Supplementary Materials

1 Proof of Drawback of Reward Normalization

Here we provide a short proof on our claim in the main text about the drawback of using reward normalization in sparse reward functions where the reward is only provided at the end of each episode (sequence of actions) leads to non-convergence.

Let the reward function R_t be of the form

$$R_t = \begin{cases} k, & \text{if } t = N \\ 0, & \text{otherwise,} \end{cases}$$

where the reward signal k only appears at the final timestep ($t = N$), and k can take on multiple values depending on the actions taken in the whole episode (i.e. $k \in \{K_1, K_2, \dots\}$). The discounted reward Q_t at each timestep is defined as

$$Q_t = R_t + \gamma R_{t+1},$$

where $\gamma \in [0, 1]$ is called the discount factor that dictates how important future rewards are to the policy. Applying this formula recursively to propagate the reward signal back to the previous action gives us the relationship

$$Q_t = \gamma^{N-t} k,$$

which is a geometric series relative to t. A common trick that is used in many RL implementations is to normalize the discounted rewards by subtracting the mean followed by dividing the standard variation for stability issues in back-propagation, as this restricts the gradient updates contributed by each timestep to be in some specified range. We can easily calculate the mean and variance of this geometric series to be

$$\mu_Q = \frac{k(1-\gamma^N)}{N(1-\gamma)} = k \cdot \alpha,$$

$$\sigma_Q = k\sqrt{\frac{1-\gamma^{2N}}{N(1-\gamma^2)} - \frac{(1-\gamma^N)^2}{N^2(1-\gamma)^2}} = k \cdot \beta,$$

where α and β do not depend on the reward signal k. Applying the normalization scheme gives us

$$Q_{norm,\,t} = \frac{Q_t - \mu_Q}{\sigma_Q} = \frac{\gamma^{N-t}k - k \cdot \alpha}{k \cdot \beta} = \frac{\gamma^{N-t} - \alpha}{\beta}.$$

We can see from the result that the reward signal k *disappears* in the normalized discounted reward terms $Q_{norm,t}$. This implies that no matter the actions taken, all the rewards seen by the algorithm is same, which means that there is no objective to be optimized for. Thus, optimizing does not happen, and the policy never converges. We leave this proof as a reminder for readers interested in re-implementation to pay attention to the reward normalization schemes employed by existing RL packages.

2 Network Architectures

In this section, we detail the network architectures used in our experiments for reproduce-ability. We use a fixed size of 64 for the dimensions of the extracted patch across all experiments such that the patches contain meaningful information of object parts or textures. An open source implementation[1] of the Dense CRF was used during inference for smoothing the raw predictions as discussed in Section 3.3.

2.1 Neural Batch Sampler

The policy of the neural batch sampler is defined by a convolutional neural network with 5 convolutional layers and 2 fully-connected layers. In addition, Batch Normalization is applied to the ReLU outputs following each convolutional layer (i.e., Conv-ReLU-BN), and a softmax is applied to the outputs of the final fully-connected layer to produce a probability distribution of the policy. To extract a patch, we crop the image based on the current center point of the patch (initialized at random). We shift the center point of the patch by a pixel distance of 24 if the sampled action from the policy corresponds to one of the eight directions and randomly select a new image (and a random initial center point) if the sampled action corresponds to *change_image*.

We provide information around the current 64×64 extracted patch to the neural batch sampler such that it can best decide its actions (shifting patch centers or changing images) by using a $128 \times 128 \times 5$ tensor as input, which corresponds to the concatenation of the RGB channels (3 channels), the current reconstruction loss (1 channel), and the binary sampling history (1 channel) of a 128×128 window centered at the current extracted patch. The network structure for the neural batch sampler is given in Table 1.

2.2 Autoencoder

The autoencoder is built in the form of an convolutional encoder-decoder with one added shortcut connection to speed up training. We apply LeakyReLUs with a negative slope of 0.2 and Batch Normalization to every layer except to the output layers of the encoder and decoder. Since the sampled training batches

[1] https://github.com/lucasb-eyer/pydensecrf/tree/master/pydensecrf

Table 1. Network architecture for the neural batch sampler.

			Layer Parameters		
	Input Dimensions	Output Dimensions	Kernel Size	Stride	Padding
Conv 1	$128 \times 128 \times 5$	$64 \times 64 \times 16$	3×3	2	1
Conv 2	$64 \times 64 \times 16$	$32 \times 32 \times 32$	3×3	2	1
Conv 3	$32 \times 32 \times 32$	$16 \times 16 \times 32$	3×3	2	1
Conv 4	$16 \times 16 \times 32$	$8 \times 8 \times 64$	3×3	2	1
Conv 5	$8 \times 8 \times 64$	$4 \times 4 \times 64$	3×3	2	1
FC 6	1024	256	-	-	-
FC 7	256	9	-	-	-

are not sampled uniformly from the data, we do **not** learn the running mean or variance for the Batch Normalization layers and use the empirical mean and variance instead as the running mean or variance can differ dramatically across different training batches. The network structure for the neural batch sampler is given in Table 2.

