Journal of Fusion Energy (2024) 43:14
https://doi.org/10.1007/s10894-024-00406-x

RESEARCH

Status of the Deep Learning-Based Shattered Pellet Injection Shard Tracking at ASDEX Upgrade

Johannes Illerhaus[1,2] · W. Treutterer[1] · P. Heinrich[1,2] · M. Miah[1,2] · G. Papp[1] · T. Peherstorfer[3] · B. Sieglin[1] · U. v. Toussaint[1,4] · H. Zohm[1,5] · F. Jenko[1,2] · the ASDEX Upgrade Team

Accepted: 18 April 2024

Abstract
Plasma disruptions pose an intolerable risk to large tokamaks, such as ITER. If a disruption can no longer be avoided, ITER's last line of defense will be the Shattered Pellet Injection. An experimental test bench was created at ASDEX Upgrade to inform the design decisions for controlling the shattering of the pellets and develop the techniques for the generation of the fragment distributions necessary for optimal disruption mitigation. In an effort to analyze the videos resulting from the more than 1000 tests and determine the impact of different settings on the resulting shard cloud, an analysis pipeline, based on traditional computer vision (CV), was created. This pipeline enabled the analysis of 173 of the videos, but at the same time showed the limits of traditional CV when applied in applications with a highly heterogeneous dataset such as this. We created a machine learning-based (ML) alternative as a drop-in replacement to the original image processing code using a semantic segmentation model to exploit the innate adaptability and robustness of deep learning models. This model is capable of labeling the entire dataset quickly, accurately and reliably. This contribution details the implementation of the ML model and the current state and future plans of the project.

Keywords Machine learning · Deep learning · Computer vision · Disruption mitigation · Fusion research · Shattered Pellet Injection

Introduction

One of the primary concerns for ITER are disruptions. These are abnormal events in which confinement of the plasma is lost and its energy is deposited onto the machine wall and other plasma facing components within hundreds of milliseconds. Future tokamaks such as ITER are expected to be critically damaged by only a small number of unmitigated disruptions [1]. The first step to avoiding disruptions is a process called disruption avoidance, where the plasma control system steers the plasma into a non-disruptive part of the plasma state space, completely avoiding the disruption. But if this fails, the damage caused by the disruption has to be mitigated. At ITER, the primary disruption mitigation system is the Shattered Pellet Injection (SPI) [2]. This system will shoot pellets against shatter heads, which are structures designed to shatter pellets into fragments with controlled size and velocity distributions. This cloud of fragments is injected into the plasma with the goal of isotropic energy dissipation, cooling the plasma, raising its density, terminating the discharge, and depositing the energy on a large surface area to mitigate structural damage to the device.

To achieve isotropic energy deposition and good material deposition the SPI setup parameters leading to the ideal fragment distribution have to be found. To be able to efficiently test different pellet parameters (e.g. composition, velocity, size), shatter head geometries (e.g. shatter angle, cross section shape) and their impact on fragment distribution, different test beds were created. One of these is a highly flexible

See author list of H. Zohm et al, 2024 Nucl. Fusion https://doi.org/10.1088/1741-4326/ad249d.

✉ Johannes Illerhaus
johannes.illerhaus@ipp.mpg.de

1 Max Planck Institute for Plasma Physics, Garching, Germany

2 Technical University of Munich, Munich, Germany

3 Technical University of Vienna, Vienna, Austria

4 Technical University of Graz, Graz, Austria

5 Ludwig Maximilian Universität, Munich, Germany

test setup [3] at the Max Planck Institute for Plasma Physics in Garching, Germany. An animation visualizing the SPI system is provided in the references [4].

At the test setup in Garching, several variables can be changed and analyzed, including the size and speed and the fraction of neon (Ne) and deuterium (D_2) of the pellet, as well as the geometry of the shatter head. A series of test shots consisting of roughly 1100 tests was completed in which the parameters of the system were varied, leading to a large and diverse dataset. Each of these shattering tests were filmed at frame rates between 20,000 and 30,000 frames per second using Phantom v2012 cameras. Since the goal is an understanding of the resulting distribution of weight and speed of each of the pellets, these videos need to be analyzed. The first iteration of this analysis was done by Peherstorfer [5], by constructing an analysis pipeline based on traditional computer vision (CV) methods.The work of Peherstorfer provided an initial insight into the dependence of fragment distributions on pellet and shattering parameters. This CV pipeline was used for a detailed analysis of 173 of the experiments but showed some of the downsides to using traditional CV on such a variable dataset. Traditional CV is particularly sensitive to changes in the videos, such as illumination. This limitation meant that only a subset of the dataset could be analyzed, since an expert needed to be present and fine-tune the settings of the pipeline regularly. To automate and improve this process and enable quick, precise and robust analysis of the entire dataset, the previous pipeline was in part replaced by a new machine learning based approach.

The purpose of this work is to explain the deep learning pipeline and its current state. The tracking in the original work is done in two steps. First, the background of the image is subtracted. This process ideally results in an image, where the fragments are marked as white pixels and all other pixels are in black. In the second step, the individual fragments are isolated in each frame and an algorithm tries to find the same fragments in two consecutive images. Currently, a semantic segmentation model called U-Net [6] is used to replace the background subtraction of the original pipeline. To function properly, the second step in the original pipeline requires images, where the shards are white and everything else is black. How well the tracking is capable of identifying and tracking shards depends primarily of how well the background subtraction is done. The background subtraction process of the original pipeline is influenced by many different parameters that need to be adjusted to work well on a given video. Changes in i.e. lighting of a video can strongly impact the quality of the background subtraction. The parameters have to be fine-tuned by an experienced human operator, who needs to validate the accuracy of subtraction across the videos it is applied to, each of which has tens to hundreds of frames, with up to several thousands of fragments visible at the same time on each frame. Traditional CV can be highly effective when used in tightly controlled environments, but in this application the fact that the test setup was frequently updated, causes video attributes such as lighting and exposure to change regularly. This in turn frequently caused settings that worked well on the videos produced on one day to become incompatible with videos from the next day. Additionally, changes within a given video, such as lighting changes introduced by plumes of dust can quickly change the lighting of a video. This can make the settings that work in the rest of the video ineffective on the frames with the dust plumes. Analogously, videos with few, large shards are generally brighter than those with more, smaller shards. This can lead to settings from one experiment not translating to the next. These are just some examples of the situations necessitating re-tuning the settings of the pipeline. This sensitivity to changes in the videos is a natural limitation of traditional CV. The need for manual fine-tuning and validation turned out to be the limiting factor on the size of the dataset that could be labeled.

We created a machine learning (ML) pipeline leveraging the 173 previously labeled videos as a training dataset to train U-Net [6] models. These models have standardized and automated the labeling process. This has also reduced the time necessary to label all 1114 videos from what would have been multiple months down to only a few hours on a single GPU. The models are highly accurate, reliable and generalize well to videos created much later than the previously labeled 173. Many of the later videos were taken after substantial upgrades to the test setup, such as changes to the illumination and background, which noticeably altered the videos. Figure 1 shows this change. The image on the left was one of the earlier pellet tests and the image on the right was one of the later ones. The ML models currently act as an automated drop-in replacement to the image processing code of the original pipeline, while tracking is still being done using the original methodology [5].

This paper explains the steps taken in dataset generation and preprocessing in ch. 2, explaining how the labels from the 173 originally labeled videos are transformed into the dataset used for training the ML model. The ML model and its training and inference strategies are discussed in the "Semantic segmentation" section, containing information on the model's implementation such as the architecture used, training and inference pipelines and discussing current results.

The scope of this paper is the explanation of the technical implementation of the semantic segmentation deep learning model in its current state. Detailed description of the physics interpretation of the results and its impact on disruption mitigation system design will be discussed elsewhere.

