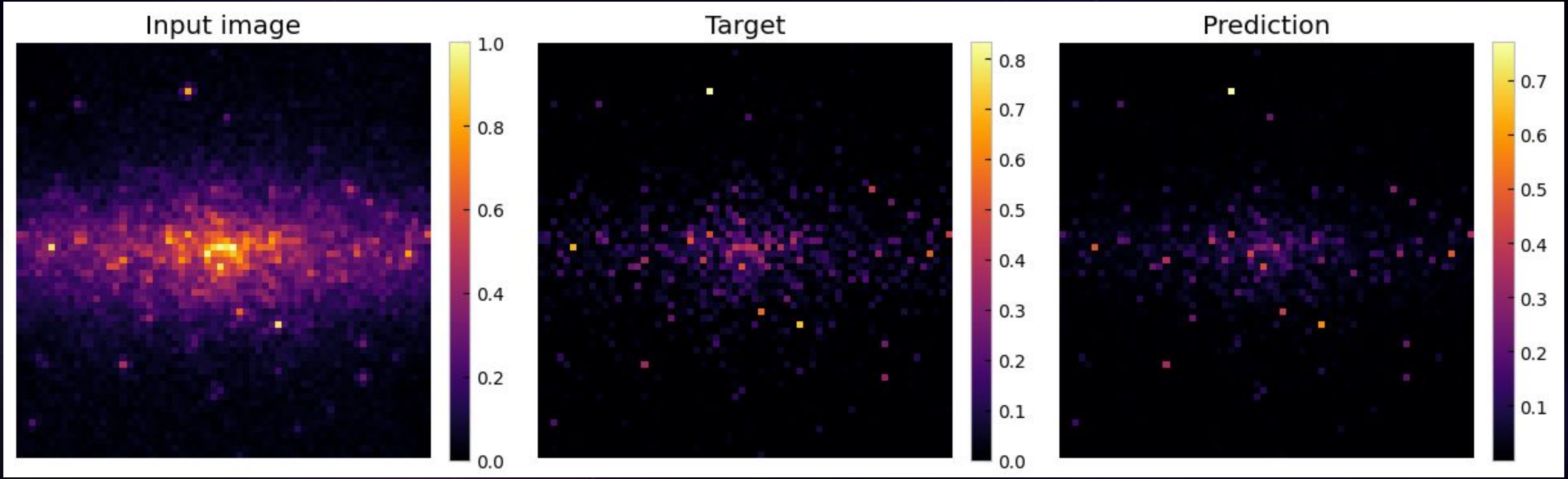
<u>Decoding/Upsizing</u>

Increasing image size

Decreasing depth

We learn **WHERE**

Part b: **Result**



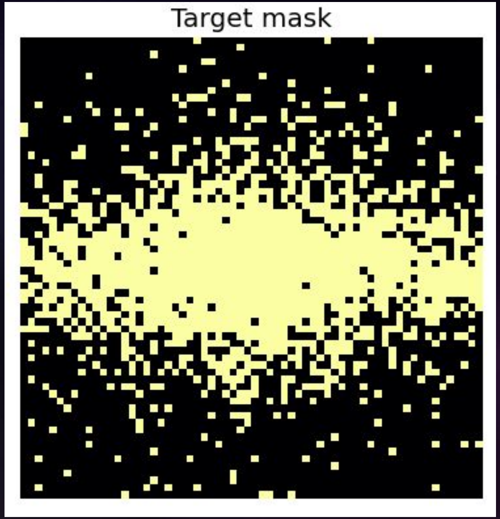
Normalized flux

Probability

Part b: **Result**

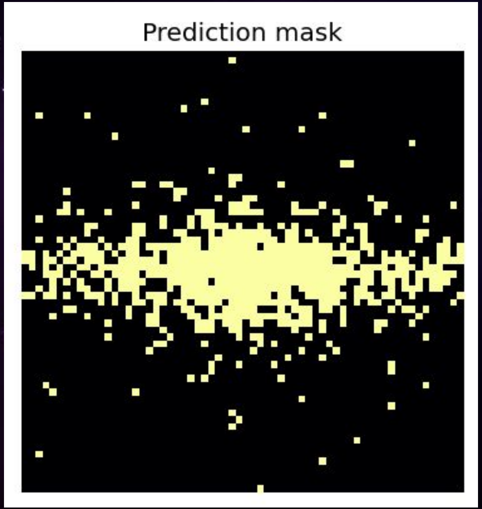
Target 1: ALL

All point source pixels



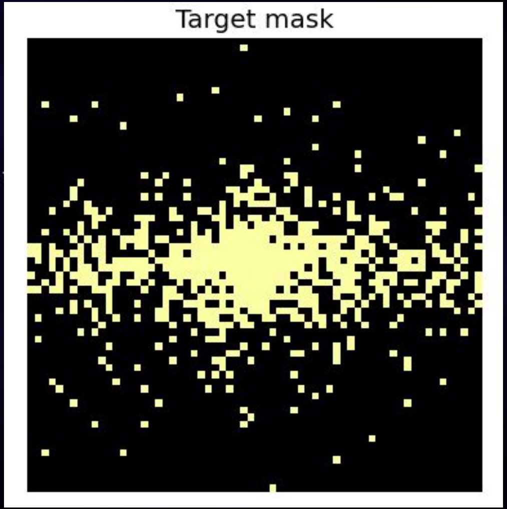
Prediction mask

Pixels above cut value 0.02



Target 2: SELECTED

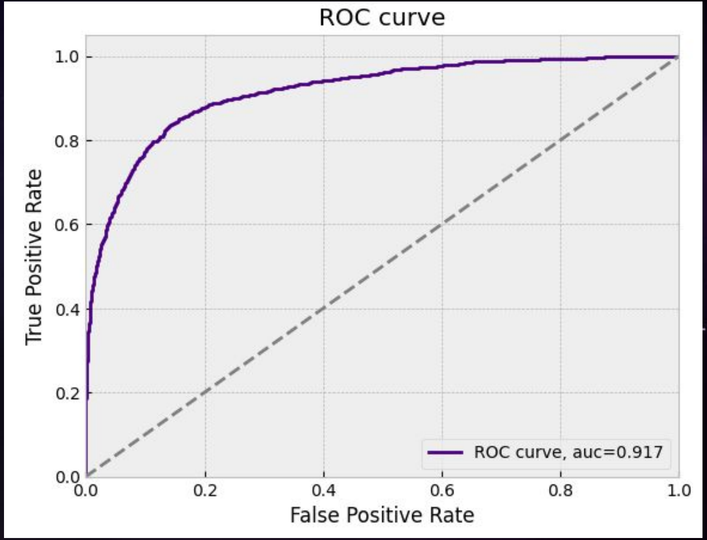
N brightest point sources



Part b: **Result**

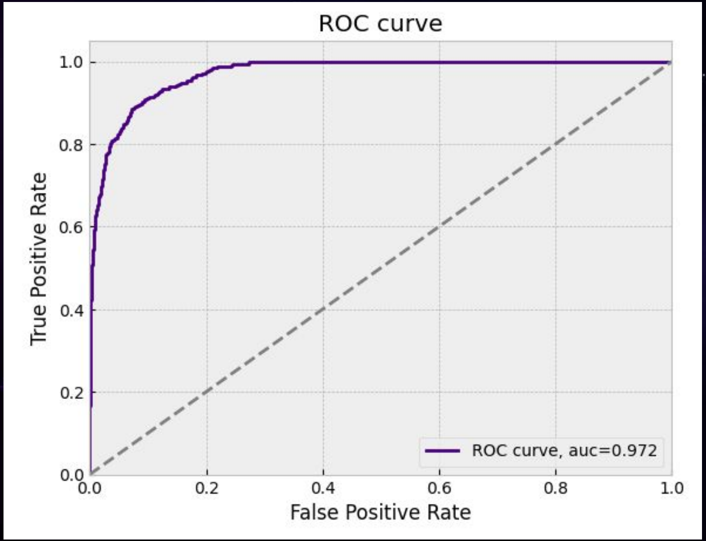
Target 1: ALL

AUC = 0.92



Target 2: SELECTED

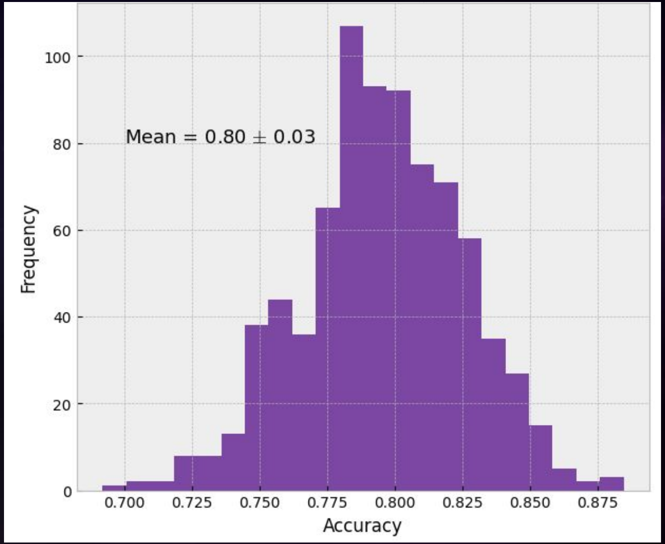
AUC = 0.97



Part b: **Result**