Table 2. Network architecture for the autoencoder. Note that we add a shortcut connection from the output of Conv5 to the output of Deconv3, doubling the input channels to Deconv4. We set $K = 200$ for MVTec AD and CrackForest and $K = 500$ for NanoTWICE due to the more complex textures.

			Layer Parameters		
	Input Dimensions	Output Dimensions	Kernel Size	Stride	Padding
Conv 1	$64 \times 64 \times 3$	$32 \times 32 \times 64$	4×4	2	1
Conv 2	$32 \times 32 \times 64$	$32 \times 32 \times 64$	3×3	1	1
Conv 3	$32 \times 32 \times 64$	$16 \times 16 \times 128$	4×4	2	1
Conv 4	$16 \times 16 \times 128$	$16 \times 16 \times 128$	3×3	1	1
Conv 5	$16 \times 16 \times 128$	$8 \times 8 \times 256$	4×4	2	1
Conv 6	$8 \times 8 \times 256$	$8 \times 8 \times 128$	3×3	1	1
Conv 7	$8 \times 8 \times 128$	$8 \times 8 \times 64$	3×3	1	1
Conv 8	$8 \times 8 \times 64$	$1 \times 1 \times K$	8×8	1	0
Deconv 1	$1 \times 1 \times K$	$8 \times 8 \times 64$	8×8	1	0
Deconv 2	$8 \times 8 \times 64$	$8 \times 8 \times 128$	3×3	1	1
Deconv 3	$8 \times 8 \times 128$	$8 \times 8 \times 256$	3×3	1	1
Deconv 4	$8 \times 8 \times 512^*$	$16 \times 16 \times 256$	4×4	2	1
Deconv 5	$16 \times 16 \times 256$	$16 \times 16 \times 128$	3×3	1	1
Deconv 6	$16 \times 16 \times 128$	$32 \times 32 \times 128$	4×4	2	1
Deconv 7	$32 \times 32 \times 128$	$32 \times 32 \times 64$	3×3	1	1
Deconv 8	$32 \times 32 \times 64$	$64 \times 64 \times 3$	4×4	2	1

2.3 Predictor

The predictor takes heavy inspiration from existing object segmentation works and is built using dilated convolutions. This allows the receptive field to scale exponentially w.r.t to the number of layers instead of linearly as with normal convolutions. In addition, we apply LeakyReLUs with a negative slope of 0.2 and Batch Normalization to every layer except for the output, where a sigmoid activation is used to provide the labels. The input is the reconstruction loss profile of individual pixels in images, which we define to be the 10 most recent losses in the history across our experiments. The network structure for the predictor on MVTec AD is given in Table 3. For NanoTWICE and CrackForest, we doubled the amount of channels in the hidden layers as we noticed that the predictor experienced significant underfitting.

Table 3. Network architecture for the predictor on MVTec AD. For NanoTWICE and CrackForest, the amount of channels in the hidden layers are doubled. W and H corresponds to the width and height of the input.

			Layer Parameters			
	Input Dimensions	Output Dimensions	Kernel Size	Stride	Dilation	Padding
Conv 1	$W \times H \times 10$	$W \times H \times 32$	3×3	1	1	1
Conv 2	$W \times H \times 32$	$W \times H \times 16$	3×3	1	2	2
Conv 3	$W \times H \times 16$	$W \times H \times 8$	3×3	1	4	4
Conv 4	$W \times H \times 8$	$W \times H \times 4$	3×3	1	8	8
Conv 5	$W \times H \times 4$	$W \times H \times 1$	1×1	1	0	0

3 Hyperparameter Choice

In this section we discuss how the hyperparameters are chosen for our experiments. Apart from the hyperparameters that are calculated solely from dataset statistics, other hyperparameters mentioned below are fixed across experiments.