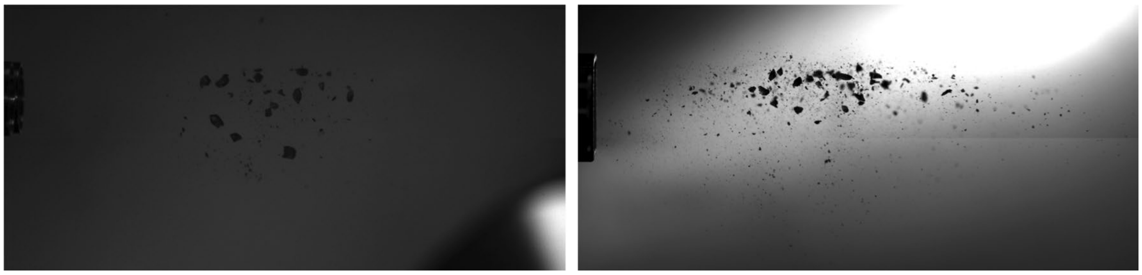

Fig. 1 Example of a frame from an early (pellet #218, left) and late (pellet #1351, right) pellet test

Dataset

Dataset Structure

Our machine learning model is a semantic segmentation model (see "Semantic segmentation" section). The model labels all pixels that are within the confines of any of the shards in the input image. The dataset structure is the same for the training dataset produced by the labeling algorithm (see "Labeling algorithm" section) and for the inference output of the models (see "Inference" section). The datasets are comprised of an input image and a label. The input image is an image taken from the SPI videos and can be either a complete frame or a cropped partial frame, such as a 256×256 cutout. The label for the image is a Boolean mask of the same dimensions as the input image, with each pixel being labeled as `true` if it is contained in a fragment and `false` for all pixels not within the confines of a fragment. The labels are produced by the labeling algorithm in the training dataset and by the trained ML models in their output datasets. The masks are then converted, by setting all `true` values to `255` and all `false` values are kept as `0`. These images are then saved as `PNG`s to more easily be able to work with them. An example of this is shown in Fig. 2, where the input image is shown on the left an the output mask is shown on the right, the white pixels being the `true` values and black pixels the `false` values.

Preprocessing and Labeling

Preprocessing

The original dataset provided by T. Peherstorfer [5] needs to be converted into a dataset of the structure described in "Dataset structure" section. The original labels provide a geometric center, an estimated diameter and an ID for each detected fragment and each frame it was detected on. The position and diameter are given in millimeters (mm) in the original dataset. The x and y values are relative to the left and lower borders of the frame respectively. The fragment diameter was estimated by using the number of pixels detected as belonging to a given fragment in the original pipeline. The first step is to convert the x and y positions and the diameter into pixel values to be able to directly project them onto the images. To get the scaling factor between mm and pixels, we scanned the labels to determine the maximum x and y values in the labeled original dataset and divided them by the width and height of the videos respectively. Both x and y gave the same scaling factor, which we took as confirmation that our methodology was sound. We also confirmed our scaling factor with the author of the original analysis. In the original analysis, the scaling factor was calculated by measuring the diameter of a pellet guide tube in the focus plane of the camera in the experimental setup and comparing it to the number of pixels it had in the videos. The center and diameters of the fragments are then used as input to the algorithm, which creates the training dataset, the labeling

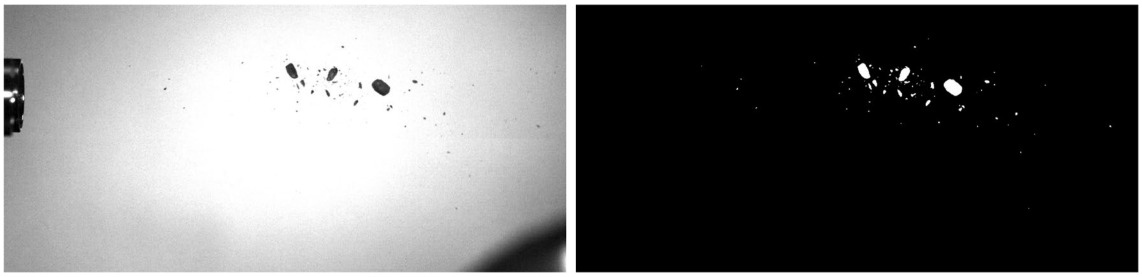

Fig. 2 Image and mask for a frame in the training dataset

algorithm. The dataset needed for training has to have the original video frames and a mask for each of the frames. These masks are Boolean matrices of the same dimensions as the original video frame, where `true` corresponds to a pixel in a shard and `false` to any other pixel.

Labeling Algorithm

For each of the fragments and each frame in the original dataset the algorithm starts by making a square, centered on the center position from the original labels with sides, which are a fraction of the reported diameter of the fragment. This area denotes the algorithm's field of view (FOV). The algorithm then returns a Boolean mask of all pixels with a pixel value below or equal to the average of all pixels in its FOV. Then each of the boundaries is examined to check if they are on or outside the fragment's edge. If not, the FOV is extended on this boundary. This edge detection is also done very rudimentarily at the moment by just checking if the number of pixels the algorithm marked in the Boolean mask on a given edge is below a threshold. This means the "edge detection" finds edges as well as open spaces outside of the shard. For each side of the FOV square where no "edge" is found, the side of the initial FOV square is moved outwards by a fraction of the maximum reported distance of the fragment. If any of the sides were moved and all of them are below a maximum distance threshold from the fragment center, the algorithm recurs with the new FOV rectangle. This functionality is illustrated in Fig. 3.

This labeling algorithm is effective, but can produce some false masks, particularly in dark parts of the images (see Fig. 4). To counteract this, the dataset was manually filtered by looking at overlaid images of the input image and masks like in Fig. 4 and removing all images with faulty masks. This reduced the effective dataset down to about 1/3 of its original size. Notably, a lot of pellets and longer series of pellets were disregarded entirely, since this algorithm is sensitive to changes in the videos. And during different times in the creation of the videos, the lighting, etc. was kept similar. So if the labeling algorithm has issues with a certain lighting setup, it is likely that a majority if not all frames on this lighting setup have to be disregarded. This leads to a notable reduction in diversity in the dataset. A large part of the raw dataset was