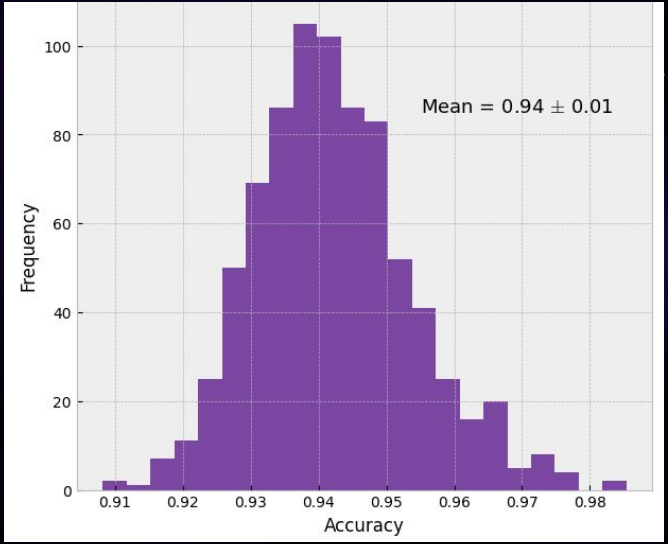
Target 1: ALL

Provides OK overall picture



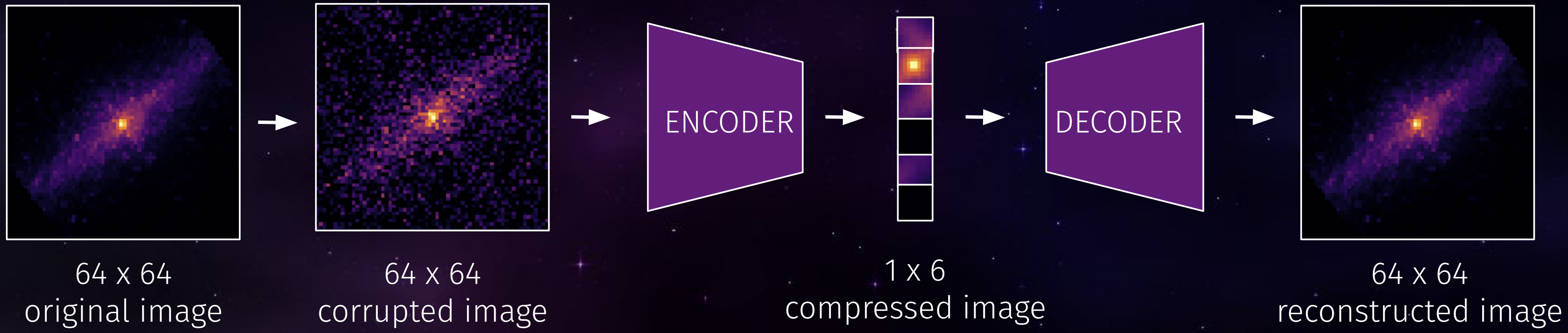
Target 2: SELECTED

Good at identifying the brightest sources



Part c: **Method**

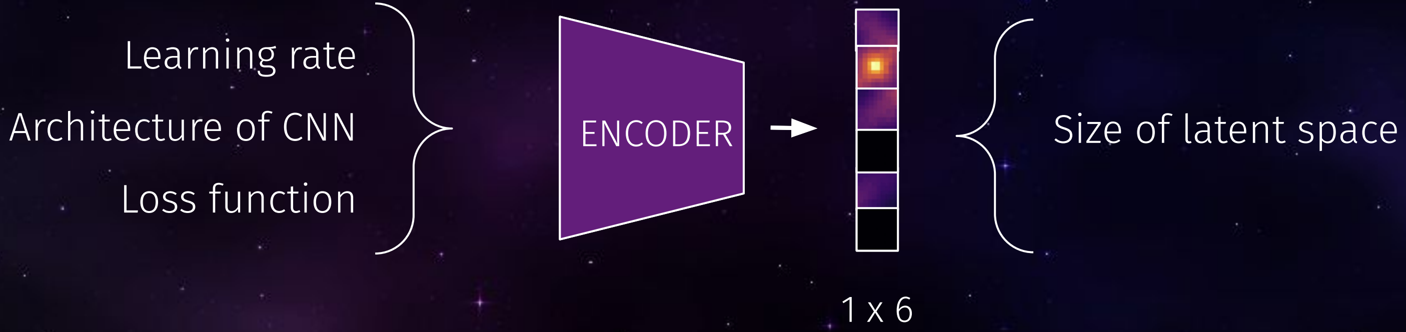
Goal: Denoise corrupted images using autoencoding



Part c: **Method**

Goal: Denoise corrupted images using autoencoding

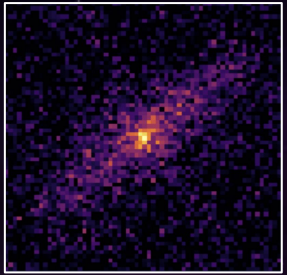
Parameters to optimize:



Part c: **Method**

Goal: Denoise corrupted images using autoencoding

Type of noise:

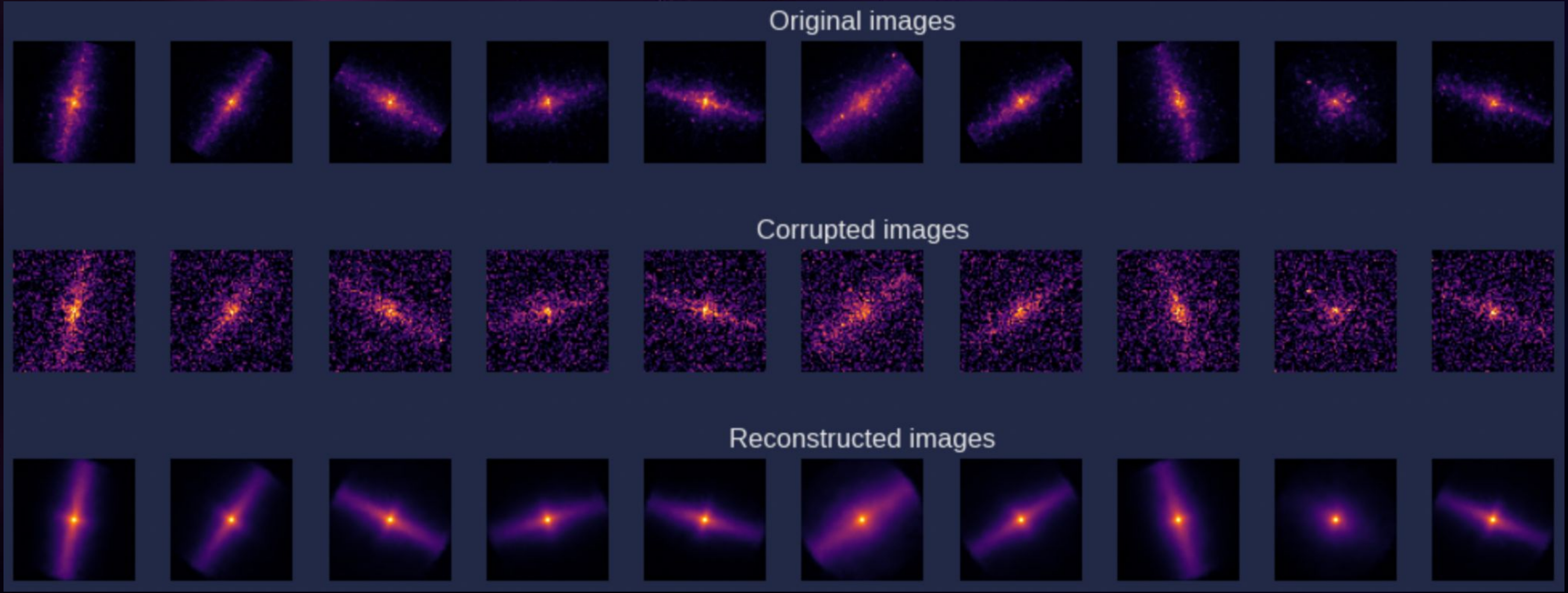


- Gaussian
- Poisson
- Masked (fraction of pixels set to 0)
- Salt&Pepper (fraction of pixels set to 0 or 1)