α reweighs the prediction loss contributed by the anomalous and non-anomalous pixels since anomalous pixels are much fewer in quantity (see in Eq. 2 in main text). It is defined and calculated mathematically as follows using the data in labeled subset:

$$\alpha = \frac{\#\text{ non-anomalous pixels}}{\#\text{ anomalous pixels}}$$

β is calculated from the current number of trained epochs j and L (see in Eq. 3 in main text). For choosing L, it is important that the chosen hyperparameter allows the right amount of transition between the two reward terms. In practice, we first looked at how long it takes for the network to learn to imitate/clone the pre-defined strategy (i.e., if we only trained on R_{clone} and $R_{coverage}$ alone), and set L accordingly such that half the weight of the reward (β) is placed on R_{clone}

as it nears convergence. Empirically, this value can change and fluctuate across tasks, but we set $K = 40000$ and calculate β accordingly, which seems to work well.

W and H is set to 256, which is the recommended input size for the images in the datasets used for evaluation. We use a batch size of $N = 8$ for our experiments.

K determines how often the autoencoder is reset (see Alg. 1 in main text). To choose this hyperparameter, we trained the autoencoder and observed how long it takes for the reconstruction losses to converge, and we set $K = 100$ as the gradient updates become small, which suggests that the network produces very similar loss history profiles after 100 training epochs.

M determines when we start collecting the loss history profiles after resetting and reinitializing the autoencoder (also see Alg. 1 in main text). The choice for this hyperparameter is rather forgiving provided that K is properly chosen, as the "noisy" loss history profiles will only make up a small amount of training data for the loss profile based predictor. We set $M = 15$ in all our experiments.

The length of the loss history profiles T is chosen as 10. Since we want to reduce the amount of overlap between two adjacent loss history profile windows, we generally want this value to be as low as possible while being able to capture a meaningful loss curve to serve as a feature to the predictor.

4 Additional Results

Here we present some additional results of the visualizations of our algorithm on the various datasets in Fig. 1, 2, and 3. Some visualization of more unseen anomaly modes during training in MVTec AD can be found in Fig. 4.

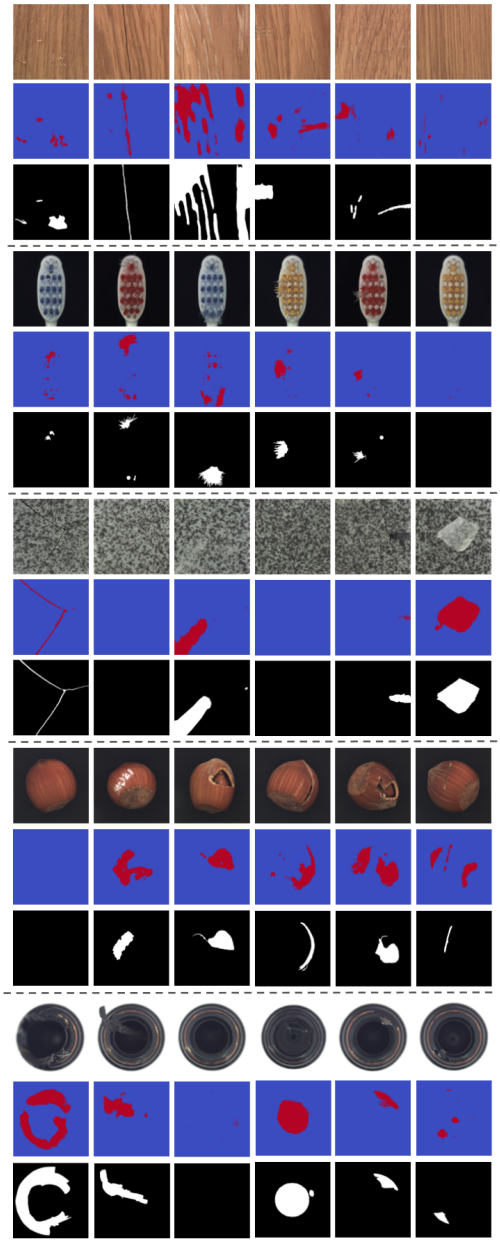

Fig. 1. Visualizations of our predictions on various classes in MVTec AD. The ordering of the rows per class are the original images, the predictions, and the ground truth.

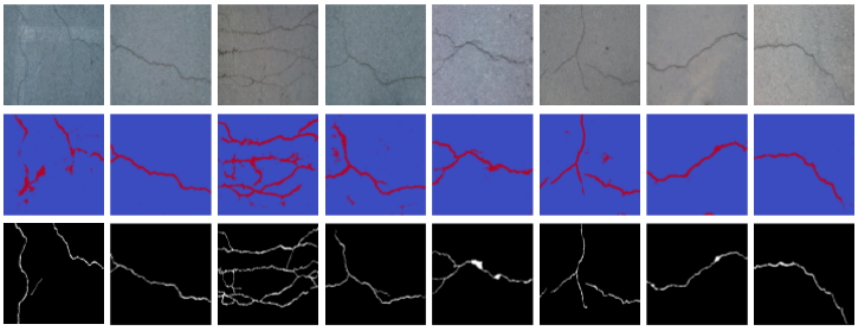

Fig. 2. Visualizations of our predictions on CrackForest. The ordering of the rows are the original images, the predictions, and the ground truth. In general, our algorithm produce thicker/larger predictions, leading to overall lower precision despite good localization of the anomalies.