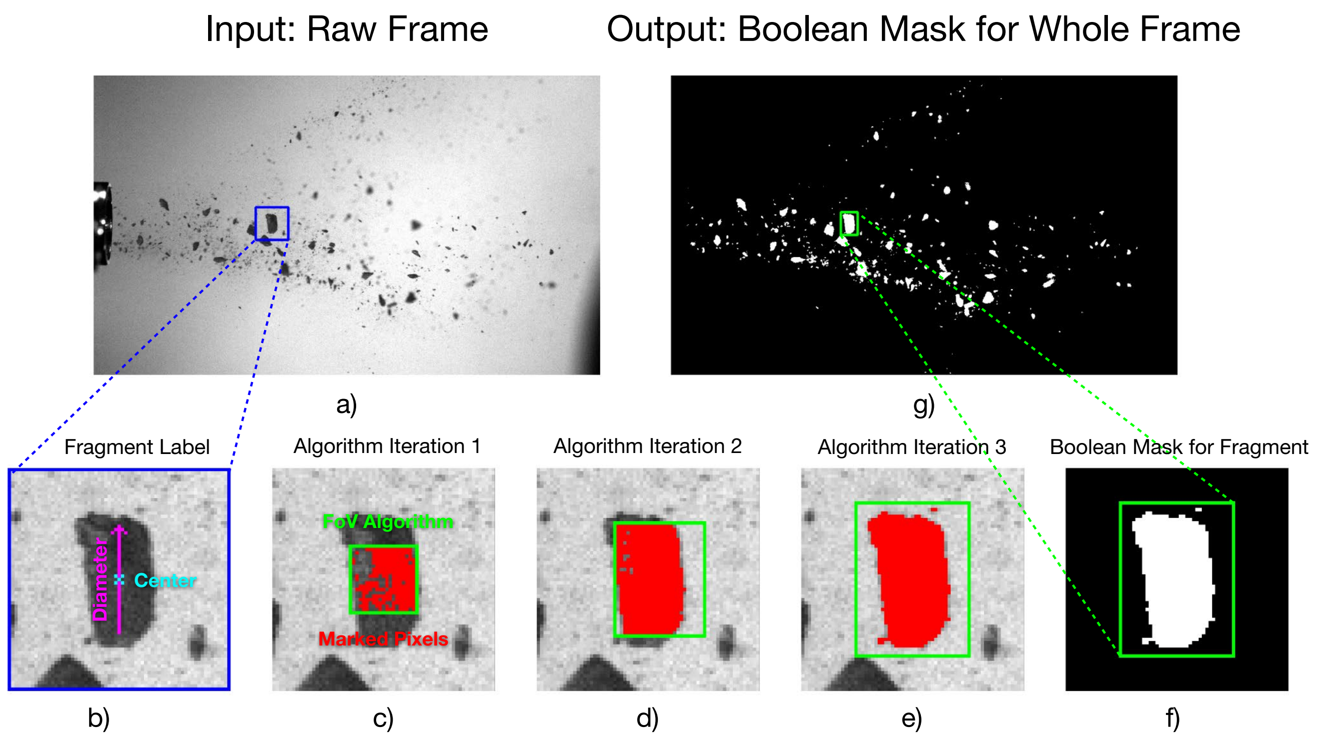

Fig. 3 Visualization of the functionality of the labeling algorithm. **a, b** For each fragment detected in the original analysis, the labeling algorithm is given a geometric center and an approximate diameter (D). **c** It starts with a Field of View (FOV) extending $0.3 \cdot D$ in all directions (green square bottom 2) from the fragment center. **c–e** In each iteration, it marks all pixels that are below the average pixel value in its FOV. It then extends all boundaries with i.e. three or more marked pixels by $0.2 \cdot D$. It then recurs until all boundaries have at most 2 marked pixels or any of the boundaries are more than D away from the center point. **d** Once at least one of these exit conditions is reached, the algorithm returns a Boolean mask, where pixels that are part of the fragment are set to true (white) and all other pixels are set to false (black). The FOV area of the algorithm is then added to the Boolean mask of the whole image with an OR operation. The combination of frame image and mask can then be used as the input and label respectively when training the ML model on this frame

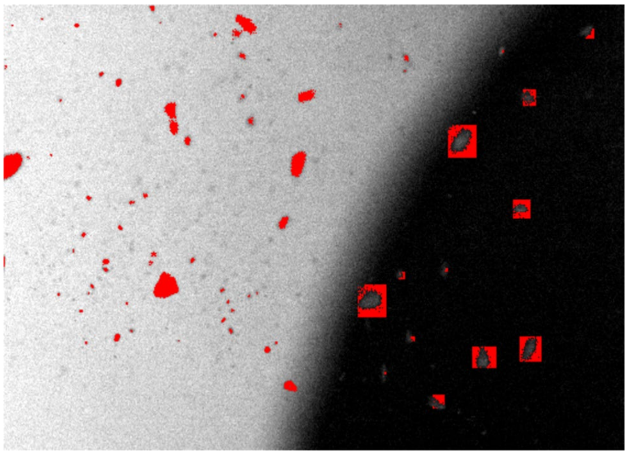

Fig. 4 Pellet 507, frame 17: Normal and erroneous behavior of the labeling algorithm. The fragments are correctly labeled in the bright, left part of the image, and their masks are inverted in the dark portion of the image

also created after the original analysis [5] was done. The highest pellet number of the labeled pellets is 740, but after this more experiments were conducted up to pellet 1390. This is important, because in this time new shattering geometries were introduced and the experimental setup was optimized. This means that ultimately the training dataset does not reflect the entirety of the dataset on which inference is done.

A majority of masks that had to be excluded in sorting was due to the simple labeling algorithm. In many of the images, there is a structure in the experiment vacuum chamber on the lower right as seen by the camera, which leads to a circular dark area in the images (see Fig. 4). If fragments fly into this area, they are still illuminated by the lights of the experimental setup, but are now in a dark background. Since the labeling algorithm looks for and marks any parts of the image around the location of the fragments, where the pixel value is less than or equal to the average pixel value of the entire partial image, it now marks the parts of the partial image around the fragment instead of the fragment itself. This behavior along with normal behavior of the labeling algorithm is shown in Fig. 4.

But even with these limitations, the ML models trained on the resulting datasets are capable of labeling fragments extremely accurately and generalizing well to later parts of the dataset, where the video settings no longer reflect those of the training dataset. Mislabeled shards are exceedingly rare, so while improving the diversity of the dataset would likely lead to some improvement in the models and therefore in the output labels, this improvement would most likely be marginal. Improvements to the labeling algorithm are ongoing but we are also focusing on improving the areas in our dataset creation pipeline with a higher impact on model performance, such as the image processing steps discussed in the next section.

Processing for Training

The dataset at this point in the processing pipeline consists of images and masks of the same dimensions as the high speed videos. An example of this can be seen in Fig. 2. This is typically a height of 608 and a width of 1280.

Image Scaling

The cameras provide videos with a 12 bit color range. To make these videos more easily compatible with many of the image processing tools like OpenCV and to retain a manageable size of the dataset, these are downscaled to an 8 bit color depth. This process inevitably destroys information, so several ways of downcasting were tested. First the naive approach of linear casting was used. Here the maximum pixel value in each frame is set to 255, the lowest is set to 0 and all others are scaled between these two values.

In many cases the actual values are not evenly distributed in the whole space of available values, but clustered around the middle of the value range. This leads to value range compression in the "naive" approach, meaning that more information is lost than necessary. To avoid this, another approach was tested, where a lower and an upper threshold based on a percentile of pixel values is calculated. This could for example be the 5th and 95th percentile. In this case, the value of the 5th percentile is seen as the lower bound, all pixel values below this threshold are set to 0. In this example the value of the 95th percentile is calculated and all pixel values above this threshold are set to 255. The pixel values between the lower and upper bounds are scaled linearly. This approach seems to increase the noise in images and the accuracy of models trained on datasets created with this approach has suffered as a result in these cases. An example of the pixel value distribution of an image typical for the dataset and the cutoff values of the 5th and 95th percentiles can be seen in Fig. 5.

Currently the "naive" approach is used for training and inference, and the percentile scaling approach is used for visualization, because the scaled images are easier for human analysis, since they are generally brighter and therefore easier to understand. In the beginning of some of the later videos, large plumes of dust are present. These clouds are often opaque in naively scaled images, but can be made transparent, when using an aggressive scaling policy such as a 50th percentile upper pixel value cutoff scaling without lower bound cutoff. These dust clouds are not present in the training set, which causes the model to identify the cloud as a single contiguous shard. This then leads to uncertainty in future downstream statistical

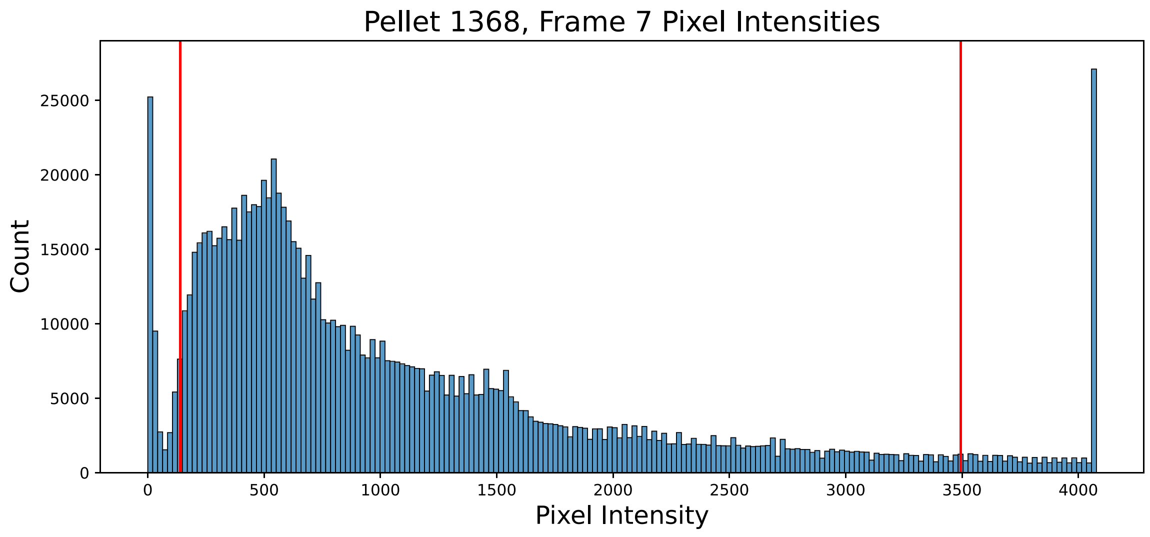

Fig. 5 Pixel value distribution of a typical image. The values of the 5th and 95th percentile of the pixel values are marked in red

analyses of these frames, since it not only introduces a large, incorrectly labeled fragment, but also precludes the model from detecting the actual fragments covered by the cloud. An example of this is shown in Fig. 6.