Part c: **Result**

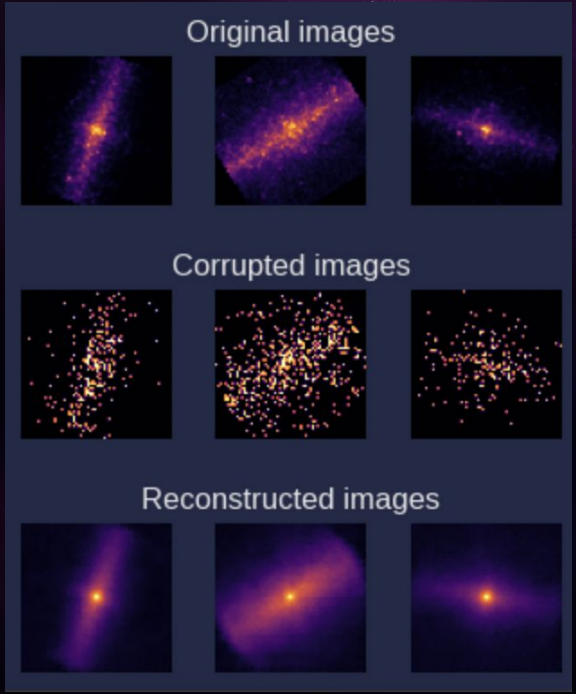
Epoch 20 → Minimum reached: MSE = 0.0011

Gaussian noise

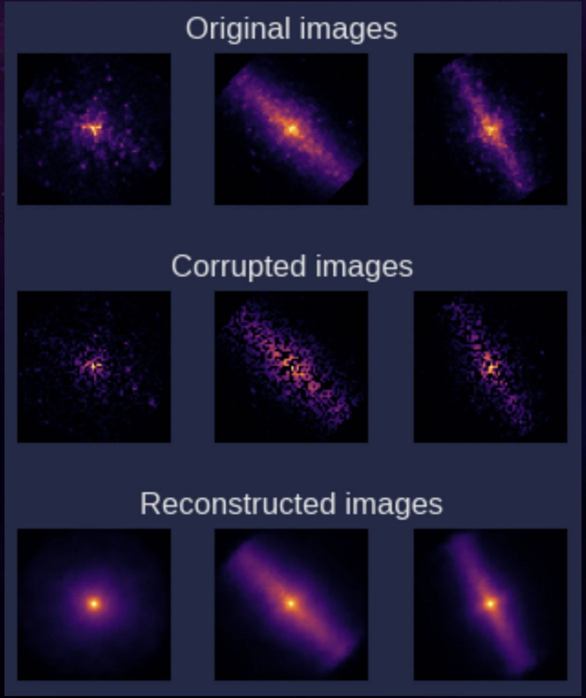


Part c: **Result**

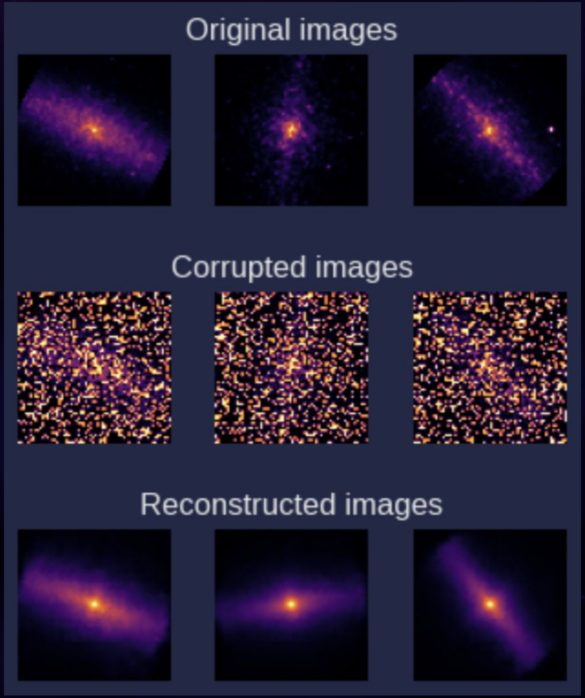
Poisson MSE: 0.0012



Masking 50% MSE: 0.0010



Salt&Pepper 50% MSE: 0.0019



Conclusion

Part a	Part b	Part c
Parameter estimation using CNNs	Semantic segmentation using U-net	Denoising images using autoencoding

Part a — min MAE values

DM:	**0.0807**
PS (total):	**0.0354**
PS (disk):	**0.0453**
PS (GC):	**0.1364**

Part b

80% accuracy for ALL point sources

94% accuracy for BRIGHTEST point sources

Part c — min MSE values

Gaussian:	**0.0011**
Poisson:	**0.0012**
Masked:	**0.0010**
S&P:	**0.0019**

Thank you for listening :-)

Appendix 0
Introduction to topic and data

Contributions

We all contributed evenly in this project.

The main responsibles in the different parts were: a) Sophia and Frederikke, b) Clotilde and Kathrine and c) Sophia

Topic

Our goal is to infer dark matter and other astrophysical contributions from simulated images of the gamma-ray sky around the Galactic Center. We aim at applying several algorithms reviewed in class but not utilized earlier in the Initial Project. We approach the dataset with several different goals in mind:

A. To predict parameters in the images using CNN.
B. To classify and locate sources in the image using a modified U–net CNN architecture.
C. To denoise the images using auto-encoding.

Data simulator

The data used in the project is generated images of the Galactic Center, generated from a simulator created by Adam Coogan in connection with a summer school Sophia attended at Montréal University in 2022.

Below is a description of the different components sampled in the simulator:

- Point sources distributed symmetrically around the Galactic Center (n_ps_gc), with related fluxes describing the point source contribution in each pixel (flux_ps_gc). The positions and fluxes of the point sources are randomly sampled for each run of the simulator.
- Point sources distributed in the galactic disk (n_ps_disk), with related fluxes describing the point source contribution in each pixel (flux_ps_disk). The positions and fluxes of the point sources are randomly sampled for each run of the simulator.
- Dark matter emission spherically-symmetric and sharply peaked towards the Galactic Center (A_dm).
- Emission tracing the gas distribution, arising from e.g. $\pi^0\rightarrow\gamma\gamma$ decay and bremsstrahlung (A_pi0).
- Inverse Compton emission from high-energy cosmic rays Compton upscattering low-energy cosmic microwave background (CMB) and starlight photons to higher energies (A_ic).
- Fermi bubbles which consists of two huge, faint lobes extending far above and below the galactic plane (A_bubbles).

The simulator generates an observation by constructing a grid of pixels and sampling the components mentioned above. Other parameters are defined such as the observation size, distance to the galactic center, pixel resolution, dimensions of the disk, smoothing factor and containment radius. It then applies a point spread function (PSF) that slightly blurs the image. Finally it samples the observation from a Poisson distribution with mean equal to the blurred image.

Appendix 1
Predicting parameters using CNNs

CNNs

Goal

The goal was to make a CNN that could predict the amount of dark matter parameter in a simulated image. The idea is that if we succeed in creating this algorithm based on simulations, one could feed the algorithm an image of the real Universe from where the algorithm could predict the amount of dark matter. Unfortunately, we don't have an image of the real Universe that corresponds to the simulations. After creating a solid CNN for predicted the dark matter parameter, we expanded the CNN to be able to predict other parameters too.

Normalization

The dark matter parameter is already normalized to be between 0 and 1. For the CNN that predicts more than one parameter, the other parameters were normalized to be between 0 and 1 in order for the loss function to weigh the accuracy of all parameters equally.

