The Open University of Israel

Department of Mathematics and Computer Science

מימוש אלגוריתם לזיהוי ואיתור שינויים בתמונות דיגיטליות

Implementation of digital image tampering detection and localization method

Final Project

Submitted as partial fulfillment of the requirements towards an

M.Sc. degree in Computer Science

The Open University of Israel

The Department of Computer Science

By

Sivan Attias

Prepared under the supervision of Dr. Ran Bittmann

March 2022

Table of Contents

1. Introduction

The procedure of image tampering is to replace the content within a region of the original image by some new content. The source and composition of the new content determines the specific type of tampering. When the new content appears entirely in some other place within the original image itself, it is named as copy-move. When the new content is taken entirely from a different source image, it is called cut-paste, or splicing. When the new content is a composition of small patches within the original image or from another image, it is called erase-fill.

The most naïve form of copy-move and cut-paste methods is simply copying of pixels, as the names suggest. However, obtaining a high-quality tampering (i.e., one that is difficult or impossible to observe), usually involves the usage of advanced image editing algorithms such as diffusion, edge blending, color change or other more sophisticated image processing steps.

There are some methods that can detect and sometimes localize the tampered areas. Some of the methods are intended for a specific type of tampering and some are general and can detect any tampered regions.

Image forgery *detection* methods authenticate the genuineness of a digital image. In other words, given an input image, they classify the entire image as either authentic or forged. Image forgery *localization* takes a step further and tells us where the forged parts (or pixels) within the image are.

This paper is a final project description. It first presents a method to detect and localize copy move forgeries based on [1]. Then I present my own implementation [2] and show improvements to the original research [1]. Finally, I discuss the results with comparison to the method in the original research [1].

2. DOA-GAN

DOA-GAN [1] is a recent method for image copy-move forgery detection and localization. DOA-GAN stands for Dual-Order Attentive Generative Adversarial Network. Like BusterNet, this method detects copy-move forged regions in images, including source/target localization. This method consists of two sub networks – Generator and Discriminator, as designed in GAN. The generator is a CNN-based end-to-end network. Given an input image, the generator calculates an affinity matrix based on the extracted feature maps. From the affinity matrix it produces two attention maps, A_1 and A_2, via a Dual-order Attention Module: A_1 is the 1st order *region-aware attention map*, which highlights regions that are likely to involve copy-move manipulations (either source or target regions). A_2 is the 2nd order *co-occurrence attention map*, that provides localization information by highlighting the locations of similar patches. The deep CNN features are then processed with the two attention maps, resulting in a fused feature map that is fed into two separate prediction heads: The first one ('Detection Branch') works on image-level, and predicts whether the entire image is forged or not, by estimating a single confidence score. The second one ('Localization Branch') works on pixel-level and produces a copy-move prediction mask in which the source and target/forged regions are distinguished.

The Discriminator sub-network's responsibility is to check whether the mask produces by the generator looks real (identical to ground truth) or fake. It is only used during training, to facilitate the GAN loss component. During inference, only the Generator is used as an end-to-end solution.

2.1 Architecture overview

A high-level overview of DOA-GAN pipeline overview is presented in Figure 1. Given an input RGB image ($H \times W \times 3$), the generator network first extracts features at 3 different scales ($\times\frac{1}{2}, \times\frac{1}{4}, \times\frac{1}{8}$ of input resolution) using the first 4 block of VGG-19, and then resizes them to the same size in order to concatenate them. The resulting feature map F_{cat} is of size $h \times w \times d$ where $h = \frac{H}{8}$, $w = \frac{W}{8}$ (in order to reduce computation time). In VGG 19 each block (seen as a gray box in the left of Figure 1) is followed by MaxPool down sampling, such that the output of block #1 (① in Figure 1) has size $\frac{H}{2} \times \frac{W}{2} \times d_1$ and is downscaled by factor 4. Similarly, ② is of size $\frac{H}{4} \times \frac{W}{4} \times d_2$ and is downscaled by factor 2. ③ is of size $\frac{H}{8} \times \frac{W}{8} \times d_3$, so that the concatenated feature has $d = d_1 + d_2 + d_3$ channels.