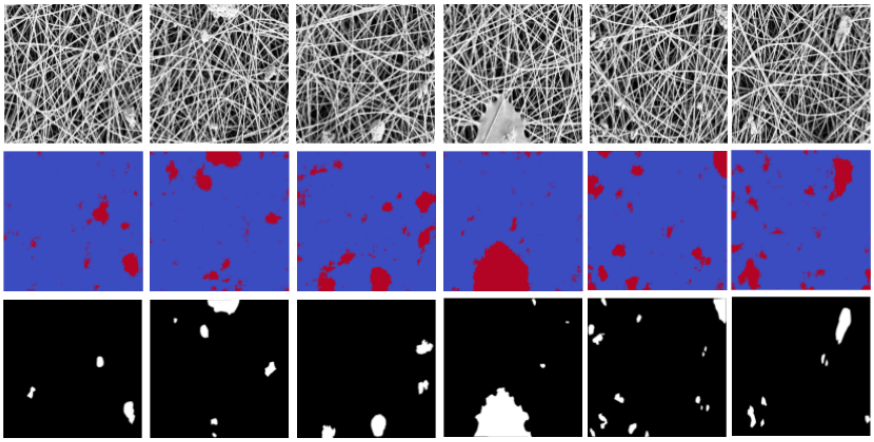

Fig. 3. Visualizations of our predictions on NanoTWICE. The ordering of the rows are the original images, the predictions, and the ground truth. In general, our algorithm predict larger anomalies, leading to overall lower precision despite good localization of the anomalies.

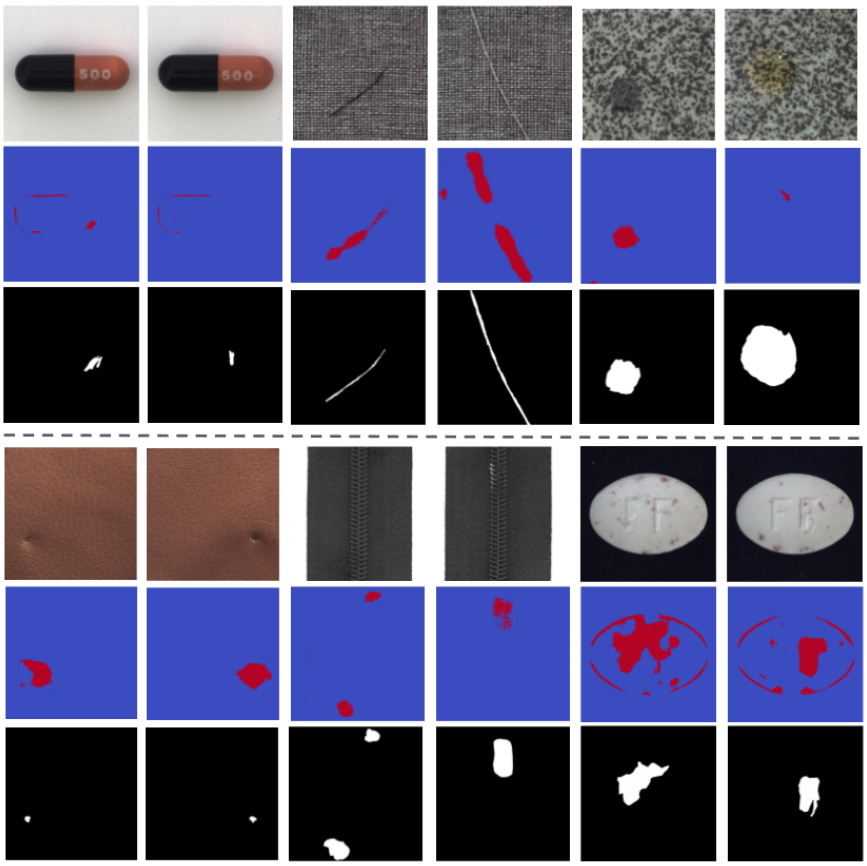

Fig. 4. Visualizations of our predictions on unseen anomaly modes in various classes in MVTec AD. The ordering of the rows are the original images, the predictions, and the ground truth. Our model has some capability to generalize to unseen anomaly modes not observed during training.