Size of dataset (one output)

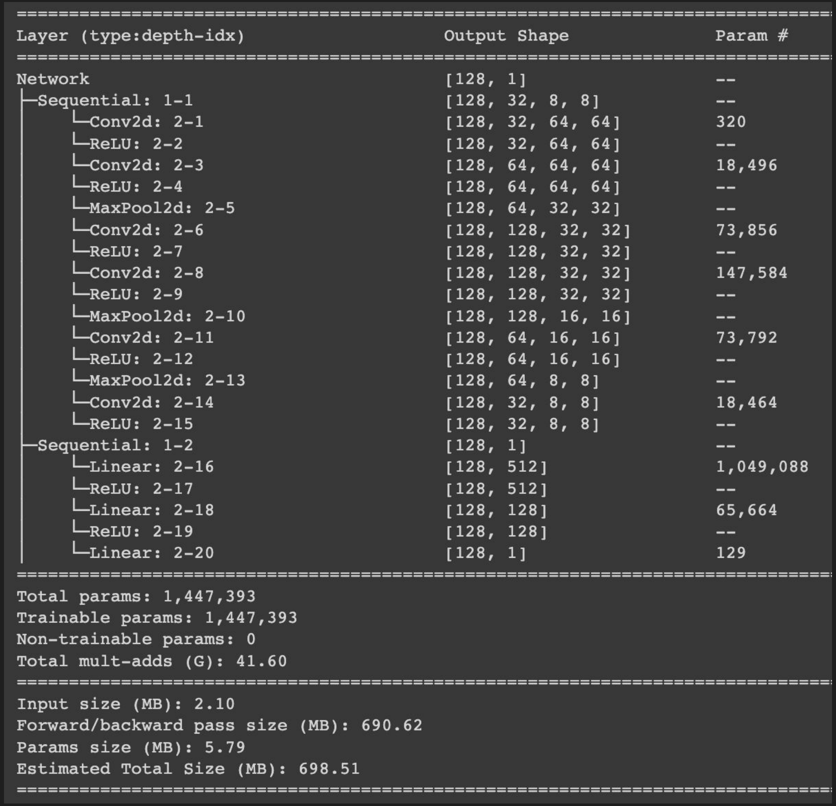
Training: 8192 simulated images with size 1x64x64
Validation: 1966 simulated images with size 1x64x64
Testing: 3072 simulated images with size 1x64x64

Size of dataset (multiple outputs)

Training: 8192 simulated images with size 1x64x64
Validation: 1024 simulated images with size 1x64x64
Testing: 1024 simulated images with size 1x64x64

** We used Google Colab (free version) in all parts of this project and therefore the size of the generated datasets are limited to the RAM that follows when using Googles GPU **

CNNs: Architecture and HP optimization



```
=====================================================================================
Layer (type:depth-idx)                        Output Shape              Param #
=====================================================================================
Network                                       [128, 1]                  --
├─Sequential: 1-1                             [128, 32, 8, 8]           --
│    └─Conv2d: 2-1                            [128, 32, 64, 64]         320
│    └─ReLU: 2-2                              [128, 32, 64, 64]         --
│    └─Conv2d: 2-3                            [128, 64, 64, 64]         18,496
│    └─ReLU: 2-4                              [128, 64, 64, 64]         --
│    └─MaxPool2d: 2-5                         [128, 64, 32, 32]         --
│    └─Conv2d: 2-6                            [128, 128, 32, 32]        73,856
│    └─ReLU: 2-7                              [128, 128, 32, 32]        --
│    └─Conv2d: 2-8                            [128, 128, 32, 32]        147,584
│    └─ReLU: 2-9                              [128, 128, 32, 32]        --
│    └─MaxPool2d: 2-10                        [128, 128, 16, 16]        --
│    └─Conv2d: 2-11                           [128, 64, 16, 16]         73,792
│    └─ReLU: 2-12                             [128, 64, 16, 16]         --
│    └─MaxPool2d: 2-13                        [128, 64, 8, 8]           --
│    └─Conv2d: 2-14                           [128, 32, 8, 8]           18,464
│    └─ReLU: 2-15                             [128, 32, 8, 8]           --
├─Sequential: 1-2                             [128, 1]                  --
│    └─Linear: 2-16                           [128, 512]                1,049,088
│    └─ReLU: 2-17                             [128, 512]                --
│    └─Linear: 2-18                           [128, 128]                65,664
│    └─ReLU: 2-19                             [128, 128]                --
│    └─Linear: 2-20                           [128, 1]                  129
=====================================================================================
Total params: 1,447,393
Trainable params: 1,447,393
Non-trainable params: 0
Total mult-adds (G): 41.60
=====================================================================================
Input size (MB): 2.10
Forward/backward pass size (MB): 690.62
Params size (MB): 5.79
Estimated Total Size (MB): 698.51
=====================================================================================
```

One output

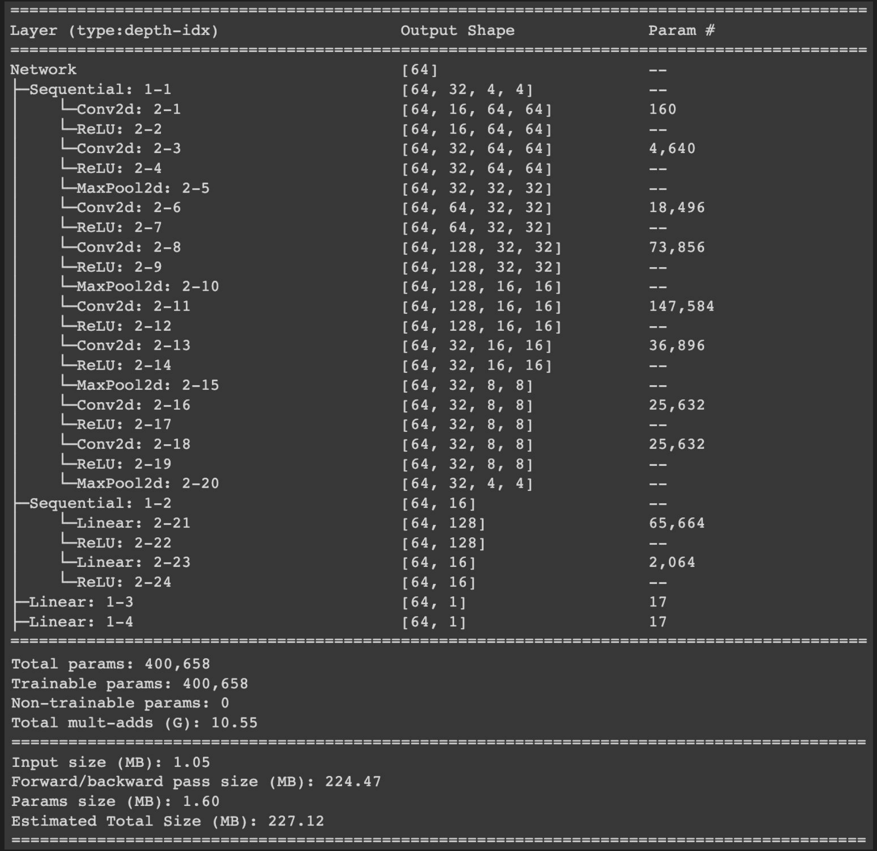
The network architecture is build from a set of convolutional layers followed by fully connected layers. Both using Sequential modules from PyTorch.

The convolutional layers uses Conv2d layers, ReLU activation functions, and MaxPool2d layers. The fully connected layers consists of Linear layers, which perform matrix multiplication on the input and apply a linear transformation. The first Linear layer takes the flattened input and produces an output of size 512. The subsequent Linear layers reduce the size to 128 and then to 1, which is the final output size of the network.

Performing “architecture” and HP optimization we concluded:

- ReLU activation performed better than LeakyReLU
- Batch size: 64 (examined alternatives: 32 and 128)
- Learning rate: 0.001