Currently, first tests are being conducted on the viability of upscaling the 12 bit to a larger data format such as `int16` or `float32`. First results look promising, but the dataset pipeline would require major integration work and extended testing & comparison with the 8 bit results.

Semantic Segmentation

The current machine learning model is a semantic segmentation model. This task trains the model to label all pixels belonging to a specific class, in this case pellet shards.

Model Architecture

The model was chosen to be a U-Net [6] with a EfficientNet B0 [7] backbone. U-Net is a medical imaging segmentation network, that is notable for its implicit data augmentation (see "Training strategy" section). This allows the model to be trained effectively on only a small number of training samples. The model is also arranged in a U-shape, responsible for its name, that passes through different cutouts of the input image at different resolutions, which makes the model well adapted to dealing with features of different sizes in the input dataset. This is particularly beneficial in our case due to the wide spread of sizes of the fragments after shattering (as a result of different pellet sizes, speeds and shatter geometries), and the reason U-Net was chosen for this task.

The backbone is an EfficientNet B0 pretrained on the ImageNet [8] dataset. The goal of the backbone network is to extract and mark pertinent features for the U-Net. An

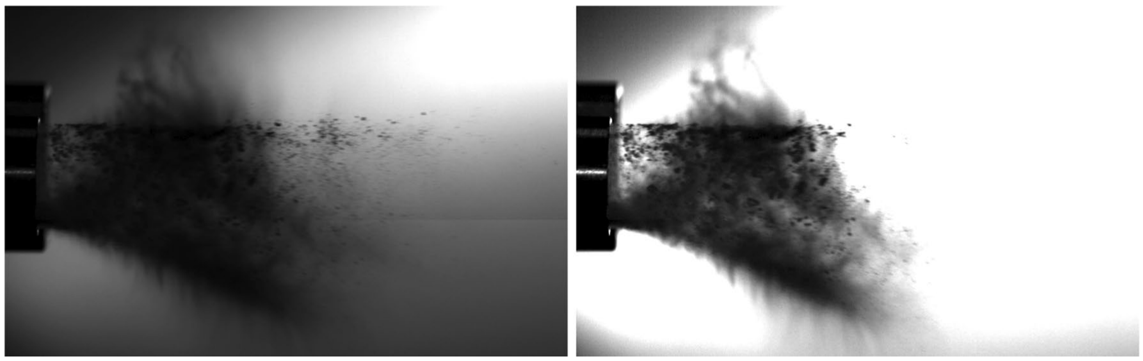

Fig. 6 Pellet 1314, frame 6: naive linear scaling (left), scaling with upper cutoff at 50th percentile (right). Large dust plume is opaque with naive scaling and transparent with 50th percentile cutoff scaling, but fragments in the bright part of the image to the right are lost

example of this would be a network that marks the whiskers for a network tasked with finding cats on images. Fine-tuning a pre-trained model improves the precision of the trained model while reducing the computational cost for training substantially and allows for training on a comparatively small dataset. EfficientNet B0 was used as a backbone, which is the smallest model in the original EfficientNet family, since labeling fragments is a comparatively simple task for computer vision deep learning. Since the accuracy of the models is limited by the quality of the training dataset, there is currently no need to increase the complexity of the model. But as discussed in "Labeling algorithm" section the impact of the dataset limitations on the models' accuracy is minute, with the models performing well above our expectations in their current state. So it is unlikely that a more complex model is needed as a feature extractor, but in later iterations, this will be tested again, to ensure that the backbone complexity is not the limiting factor for model accuracy.

In testing, the models perform well and generalize reliably to unknown situations and parts of the dataset that are different than the training dataset.

Loss Function

The training process of ML models works by trying to find the global minimum of a loss function, which is a function describing the error of a given task. This means that by choosing a loss function, the operator training the model directly chooses the goal the model will be trained towards, making this choice one of, if not the most important choices while training the model. If the loss is not representative of the error of a given task, the model will be trained to solve a different task altogether. Because of the impact of the loss function, this section will explain our reasoning for choosing our loss function, which is a sum of the focal and Jaccard losses:

$$\mathcal{L} = \mathcal{L}_{\text{Focal}} + \mathcal{L}_{\text{Jaccard}}. \tag{1}$$

One of the primary motivations for this choice was the class imbalance inherent in many semantic segmentation tasks, which can also be seen in our dataset. The goal of semantic segmentation is to label the pixels belonging to a certain class. This means that each pixel is assigned a class. In our case two classes exist. Pixels within the confines of any of the fragments are assigned the target class $y = 1$. All other pixels are given the label $y = 0$ denoting them as belonging to the background class. This assignment is first done by the labeling algorithm as discussed in the "Labeling algorithm" section to create the training dataset and later by the trained ML model in inference. The large class imbalance comes from the fact that even in images with a large number of shards, the overwhelming majority of pixels are part of the background class with only a small number of target pixels. Many commonly used loss functions such as $1 - accuracy$ are based on the assumption that the classes are somewhat equally represented. The number of target pixels make up roughly 0.4% of the total pixels in our training dataset. So if for example we were to use *accuracy* as our metric, meaning $1 - accuracy$ as our loss function, a model that exclusively marked all pixels as $y = 0$ would be 99.6% accurate and have a loss of 0.04. This solution is more accurate than any solution where the model actually learns to recognize the fragments. So a loss function needs to be used that prioritizes the *precision* of predicting the target class $y = 1$ to train a model capable of identifying shards. The primary loss function we chose for this purpose was the Jaccard or Intersection over Union (IoU) loss:

$$\mathcal{L}_{\text{Jaccard}} = 1 - J(GT, Pred) \tag{2}$$

where

$$J(GT, Pred) = \frac{N_{Pred} \cap N_{GT}}{N_{Pred} \cup N_{GT}} \tag{3}$$

is the Jaccard index between the set of target pixels of the Ground Truth (GT) and the Prediction ($Pred$). The ground truth target set is the pixels set to $y = 1$ by the labeling algorithm and the prediction set is the set of pixels set to $y = 1$ by the ML model. The Jaccard index divides the number of pixels that are in both the ground truth (N_{GT}) set and the prediction (N_{Pred}) set (the intersection of the two sets) by the number of pixels that are in either or in both sets (the union of the sets). This makes Jaccard loss independent of background pixels, making it a well suited and common loss function in this type of application. But this is not only a benefit, since the Jaccard coefficient will become infinite in cases where the two sets have no union or 0 in cases where there is no intersection. The second case is particularly common at the beginning of training, where the model's weights are randomly initialized. In practice, this often causes the model to either predict all or no pixels, causing an intersection that is 0 or one that will be rounded to 0 due to datatype constraints. This causes a model that is highly volatile with quickly changing weights or one that does not converge at all. To address these stability issues in early epochs of training, a second cost function was added to the Jaccard loss. The focal loss [9] (FL) was chosen:

$$\mathcal{L}_{\text{Focal}} = -(1 - p_t)^{\gamma} log(p_t), \tag{4}$$

where

$$p_t = \begin{cases} p & \text{if } y = 1, \\ 1 - p & \text{otherwise,} \end{cases} \tag{5}$$

and where $p \in [0, 1]$ is the probability the ML model assigns to a given pixel belonging to the target class $y = 1$. And where y in the condition refers to the ground truth label of the pixel, not the label predicted by the model.