CNNs: Architecture and HP optimization



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Network                                  [64]                      --
├─Sequential: 1-1                        [64, 32, 4, 4]            --
│    └─Conv2d: 2-1                       [64, 16, 64, 64]          160
│    └─ReLU: 2-2                         [64, 16, 64, 64]          --
│    └─Conv2d: 2-3                       [64, 32, 64, 64]          4,640
│    └─ReLU: 2-4                         [64, 32, 64, 64]          --
│    └─MaxPool2d: 2-5                    [64, 32, 32, 32]          --
│    └─Conv2d: 2-6                       [64, 64, 32, 32]          18,496
│    └─ReLU: 2-7                         [64, 64, 32, 32]          --
│    └─Conv2d: 2-8                       [64, 128, 32, 32]         73,856
│    └─ReLU: 2-9                         [64, 128, 32, 32]         --
│    └─MaxPool2d: 2-10                   [64, 128, 16, 16]         --
│    └─Conv2d: 2-11                      [64, 128, 16, 16]         147,584
│    └─ReLU: 2-12                        [64, 128, 16, 16]         --
│    └─Conv2d: 2-13                      [64, 32, 16, 16]          36,896
│    └─ReLU: 2-14                        [64, 32, 16, 16]          --
│    └─MaxPool2d: 2-15                   [64, 32, 8, 8]            --
│    └─Conv2d: 2-16                      [64, 32, 8, 8]            25,632
│    └─ReLU: 2-17                        [64, 32, 8, 8]            --
│    └─Conv2d: 2-18                      [64, 32, 8, 8]            25,632
│    └─ReLU: 2-19                        [64, 32, 8, 8]            --
│    └─MaxPool2d: 2-20                   [64, 32, 4, 4]            --
├─Sequential: 1-2                        [64, 16]                  --
│    └─Linear: 2-21                      [64, 128]                 65,664
│    └─ReLU: 2-22                        [64, 128]                 --
│    └─Linear: 2-23                      [64, 16]                  2,064
│    └─ReLU: 2-24                        [64, 16]                  --
├─Linear: 1-3                            [64, 1]                   17
├─Linear: 1-4                            [64, 1]                   17
==========================================================================================
Total params: 400,658
Trainable params: 400,658
Non-trainable params: 0
Total mult-adds (G): 10.55
==========================================================================================
Input size (MB): 1.05
Forward/backward pass size (MB): 224.47
Params size (MB): 1.60
Estimated Total Size (MB): 227.12
==========================================================================================
```

Multiple output

The architecture for the CNN with multiple outputs is deeper and more narrow compared to the CNN with a single output. It has approximately ⅓ number of parameters. We tried wider networks with more parameters and the result was approximately the same and therefore not worth the extra computational expense.

The same network architecture was used for the model with two and three outputs. One could expect the results of the three parameters would increase if the network was made even deeper.

CNNs: Evaluation

Own evaluation
Overall we were very impressed with the CNNs and their ability to predict the dark matter parameter as the amount of dark matter in the images is impossible to asses with the human eye (shout out to ML!). We also succeeded in creating a model that could determine the total number of points sources with a satisfying MAE score. We were, however, disappointed that the model could not predict the number of points sources in the Galactic center decently. We believe that one of the reasons that the CNN struggles with the number of points sources in the Galactic center is because this number is between 200 and 800 while the number of point sources in the disk is between 1000 and 4000. The intensity of the point sources in the disk is therefore a lot brighter.

Next steps / Improvements
Buying more RAM on google Colab would allow us to generate more training data which most likely would improve our results and maybe even be enough to reach a decent MAE score for the number of points sources in the Galactic center. Alternatively, one could build an ever deeper network and tune the architecture even more.

Appendix 2
Semantic segmentation using U-net

References for Appendix 2:
https://aditi-mittal.medium.com/introduction-to-u-net-and-res-net-for-image-segmentation-9afcb432ee2f
https://medium.com/analytics-vidhya/what-is-unet-157314c87634
https://medium.datadriveninvestor.com/an-overview-on-u-net-architecture-d6caabf7caa4

U-net

Goal

The goal was to construct a Fully Convolutional Neural Network using the U-net architecture, in order to categorize which pixels in the simulated images originate from source points. This could eg. be useful for astrophysicists, for identifying point sources from telescope images. In this case it is used as part of the problem of distinguishing light from dark matter and point sources.

Normalization

All the simulated images were normalized by dividing the pixel values in each image with the maximum pixel value in that image. Alternative types of normalization were also tested e.g. dividing all pixel values in all images with the maximum pixel value across all images. Since the maximum value varies by approximately a factor of 100 in a dataset of N images the latter method did not give optimal results, and was therefore not used.

Size of dataset

Training: 6592 simulated images with size 1x64x64
Validation: 800 simulated images with size 1x64x64
Testing: 800 simulated images with size 1x64x64

U-net: Architecture

The U-net Convolutional Neural Network architecture consists of two symmetric paths: An encoder and a decoder path separated by a bridge (‘bottle-neck layer’). An overview of layer type and output shape for each layer is presented in two slides.

The encoder (contraction) path consists of four blocks and is used to capture the context of the image. Each block gets an input, applies two 3x3 convolutional Leaky Relu layers with a Dropout layer in between, batch normalization, Leaky ReLU activation functions followed by a 2x2 maxpooling layer.

The bridge connects the encoder and decoder paths. It consists of two 3x3 convolutional Leaky Relu layers separated by batch normalisation and a leaky relu activation function.

The decoder (expansion) path also consists of four blocks and enables precise locations of pixels. Each block gets an input, applies a 3x3 transposed convolutional layer, then includes a concatenation with the correspondingly cropped feature map from the contracting path (skip-connections), batch normalization, and a Leaky ReLU activation function. The output of the last decoder block passes through a 1x1 convolutional layer with a sigmoid function, to give the segmentation mask representing the pixel-wise classification.

U-net: Architecture

We wish to highlight the use of skip-connections between encoder and decoder blocks as this is an important feature of the network: By concatenating the output of the transposed convolution layers with the feature maps from the encoder at the same level, the algorithm combines location information from the downsampling path with the contextual information in the upsampling path, to finally obtain general information combining localization and context, to predict a good segmentation map. So the skip-layers provide information on features from earlier layers that are sometimes lost due to the depth of the network.

Starting with the architecture from the original U-net and by doing 'architecture optimization' we concluded that

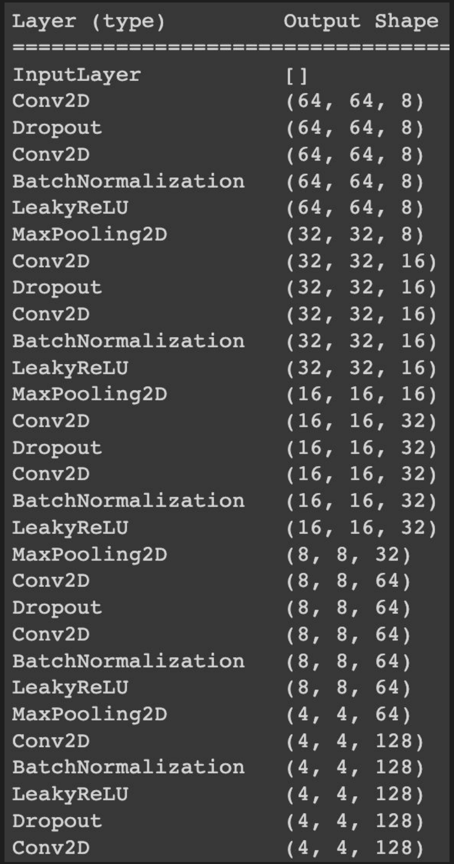
- Leaky ReLu improved the result compared to ReLu
- Dropout layers was a nice feature to introduce, to prevent overfitting by ignoring randomly selected pixels
- Batch normalization layers are included, because they make the network more steady while training

Why choose the U-net architecture for this problem?
The U-net architecture is a specific kind of CNN originally developed for biomedical image segmentation. It is known to be one of the best networks for fast and precise segmentation of images, designed to learn from fewer training samples. This was beneficial for us, as our training data was somewhat time-consuming to produce, took up a lot of memory, and we want the algorithm to work as fast as possible. Furthermore the U-net only contains convolutional layers and not dense layers, meaning it can accept images of any size (since only parameters to learn on convolution layers are size of kernel, which is independent from input image size), making it more generalizable.
There are several other alternatives to the U-net architecture for doing semantic segmentation, which might work just as well or even better than the algorithm we have worked with. This would be interesting to investigate in further work.

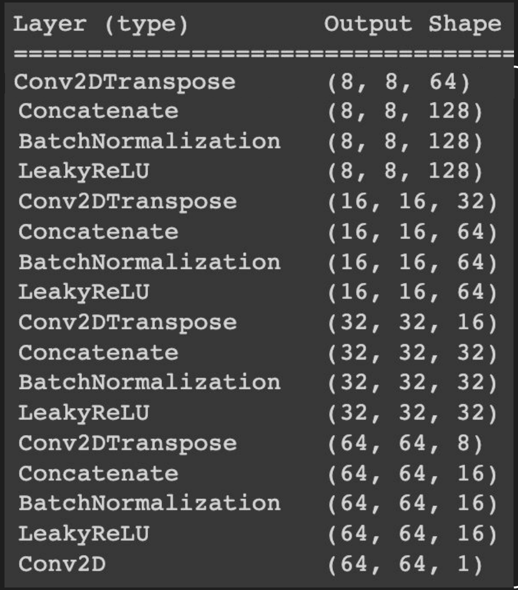
U-net: Architecture overview

Encoder

Bridge



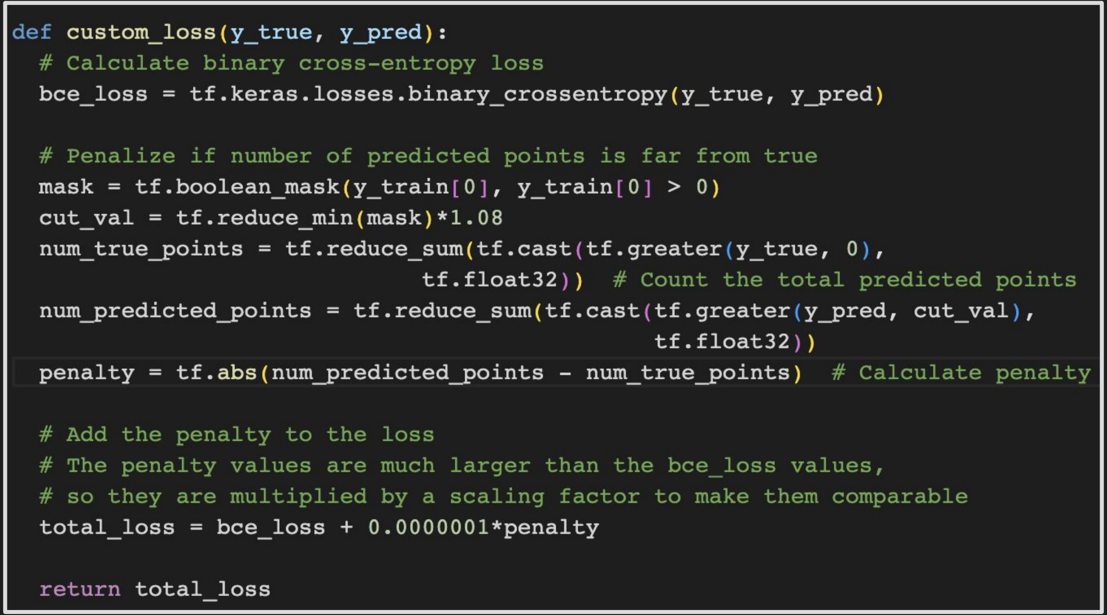
```
Layer (type)           Output Shape
====================================
InputLayer             []
Conv2D                 (64, 64, 8)
Dropout                (64, 64, 8)
Conv2D                 (64, 64, 8)
BatchNormalization     (64, 64, 8)
LeakyReLU              (64, 64, 8)
MaxPooling2D           (32, 32, 8)
Conv2D                 (32, 32, 16)
Dropout                (32, 32, 16)
Conv2D                 (32, 32, 16)
BatchNormalization     (32, 32, 16)
LeakyReLU              (32, 32, 16)
MaxPooling2D           (16, 16, 16)
Conv2D                 (16, 16, 32)
Dropout                (16, 16, 32)
Conv2D                 (16, 16, 32)
BatchNormalization     (16, 16, 32)
LeakyReLU              (16, 16, 32)
MaxPooling2D           (8, 8, 32)
Conv2D                 (8, 8, 64)
Dropout                (8, 8, 64)
Conv2D                 (8, 8, 64)
BatchNormalization     (8, 8, 64)
LeakyReLU              (8, 8, 64)
MaxPooling2D           (4, 4, 64)
Conv2D                 (4, 4, 128)
BatchNormalization     (4, 4, 128)
LeakyReLU              (4, 4, 128)
Dropout                (4, 4, 128)
Conv2D                 (4, 4, 128)
```



```
Layer (type)           Output Shape
====================================
Conv2DTranspose        (8, 8, 64)
 Concatenate           (8, 8, 128)
 BatchNormalization    (8, 8, 128)
 LeakyReLU             (8, 8, 128)
 Conv2DTranspose       (16, 16, 32)
 Concatenate           (16, 16, 64)
 BatchNormalization    (16, 16, 64)
 LeakyReLU             (16, 16, 64)
 Conv2DTranspose       (32, 32, 16)
 Concatenate           (32, 32, 32)
 BatchNormalization    (32, 32, 32)
 LeakyReLU             (32, 32, 32)
 Conv2DTranspose       (64, 64, 8)
 Concatenate           (64, 64, 16)
 BatchNormalization    (64, 64, 16)
 LeakyReLU             (64, 64, 16)
 Conv2D                (64, 64, 1)
```

Decoder

U-net: Optimization



```python
def custom_loss(y_true, y_pred):
  # Calculate binary cross-entropy loss
  bce_loss = tf.keras.losses.binary_crossentropy(y_true, y_pred)

  # Penalize if number of predicted points is far from true
  mask = tf.boolean_mask(y_train[0], y_train[0] > 0)
  cut_val = tf.reduce_min(mask)*1.08
  num_true_points = tf.reduce_sum(tf.cast(tf.greater(y_true, 0),
                                          tf.float32))  # Count the total predicted points
  num_predicted_points = tf.reduce_sum(tf.cast(tf.greater(y_pred, cut_val),
                                               tf.float32))
  penalty = tf.abs(num_predicted_points - num_true_points)  # Calculate penalty

  # Add the penalty to the loss
  # The penalty values are much larger than the bce_loss values,
  # so they are multiplied by a scaling factor to make them comparable
  total_loss = bce_loss + 0.0000001*penalty