FL is a version of the Binary Cross Entropy (BCE), or Log loss, $\mathcal{L}_{\text{BCE}} = log(p_t)$ tailored for use on unbalanced datasets. The additional term in the focal loss is used to suppress the impact of background samples on the overall calculated loss. γ was set to 2 in our training.

Both a version of the model using $\mathcal{L}_{\text{Focal}} + \mathcal{L}_{\text{Jaccard}}$ and $\mathcal{L}_{\text{BCE}} + \mathcal{L}_{\text{Jaccard}}$ were trained and tested and the version using $\mathcal{L}_{\text{Focal}}$ produced slightly more accurate segmentation masks, although the impact on model accuracy, verified through visual inspection of the outputs, was functionally negligible. Additionally other loss functions such as Sørensen dice loss were tested both with and without the addition of a $\mathcal{L}_{\text{BCE}}$ and $\mathcal{L}_{\text{Focal}}$ terms. The models all performed worse than, or indistinguishably from, those trained using the cost function that was ultimately chosen. This result is expected since the Sørensen dice coefficient and Jaccard loss serve very similar purposes.

The Sørensen dice coefficient was also calculated for all models, to have an independent metric to be able to directly compare the training results of different models (see "Meta analysis" section).

Training

General Explanation of Training and Definition of Terms

Deep learning models are generalized function approximators. The function we are trying to approximate is a function that can map inputs, in our case images generated from the SPI videos, to outputs. In our case the outputs are masks (see "Dataset structure" section). To judge how good a model is at mapping the inputs to the output, a loss function is defined, which is a description of the error of the model on a given application (see "Training Stack" section). The goal of training is to adjust the parameters of the model in such a way that the loss becomes minimal over the training dataset.

To achieve this, the model is first randomly instantiated according to its definition, meaning how many parameters and in what configuration should be contained within the model. Computer vision deep learning models of the type we are using, convolutional neural networks, typically contain tens of millions to hundreds of millions of parameters. This high amount of parameters allows the model to "memorize the dataset" instead of learning a generalized representation for the input to output mapping in a process called overfitting, causing the model to become functionally useless when applied to data it has not seen in training. In our case, the model has roughly 10 million parameters. The steps we take to mitigate overfitting are discussed in the "Overfitting Mitigation" section. The parameters are randomly instantiated at the beginning of training and adjusted over several epochs, meaning a single pass where the model is shown every training sample in the dataset once. In our case this could mean, the model being adjusted on all frames of 75% of the 173 labeled videos, the training dataset.

An epoch works by the training dataset being split up into mini-batches (from now on referred to as "batches") of samples. These are groups of images. In our case this could mean a group of 64 256×256 pixel cutouts of images from the dataset. For each of these batches the model undergoes one training step. This step is split into two separate sub steps:

1. *The Forward Pass* in which the e.g. 64 samples of the batch are passed to 64 copies of the model with the current sets of parameters and the models are used to predict the outputs,
2. *Backpropagation* here the predictions of the models are compared to the ground truth, which in our case would be masks labeled by the labeling algorithm (see "Preprocessing and labeling" section). The predictions of the model are compared to the ground truths and an algorithm called an optimizer (in our case we use Adam [10]) is used to determine how much each of the parameters needs to be adjusted for the loss to be minimal on this batch. These gradients are then scaled with a factor called the learning rate (lr) to ensure the model learns iteratively instead of completely overwriting changes of previous training steps.

After all batches are used once, the training is completed for the current epoch. Before the training for the next epoch is started the model has to undergo validation. This means using the model with the current weights after the training of the epoch to make predictions on a dataset it has not seen before. In our example this would be all frames of the remaining 25 % of videos not used in training. Validation is also done by passing the model batches of samples until all samples in the validation set are used up. The average loss and optionally other metrics are calculated for the model on the entire validation dataset. The model's performance on the validation set compared to the training set is particularly important for overfitting mitigation strategies where the hyperparameters, such as the lr are adjusted after validation and before the next epoch (see "Mixed precision Training and Benchmark" and "Training strategy" sections).

After validation and any inter-epoch adjustments are made, the next epoch is started. This is continued until some criterion is met. Often this means that training is stopped once a preset number of epochs is reached. In our case, training is stopped once the validation loss has not improved for a pre-set time, e.g. 10 epochs.

Training Stack

The model was trained on GPU using the TensorFlow [11] framework, with the Keras [12] API. The `segmentation_models` [13] python module was used for the U-Net implementation.

Mixed Precision Training and Benchmark

Mixed precision training is a training strategy, where calculations such as matrix multiplication are done in `FP16` or half precision, but outputs are still saved in full precision (`FP32`). This process, in our case, enables the use of tensor cores in modern NVIDIA GPUs. Depending on the generation of GPU, this can massively speed up training times. This is particularly important when using Turing generation GPUs, since TensorFloat32 (TF32) functionality was introduced in the following Ampere generation. TF32 translates certain operations to `FP16` automatically without needing any explicit implementation. Another benefit of mixed precision is halving the size of training samples and the parameters of the model and optimizer copies loaded onto GPU memory during training, typically doubling the batch size possible on the same hardware, further speeding up training.

We altered the `segmentation_models` code slightly, by forcing `FP32` as the data type for the output layer. Without this change models trained in mixed-precision training, were not numerically stable and as a result did not converge.

Table 1 shows the results of a short benchmark, where the effect of mixed precision on our application was tested. All GPUs followed the same training code and trained models on roughly 10,000 images of the size of 608 × 1024 pixels. The benchmark started with the largest possible batch size when not using mixed precision. This means the RTX 2080ti trained the models in `FP32` mode and all other cards trained in `TF32` mode. The second test switched to mixed precision, while still using the same batch size as the non-mixed precision training run. Finally on the last run, the batch sizes were doubled. The results show a large speed up of all cards when going from non-mixed precision to mixed precision. The 2080ti also shows a large increase in speed when doubling the batch size, but this is most likely due to the fact that it is the card with the lowest amount of VRAM, which means that `FP32` training has to be done with a batch size of 2. Going to 4 increases the GPU usage significantly. The A4000 shows a relatively small benefit when going from a batch size of 4 to 8. The other two cards would actually have been capable of a `TF32` batch size of 8, but they brought our current version of the data loader to its limits at higher batch sizes. This can be seen by the increasing training time of the 3090 and especially the 4090 when going from a batch size of 4 to 8. Both GPUs, but specifically the 4090 showed a low GPU usage or about 60–80% in training, with none of the CPU cores being at a high usage either. This means that we need to transition our dataset away from what are currently `PNG` images to something like TensorFlow Record (`TFRec`) files to use these GPUs effectively when training with large images, like in the benchmark. The `TFRec` format was developed by the TensorFlow [11] project for high throughput applications such as inference with Tensor Processing Units (TPUs). They enable easily parallelizable data streams from multiple files.

The benchmark shows that mixed precision training leads to a substantial improvement to the training times. Additionally, in current versions of the most popular deep learning frameworks, mixed precision can generally be implemented with minimal effort. Our observation, however, is that in ML applications in the physics community specifically, this step is often skipped, so this benchmark is meant to show the benefits to another audience.

Training Strategy

The models were trained starting with a high *lr*, chosen to be about one fifth to one half of the highest *lr* at which the models still reproducibly converged. This could conceivably somewhat counteracts the advantages of using pre-trained feature extractor networks, since the pre-trained parameters are changed in large steps per training step. Models were also trained using a lower starting *lr*, but showed worse accuracy with a longer convergence time. To counteract model overfitting, aggressive *lr* reduction and early stopping policies were adapted. The *lr* was reduced every time the validation loss stagnated for 5 epochs to 0.2 times the previous *lr*. This was done up to a minimum *lr* of 1e-7. And if the validation loss stagnated for 10 epochs, the training was stopped. The

Table 1 Result of benchmark with and without mixed-precision training

Architecture	GPU	BS	t_{fp32} / t_{TF32} $1 \cdot BS$	t_{fp16} $1 \cdot BS$	t_{fp16} $2 \cdot BS$
Turing	RTX 2080 ti	2	522 s (100%)	365 s (70%)	289 s (55%)
Ampere	RTX A4000	4	559 s (100%)	345 s (62%)	327 s (58%)
Ampere	RTX 3090	4	292 s (100%)	168 s (58%)	176 s (60%)
Ada Lovelace	RTX 4090	4	187 s (100%)	111 s (59%)	148 s (79%)

The GPUs were tested in mixed precision both with the same batch size they had when training without mixed precision (BS) and with a doubled batch size model used in inference is the one with the minimum validation loss, not the model after the final epoch of training. This was done to reduce the effect of overfitting. The maximum training epochs were capped at 50, although this number was never reached. Vanilla Adam [10] was used as the optimizer. The dataset was split into a training dataset containing the frames of 75% of the pellets and a validation set, containing 25% of the pellets. The dataset was split by pellet and not by frame to avoid the model seeing different frames of the same pellet during training, leading to a data leak, although this increases the reduction to dataset diversity incurred from removing all frames from individual videos.

Overfitting Mitigation

Due to the limited size and diversity of the training dataset, coupled with the large number of parameters, even in a small computer vision deep learning model like the one used in this application, overfitting is a significant concern. Overfitting is the process in which the model "memorizes" the dataset instead of learning the rules governing the data, causing the model to become incapable of generalizing to unknown data. This is particularly problematic in deep learning models, that can easily have millions of trainable parameters, enabling them to encode even large datasets in those parameters. ML models are trained by finding the set of parameters, which minimize error, represented by the loss function (see "Loss function" section), on the training dataset. But the optimal solution is one in which the model has encoded all points from the dataset in its parameters, because this solution has a loss of 0. This type of model is, however, useless, since it only works on the training data and not on new data. To combat this and to force the model to generalize, several approaches are used here:

- *Shuffling* Possibly the most basic overfitting mitigation strategy is making sure the model does not see the training samples in the same order when it is repeatedly exposed to them in subsequent epochs of training. Although shuffling is omnipresent as an overfitting mitigation strategy, the finer details of how the dataset is shuffled influence how effective it is. To ensure that shuffling has the highest positive impact, our shuffling strategy is to randomly assemble all batches completely from scratch in each epoch, ensuring that the model sees the images in a different order and as part of different batches.
- *Data Augmentation* Data augmentation is a process in which the data used for training is adjusted at train time, to increase the effective size of the dataset. Examples of this are flipping images, changing focus etc. An example of this is shown in Fig. 7. Currently the infrastructure for more complex data augmentation has been implemented in that the models are trained on cropped partial images (e.g. 256 × 256 pixels) instead of the whole image. Some data augmentation has been implemented into the training pipeline in addition to the inherent data augmentation of U-Net, but this will likely be expanded in the future.
- *Early Stopping* If the model's validation loss does not improve for 10 epochs above a threshold, training is stopped.
- *Checkpoint Saving* The model is only saved if its validation loss improves above a certain threshold. This means that the model that is ultimately used for inference on the dataset, the production model, is the one before the final epochs in which the model's stagnation caused the early stopping callback to trigger.

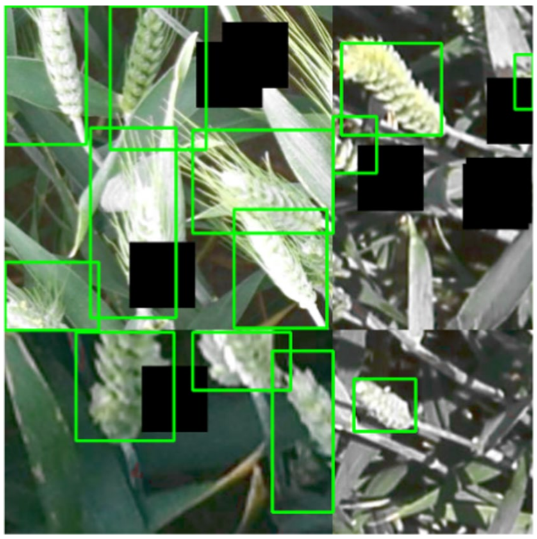
Fig. 7 Example of data augmentation [14]. Input image to the model with ground truth overlaid as the green rectangles. This image is a composition of multiple other images, each with different data augmentation. some sub-images have changed saturation levels, others have black squares added to obscure part of the image. The ground truth needs to be adjusted to reflect the visible features in the augmented input image

Our mitigation strategies can be divided into two groups, based on their objectives. The first group consisting of shuffling and data augmentation tries to make overfitting more complicated and on average make the loss improvements the models gain by overfitting lower than those they gain by understanding the dataset's governing principles for as long as possible. The second group consisting of early stopping and checkpoint saving try to limit the amount of impact overfitting has on the production model after that point.

Our models show strong generalization capabilities and low levels of overfitting behavior. This can be attributed to both the mitigation strategies introduced in this section, but also to the U-Net [6] architecture itself. One primary goal of the development of the U-Net architecture was to enable training medical segmentation models on the small datasets available in these applications, so U-Net is designed with inherent data augmentation. This resistance to overfitting makes U-Net well-suited to our application.

Reproducibility

In many cases it can be necessary to reproduce an earlier model when analyzing results, for example. The model checkpoints of computer vision deep learning models are often hundreds of megabytes to gigabytes in size and in the process of a single project such as this, often hundreds of different models are trained on different workstations. To enable sensible tracking and analysis of the models, it is therefore necessary to be able to reproduce them from code files alone if the need arises. For this reason several steps were taken to ensure reproducibility:

- *git Repository* The code files are saved in a git repository as is standard. But often, especially when training multiple models in a single day, only small changes to the code are made, such as changing the learning rate in between runs. Committing every single small change to the repository is unreasonable, so the decision was made to have a feature-based commit policy for the repository and create an automated backup pipeline to track model-based changes.
- *The Model Directory* At the beginning of training, the model is automatically given a unique model version number and a directory is created for it. This directory is then filled with both the model checkpoint at the epoch with the lowest validation loss and all necessary files and settings needed to recreate the model. How the file backup works is explained in the automatic backup point of this list. The settings backup is explained in the next point about the PARAMS class.
- *The PARAMS Class* At the top of the training notebook, a `PARAMS` class containing important settings for training and the hyper parameters for the model is defined. These settings include all the relevant values such as the version number, starting learning rate, random seed, the architecture, cost function and optimizer of the model, a field for describing the change made for training this model and more. At the end of the training notebook some of these values are corrected. For example models generally stagnate before the set amount of epochs is reached, the `early_stopping()` callback then stops training. At the end of the training notebook `PARAMS["epochs"]` is set to the number of epochs after which training was stopped. Finally, the `PARAMS` class is saved into a JSON file in the model's directory. These settings can then later be used to compare or recreate models later.
- *Seeding* To improve reproducibility of the trained models, some processes that involve random number generation are seeded. Currently the random state of `NumPy`, `TensorFlow` and `random`, as well as the `PYTHONHASHSEED` environment variable are set to the random seed set in the `PARAMS` class at the beginning of the training notebook. This also influences the train/test split, which is important for being able to test models later on only their respective test set. Notably TensorFlow is not set to be deterministic as this would impact performance and only slightly change the resulting models. This means that for example samples can be loaded in different orders when retraining a model, since multiple instances of the data loader class are used and depending on which processor core a given worker was instantiated in a given run, processor temperature, different processors in different machines etc. can impact which of the workers finishes data loading first, changing the order of samples as they are loaded in training.
- *Automatic Backup* When a model finishes training, all of the scripts used in the training pipeline to arrive at this specific model are backed up. To ensure that none of these scripts were changed, this process works by recursively propagating through the entire training pipeline. This means that when a dataset is created, all of the files that are used to create this exact dataset are saved in a backup directory of the dataset directory. When the training script finishes training the model, it automatically copies all of the backup files from the backup directory of the dataset it loads its images from for training into the model's backup directory. This ensures, that every line of code needed to recreate the model is saved.
- *Containerization* Another important aspect of reproducibility is the environment in which the scripts are run. An example would be if the same model is re-run on a later installation of the same modules, this would likely lead to it being trained on a different version of TensorFlow and CUDA [15], possibly influencing the result. For this reason training, as well as all other scripts are run in a Docker [16] container, whose `Dockerfile` as well as a Bash file with run instructions is included in the git repository, to be able to reproduce the environment at the time a given model was trained.