  return total_loss
```

Loss function

We wrote a customized loss function to implement a penalty term that accounted for the algorithm detecting more/less point source pixels than the actual true amount (which we saw that it did).
The binary cross-entropy loss was used as a basis since our target is binary (point source or non-point source). The difference between i) the true amount of point sources and ii) the predicted amount of point sources with a score larger than 108% of minimum, was scaled by a small factor and added to the basis score, allowing us to penalize the algorithm for detecting the wrong amount of point sources. The value of 108% was optimized by trying values in teh range 105-120 %.

Hyperparameter optimization

Performing hyperparameter-optimization we furthermore found optimal values for

- Batch size: 128 (examined alternatives: 64 and 256)
- Learning rate: 0.001 (examined range: 0.00001 - 0.1)

U-net: Result evaluation

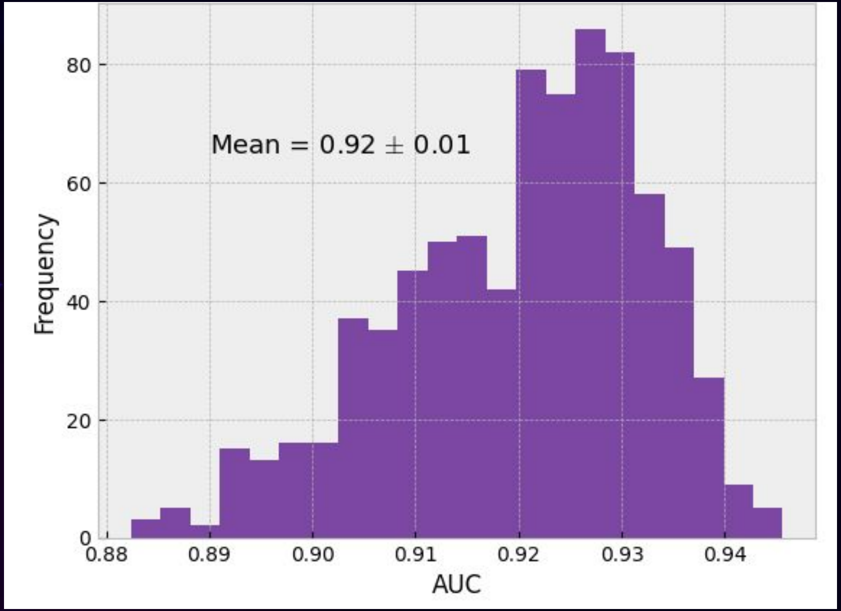
How do we evaluate the results?
The output of the algorithm is a segmentation mask, representing the pixelwise classification (point source or non-point source) as a float in the range]0,1]. As the algorithm assigns all pixels a value >0, there are different ways to compare this output to the input image containing normalized fluxes. We choose two different approaches (denounced A and B) explored below, which will be appropriate depending on the purpose of the problem. In both cases we choose a cut-off value on the probability, only considering predictions with this probability or higher as point sources when comparing to the target pixels. The cut-off value used here has to be carefully considered. We tested several different values taking into account which gives better/worse accuracy as well as how much data we discard in the process for the accuracy measure. We ended up choosing a cut-off value of 0.02 because a lower value would in many cases include very improbable pixels that added weird patterns in the mask. At the same time a higher value would just discard pixels unnecessarily.

Approach A evaluation
All pixels from the true mask with a flux >0 are used in the target mask (including very low fluxes that are extremely difficult to predict). Overall the U-net succeeds in identifying the point source pixels with a mean accuracy of 0.80+/-0.03. The result furthermore reveals an average AUC-score of 0.92+/-0.01 (see histograms of AUC-scores on next page).

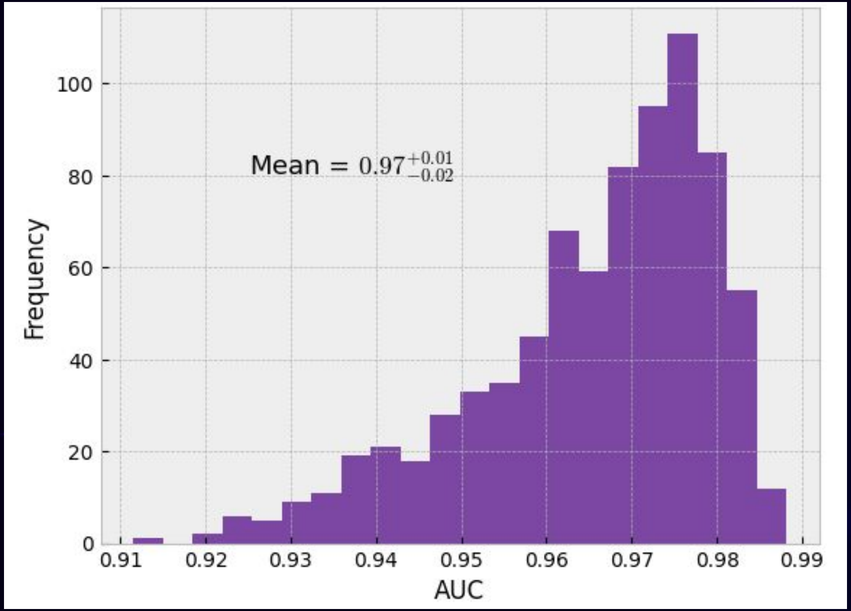
Approach B evaluation
We compare the N pixels from the prediction with a probability higher than the cut value to the N pixels with the largest flux in the target image. This results in a mean accuracy of 0.94+/-0.01. The result furthermore reveals an average AUC-score of 0.97 + 0.01 -0.02.

U-net: AUC histogram

A



B



U-net: Result evaluation

Comparing the two methods
Approach A including all points gives more of a full picture of the algorithm performance. A downside of this approach is that it includes images with very faint point sources, which are very unlikely to be detected. This affects the accuracy considerably compared to approach B.

Approach B includes only the N brightest point sources. This approach evaluates the performance of the algorithm on only the ‘most certain/bright points’ and results in much more accurate predictions compared to approach A. A downside to this approach is that the more dim point sources of the true data are neglected. Choice of the two approaches depend on the goal you are trying to achieve - Are you trying to get a more general idea of how the total galaxy looks, or are you interested in identifying more specifically the exact locations of the brightest point sources out there? The cut-off value can furthermore be chosen to give a better accuracy in one of the approaches - but not both at the same time.

Next steps / Improvements
Further hyperparameter-optimization could be done to the network. We choose the image size and size of the data set ourselves, so increasing these could be an option too. Further work on the penalty term in the loss function as well as a systematic investigation and optimization of the cut-off value would also be interesting. It would furthermore be interesting to test out a completely different type of algorithm for semantic segmentation and compare to the U-net, to see if we actually have chosen the most optimal algorithm for the problem, or if out results could be optimized even further.

Appendix 3
Denoising images using autoencoding (DAE)

DAE

Goal
The goal was to make an autoencoder for denoising images where adative isotropic Gaussian noise was added. To make the simulated images differ more than in the previous parts, the simulation was modified by introducing more randomness. An example of this is the random angle of which the original image is rotated. This was inspired by real space telescopes as these rotate themselves.

Normalization
All the simulated images were normalized by divided the pixel values in <u>each</u> image with the maximum pixel value in <u>that</u> image. Alternative types of normalization were also tested e.g. dividing all pixel values in <u>all</u> images with the maximum pixel value <u>across all</u> images. Since the maximum value varied by approximately a factor of 100 in a dataset with 10,240 images the latter method for normalization did not perform well.

Size of dataset
Training: 8192 simulated images with size 1x64x64
Validation: 1024 simulated images with size 1x64x64
Testing: 1024 simulated images with size 1x64x64

DAE

Why use an autoencoder for this task?
We use an autoencoder for this task since this type of network has the ability to learn compact representations of the images and capture meaningful features in the original images. By limiting the capacity of the autoencoder (through a bottleneck layer with lower dimensions), it forces the model to capture the most prominent features of the input. This regularization effect can help denoise the images by discarding the noisy components and emphasizing the essential details. In addition, since we use non-linear activation functions (Leaky ReLu) in the hidden layers the autoencoder can capture nonlinear relationships between the noisy and clean images. Lastly, an autoencoder works well on unsupervised data as it learns to encode and decode the images without the need for explicit annotations of the noise (labels).

DAE: Architecture and HP optimization

```python
class Encoder(nn.Module):
    def __init__(self, encoded_space_dim, fc2_input_dim):
        super().__init__()

        self.encoder_cnn = nn.Sequential(
            nn.Conv2d(1, 8, 3, stride=2, padding=1),
            nn.BatchNorm2d(8),
            nn.LeakyReLU(inplace=True),
            nn.Conv2d(8, 16, 3, stride=2, padding=1),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(inplace=True),
            nn.Conv2d(16, 16, 3, stride=2, padding=1),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(inplace=True),
            nn.Conv2d(16, 32, 3, stride=2, padding=1),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(inplace=True),
            nn.Conv2d(32, 64, 3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(inplace=True),
            nn.Conv2d(64, 64, 3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(inplace=True),
            )

        self.flatten = nn.Flatten(start_dim=1)

        self.encoder_lin = nn.Sequential(
            nn.Linear(4 * 4 * 64, fc2_input_dim),
            nn.LeakyReLU(inplace=True),
            nn.Linear(fc2_input_dim, encoded_space_dim),
            nn.LeakyReLU(inplace=True),
            )

    def forward(self, x):
        x = self.encoder_cnn(x)
        x = self.flatten(x)
        x = self.encoder_lin(x)
        return x
```

The denoising autoencoder (DAE) consists of an encoder and a decoder component. The encoder applies a series of convolutional layers with batch normalization and Leaky ReLU activation functions to progressively reduce the spatial dimensions of the input image from 1x64x64 to 64x4x4. During 'architecture optimization' we concluded that

- Adding more conv-layers generally improved the result
- Leaky ReLu improved the result compared to ReLu
- Adding a Sigmoid activation function before returning the output made the result worse

Next, the output is flattened and passed through fully connected layers to further compress the information into a lower-dimensional encoded space. The size of the latent space proved to be one of the most important hyperparameters. Performing hyperparameters optimization we concluded

- Size of latent space: 1x6 (examined range: 1x2 - 1x10)
- Learning rate: 0.001 (examined range: 0.0005 - 0.0100)
- Loss function: MSE (alternative loss function: MAE)
- Batch size: 64 (examined alternatives: 32 and 128)

DAE: Different types of noise

```python
def add_gaussian_noise(inputs, noise_factor = 0.1):
     noisy = inputs+torch.randn_like(inputs) * noise_factor
     noisy = torch.clamp(noisy, 0., 1.)
     return noisy

def add_poisson_noise(inputs, noise_factor = 1.0):
    noisy = torch.poisson(inputs * noise_factor) / noise_factor
    noisy = torch.clamp(noisy, 0., 1.)
    return noisy

def add_masked_noise(inputs, mask_fraction = 0.1):
    mask = torch.rand_like(inputs) <= mask_fraction
    noisy = inputs.clone()
    noisy[mask] = 0.0
    return noisy