Inference

Inference Stack

Inference was implemented using the TensorRT [17] backend in TensorFlow [11], which uses TensorRT inference for all operations with native TensorRT implementations, but falls back to TensorFlow for all others to reduce inference time compared to inference using only TensorFlow. This reduces the inference time per image from roughly 20 ms to 1 ms.

Output Encoding

Since dozens of models are trained currently and the number might increase to hundreds of models in the future; and even in an optimized state, inference typically takes on the order of 1 to 10 GPUhrs, depending on what GPU is used, it is beneficial to save the inference results. Unfortunately, a single labeled version of the entire dataset is roughly 50 GB in size if saved as images. Being able to compare the output of different models would quickly become unreasonable. To enable comparison and save on storage space, we made the decision to save the model output in the form of run-length encodings (RLEs). Each of these encodings is a string for each image of the type `"beg_0 len_0, beg_1, len_1, ..., beg_N len_N"`, where `beg_n` is the index of the first pixel in cluster `n` of `true` values (representing detected fragment pixels) on the output mask of the model and `len_n` the number of pixels in that cluster. While this reduced the size of a labeled dataset from 50 GB to roughly 500 MB, the typical algorithm to encode the masks into RLEs took roughly 1 s per image, putting the time to convert the entire dataset at roughly 30 h without inference. Since this was unacceptable, we developed a new encoding algorithm using a single 1D convolutional layer to easily parallelize the encoding process and outsource the computation to the GPU. This causes a speedup of around 200x of the encoding process when compared to the traditional encoding algorithm, when used as a separate layer and an even more substantial speedup when the layer is directly attached to the neural network during inference. This layer has a single kernel of the size 2 with the kernel values `[1,2]`. If this kernel is applied to a mask, where pixels within the confines of a shard in the corresponding image are labeled with 255 and all other pixels are labeled as 0, only 4 possible values emerge. For a pixel at the beginning of a cluster of labeled pixels, the output will be $0 \cdot 1 + 255 \cdot 2 = 510$, analogously, a pixel at the end of a cluster will have $255 \cdot 1 + 0 \cdot 2 = 255$, a pixel on the inside of a cluster $255 \cdot 1 + 255 \cdot 2 = 762$ and a pixel outside of any cluster will have $0 \cdot 1 + 0 \cdot 2 = 0$. The output is another mask, which is padded to have the same length as the input mask. The output filtering can then easily be vectorized. The benefit of this method is that it can easily be used on a GPU, which results in a reduction of time from roughly 1 s to around 5 ms per image if the encoding is done as a separate step after inference (Fig. 8). The layer can also be added as the final layer to the model during inference and thereby included into the network graph, which increases the performance gain even further, but this application is limited to networks that infer over whole images instead of partial images, like the 256×256 cutouts discussed as an example in “Training strategy” section.

Inference Time

In total using these methods, the inference time for the whole dataset containing 100,000 images down to about 1 h on an Nvidia RTX 4090 or 4 h on an RTX A4000 Ampere generation and the output of each of these runs is roughly 500 MB. Turning the RLE CSV file back into the Boolean mask PNGs for visual analysis takes roughly 15 s on a 16 core processor.

Meta Analysis

It is important to have a robust meta analysis tool to allow quick iteration and prototyping of new models, and effectively compare their capabilities and possible issues, or to see if a given change to a model has had a positive or negative impact on its performance, or to compare all models using a given architecture against all models using another one. To enable such a tool, it is necessary to save all pertinent training information for each model trained. In the case of the SPI tracking project, the training progress of each model is saved as a `history.json` file into the model directory after training is complete. This file contains the learning rate, training and validation loss, and dice coefficient, as a standard metric for comparing different models, for each epoch of training. Additionally within the root model directory a `results.csv` file is updated when training is completed for a new model. This file contains all the notable settings of that model’s `PARAMS` class and its training results, making it easy to compare models. It, together with the `PARAMS.json` files of the models can be used to filter models and load their respective `history.json` files to compare their training results. To quickly analyze model performance of different models, we created a meta analysis tool to easily filter models and plot their training curves against each other.

Semantic Segmentation Results

Although many of the parameters of the training pipeline are still being optimized, the models already perform accurately and reliably on most of the dataset. How well models perform is judged primarily by human analysis. Although metrics such as the Sørensen dice can be helpful when directly comparing multiple models trained on the same dataset with trusted labels, we have seen, that directly looking at the resulting model output is the quickest and most holistic way of identifying issues with a model and comparing its performance to other models. In this instance, there are unfortunately no perfectly labeled ground truths available. One could conceivably label pixels by hand, but that requires a prohibitively large workforce investment. Therefore, there is no purely objective metric to use to analyze the accuracy of the models. In our experience, working with the data, the ML models’

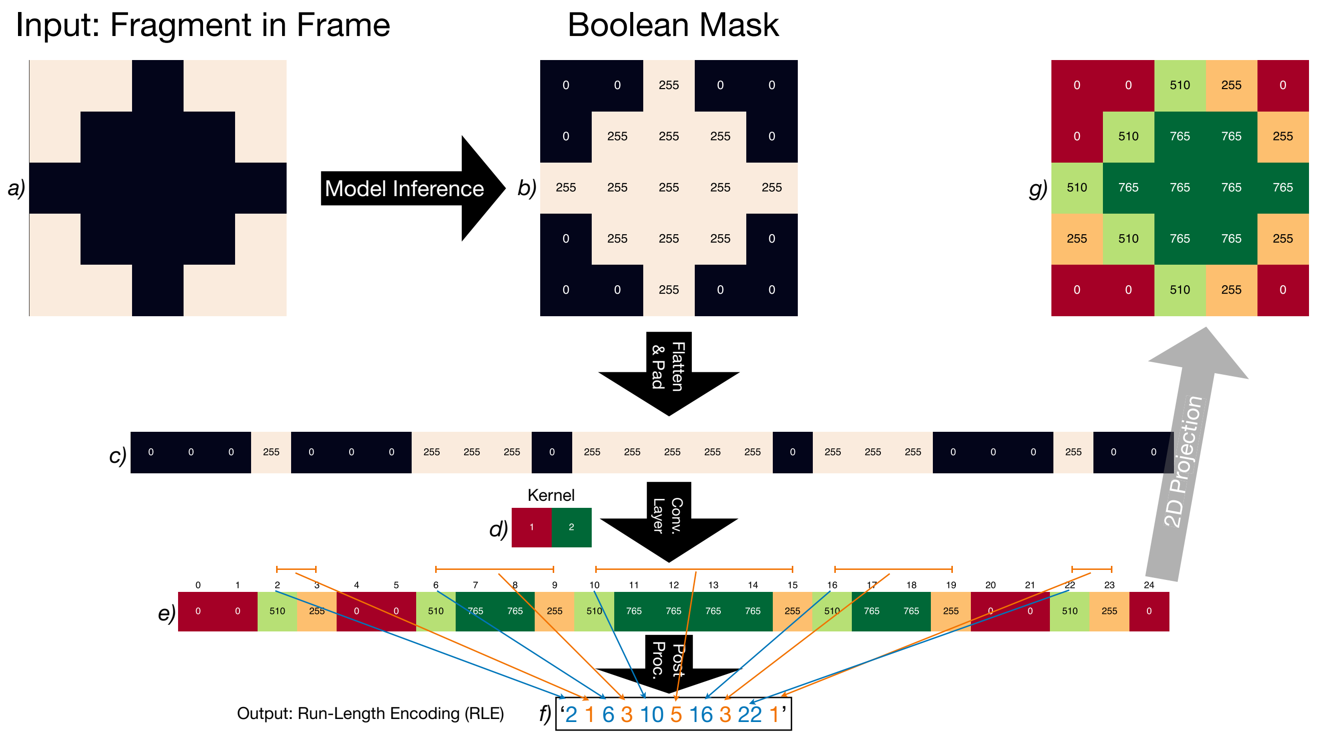

Fig. 8 Visualization of the steps of the inference process. **a** The input for an inference step is one (partial) frame of a video. **a**, **b** The model predicts the Boolean mask of this frame, and true values are set to 255, false values are kept as 0. **b**, **c** This mask is then flattened into a 1D array and padded by a leading 0. **c**–**e** The 1D convolutional encoding layer with kernel *d*) is then applied to mask vector **c**. This results in the output shown in **e**. Only four unique values remain: 0, 765: pixel is in the middle of false or true group respectively, 510: pixel is first pixel in group of true values and 255: pixel is first pixel in group of false values. **g** shows a 2D projection of **e** to visualize the values' position in the image. **e**, **f** vector **e** can then easily be searched for values of 510 and 255 and the resulting indices are used to create the Run-Length-Encoding **f**, which is the output to the inference process of frame **a**

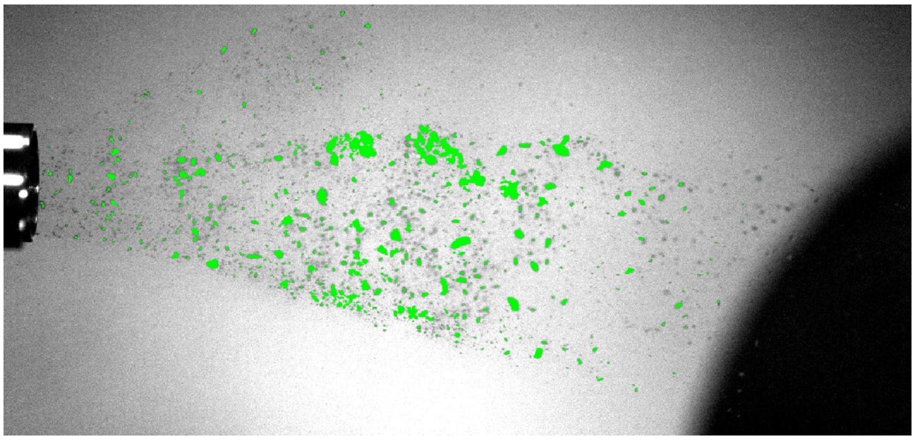

Fig. 9 Pellet 441, frame 15: Frame labeled by the semantic segmentation model. The Boolean mask output by the model is projected onto the input image in green

labels are more accurate than the training dataset labels. This analysis typically takes around 15–30 min after inference is complete and is meant more to get an overview over a model's capabilities than fine-grained analysis of individual samples.

At this time, the models allow us to label the entire dataset, which was previously not feasible. The complete inference process takes a few hours for each iteration, which is fast enough to allow for quick iterative improvements to the models.

The models are generally accurate and adaptable to later parts of the dataset. A representative example of a labeled frame of our semantic segmentation model is shown in Fig. 9. The ultimate goal of the project is to find an accurate distribution of the size and speed of the individual shards. To achieve this, the same shards will have to be identified throughout the entire video.

Summary and Outlook

In this paper, we discussed the current state of our implementation of a deep learning-based shard tracking pipeline. The results from a previous analysis by T. Peherstorfer [5] were used to create a training dataset, which was then used to train U-Net [6] models to locate shards in video frames. These models are capable of automatically labeling the entire dataset of roughly 1100 videos within only a few hours, expanding the number of labeled videos from 173. Visual analysis confirms a high level of accuracy in the labels of the dataset.

In the next step, the focus will be on improving or replacing the existing tracking algorithm to produce accurate size and speed distributions for the fragments of the dataset. These distributions are planned for use in validating the predictions of theoretical models on how different setup parameters in the SPI system impact these distributions and ultimately inform design decisions of the ITER SPI system.

Acknowledgements ITER is the Nuclear Facility INB no. 174. The views and opinions expressed herein do not necessarily reflect those of the ITER Organization. This work was performed in collaboration with the ITER DMS Task Force and received funding by the ITER Organization under contracts IO/CT/43-2084 and IO/CT/43-2116.

Author contributions J.I. wrote the main manuscript text and created the figures. T.P. provided feedback on the accuracy of sections describing his original analysis. P.H., G.P. helped with formulation and fact checking of different sections, particularly those describing the SPI system. W.T. helped with formulation and decisions on which material to include. M.M. provided analyses of output datasets and analyses motivating different statements on impact of different settings to model accuracy. All authors reviewed the manuscript

Funding Open Access funding enabled and organized by Projekt DEAL.

Data availability Base dataset was created with funding by ITER. If access is requested, please contact the corresponding author, who will have to ask ITER for permission.For information about access to original dataset, created by T. Peherstorfer, please also contact the corresponding author. This dataset was also created with funding by ITER and they will have to be consulted.

Declarations

Conflict of interest The authors declare no competing interests.

References

1. M. Lehnen et al., Disruptions in iter and strategies for their control and mitigation. J. Nucl. Mater. **463**, 39–48 (2015). https://doi.org/10.1016/j.jnucmat.2014.10.075
2. M. Lehnen, et al. Physics basis and technology development for the iter disruption mitigation system. (2023). 29th IAEA Fusion Energy Conference London
3. M. Dibon et al., Design of the shattered pellet injection system for ASDEX Upgrade. Rev. Sci. Instrum. **94**(4), 043504 (2023). https://doi.org/10.1063/5.0141799
4. P. Heinrich, SPI animation video (2023). https://datashare.mpcdf.mpg.de/s/DlMzGcWnZwoHMjq
5. T. Peherstorfer, Fragmentation Analysis of Cryogenic Pellets for Disruption Mitigation (2022). arXiv:2209.01024
6. O. Ronneberger, P. Fischer, T. Brox, U-net: Convolutional networks for biomedical image segmentation, in *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015*. ed. by N. Navab, J. Hornegger, W.M. Wells, A.F. Frangi (Springer, Cham, 2015), pp.234–241
7. M. Tan, Q. Le, EfficientNet: Rethinking model scaling for convolutional neural networks, in *Proceedings of the 36th International Conference on Machine Learning. Proceedings of Machine Learning Research*, ed. by Chaudhuri, K., Salakhutdinov, R. vol. 97, pp. 6105–6114 (2019). https://proceedings.mlr.press/v97/tan19a.html
8. J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, Imagenet: A large-scale hierarchical image database, in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, (IEEE, 2009), pp. 248–255
9. T.-Y. Lin, P. Goyal, R. Girshick, K. He, P. Dollár, Focal Loss for Dense Object Detection (2018)
10. D. Kingma, J. Ba, Adam: A method for stochastic optimization, in *International Conference on Learning Representations (ICLR)*, San Diega, CA, USA (2015)
11. M. Abadi, et al., TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org (2015). https://www.tensorflow.org/
12. F. Chollet, et al. Keras. GitHub (2015). https://github.com/fchollet/keras
13. P. Iakubovskii, Segmentation Models. GitHub (2019)
14. J. Illerhaus, Global Wheat Detection Training Repo. fine-grained attributions for dataset (photos and labels), original augmentation algorithm before alterations, etc. can be found in README.md (2020). https://github.com/jillerhaus/Global_Wheat_EffDet_Training
15. NVIDIA, P. Vingelmann, F.H.P. Fitzek, CUDA, release: 10.2.89 (2020). https://developer.nvidia.com/cuda-toolkit
16. D. Merkel, Docker: lightweight linux containers for consistent development and deployment. Linux J. **2014**(239), 2 (2014)
17. NVIDIA: TensorRT. GitHub (2019). https://github.com/nvidia/tensorrt