def add_saltpepper_noise(inputs, noise_fraction = 0.1):
    mask = torch.rand_like(inputs) <= noise_fraction
    noisy = inputs.clone()
    min_value = torch.min(inputs)
    max_value = torch.max(inputs)
    salt_pepper = torch.randint(0, 2, inputs.shape, device=inputs.device)
    noisy[mask & (salt_pepper == 0)] = min_value
    noisy[mask & (salt_pepper == 1)] = max_value
    return noisy
```

We trained and ran the DAE on different types of noise:

- Gaussian: Adds random Gaussian distributed values isotropically to the input data. The noise values are independent and have equal variance in all dimensions.
- Poisson: Adds noise based on the Poisson distribution to the input data, simulating the randomness of these events.
- Masked: Randomly selects a fraction of the input and sets those elements to zero, effectively masking them out. The remaining elements remain unchanged.
- Salt-and-Pepper: Randomly selects a fraction of the input and replaces those elements with either the minimum or maximum value. This creates "salt" (maximum value) and "pepper" (minimum value) artifacts in the data.

The DAE optimization is based on images with Gaussian noise. In order to make an optimal algorithm for alternative types of noise the optimization should be repeated for each alternative type of noise.

DAE: Different types of noise

The MSE-score of the different types of noise cannot be directly compared since the amount of noise added is not identical in the four different cases (except for the masking and salt-and-pepper noise). The best MSE-score for the four different types of noise are

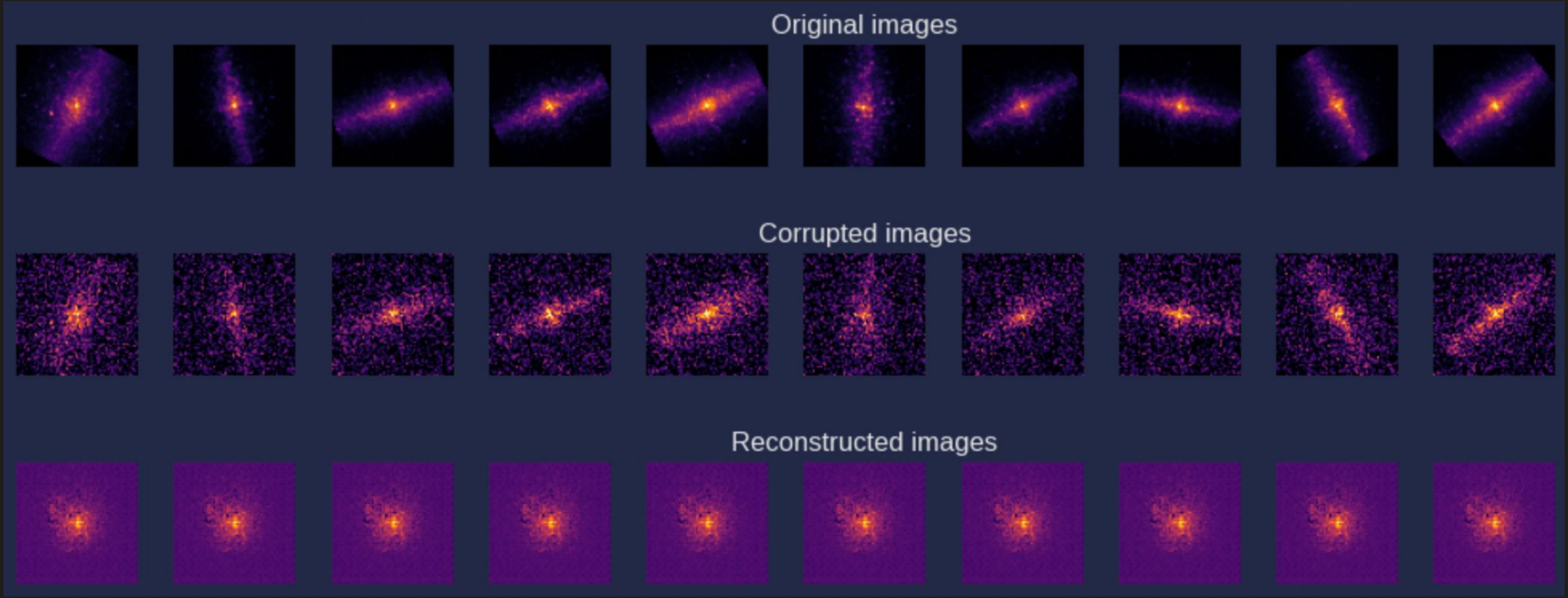
Gaussian w. noise factor = 0.2	MSE = 0.0011
Poisson w. noise factor = 1.0	MSE = 0.0012
Masked w. noise fraction = 0.5	MSE = 0.0010
Salt-and-Pepper w. noise fraction = 0.5	MSE = 0.0019

In the next two slides you can see the performance of the network after 1 and 6 epochs, respectively. After 20 epochs the loss hit a minimum (see slide 20) and after around 30 epochs the network was overtrained.

DAE results after 1 epoch

The network has learnt that the flux intensity is the highest in the center

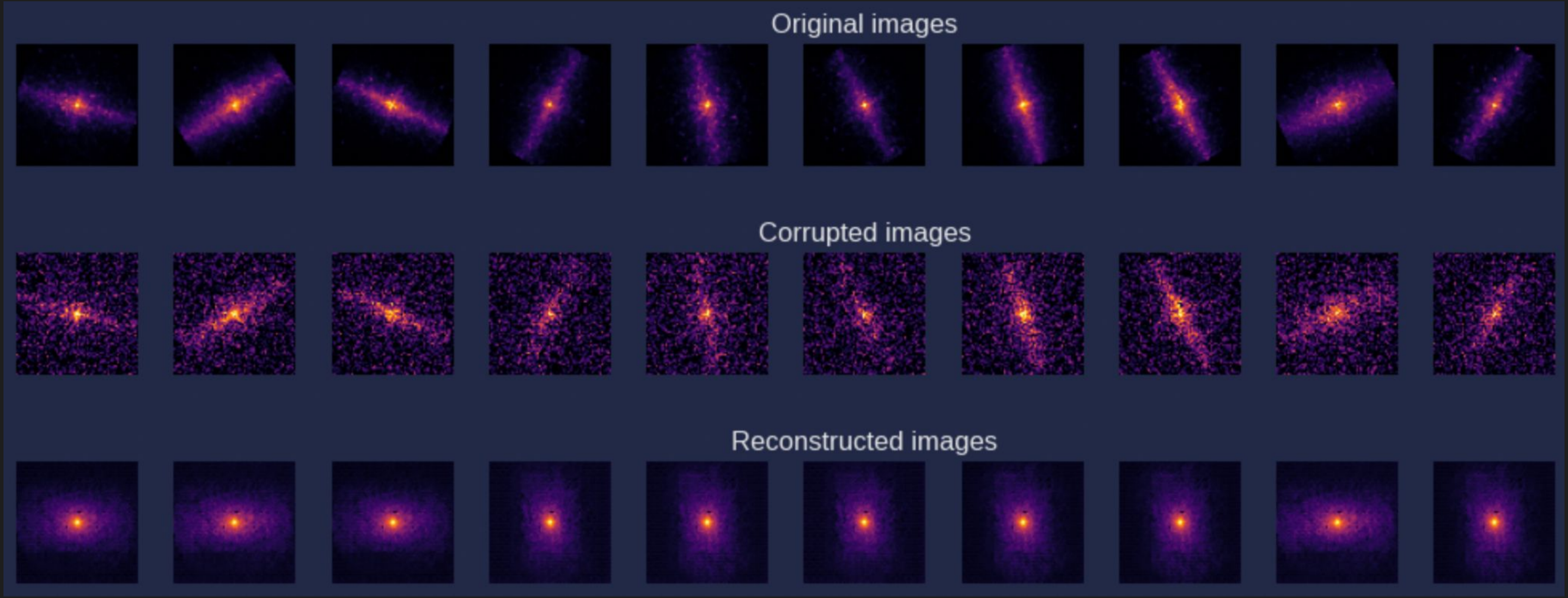
Gaussian noise



DAE results after 6 epochs

The network has learnt that the images are rotated in different directions

Gaussian noise



DAE

Own evaluation

Overall the DAE succeeds in removing the Gaussian noise. It is also good at capturing the angle of which the images are rotated (even the very faint simulations) as well as the flux in the images. However, the output images are far more smooth than the original images. In general, we strive for the output image to be identical to the original image, but in this case the smoothness might be an advantage since this is often a very standard thing to do when dealing with astronomical observations. Both the training and validation data converge towards a MSE value at 0.0011 after approximately 20 epochs. We also ran our algorithm on other types of noise (mostly for fun, since we did not do new optimizations with the alternative types of noise). The algorithm performed with similar loss (0.0012) on the amount of Poisson noise we added. The masking and salt-and-pepper noise can be compared directly since the noise fractions selected were equal. The masking performed significantly better than the salt-and-pepper noise (0.0010 vs. 0.0019) which makes sense when looking at the corrupted images (i.e. the pixels in the background are originally 0 and are therefore only affected by the salt-and-pepper noise and not the masking noise).

Next steps

An easy-to-implement next step could be to mix images with different types of noise in order to make the algorithm more robust. One could also develop the algorithm further, making it classify the different types of noise.