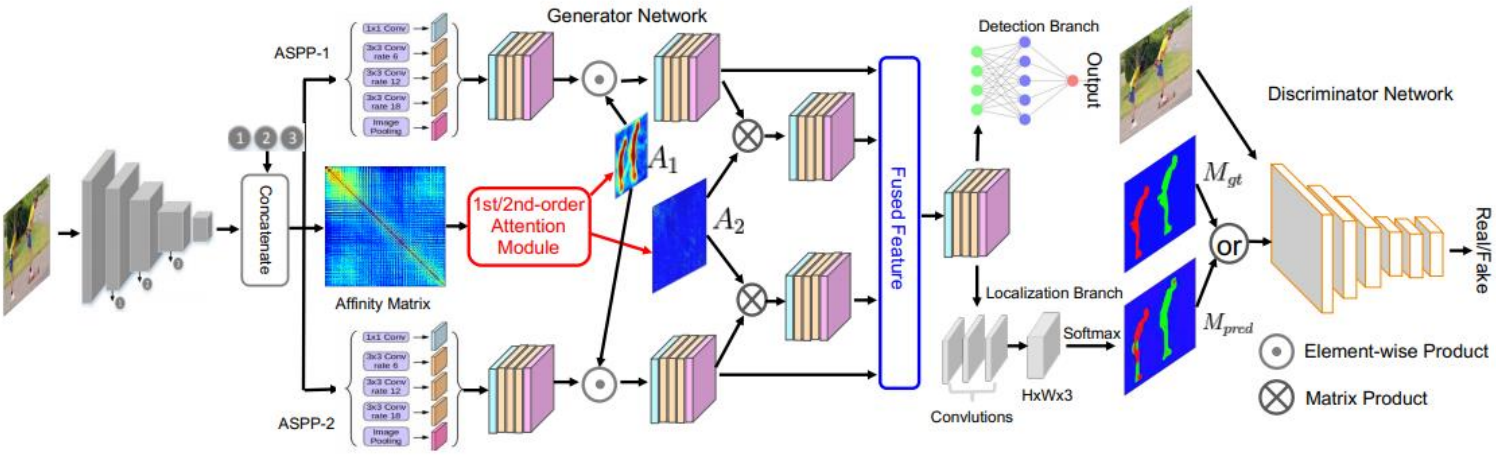

Figure 1 :DOA-GAN architecture overview [1]

F_{cat} is then fed into 3 branches: Affinity matrix (followed by Dual-Order Attention Module) and two Atrous Spatial Pyramid Pooling (ASPP) modules, named ASPP-1, ASPP-2.

The Affinity matrix S is calculated as $S = F'_{cat}F'^{T}_{cat}$ (matrix multiplication) as illustrated in Figure 2, where F'_{cat} is the result of flattening F_{cat} from $h \times w \times d$ to $hw \times d$ so that the rows of F'_{cat} represent the feature vectors at each of the $h \times w$ locations.

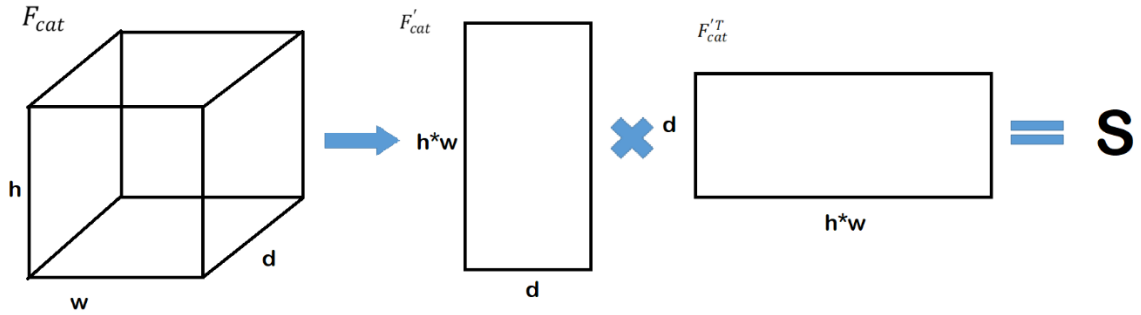

Figure 2:Calculation of Affinity Matrix S

The resulting S is a symmetric matrix of size $hw\ x\ hw$. Since it represents the feature map's self-similarity, the diagonal of S represents the correlation of an image patch with itself, so the values in the diagonal (and in other locations which represent patches that are spatially very close) are obviously very high, possibly masking off-diagonal copy-move regions.

To resolve this issue, the correlation score between the same (or very close) parts of the image is diminished using a gaussian kernel. Specifically, a function G is defined as follows: $G(i,j,i'j') = 1 - e^{-\frac{(i-i')^2+(j-j')^2}{2\sigma^2}}$, for each pair of x-y coordinates (i,j) and (i',j'). When the Euclidean distance d between 2 locations (i,j) and (i',j') is 0 or very small then $G \approx 0$ (full reduction), and when d is large

enough then $G \approx 1$ (no reduction). The goal is to penalize correlations between close or identical patches. G is applied on S and the modified affinity matrix S' is obtained: $S' = S \odot G$, where $\odot$ denotes the element-wise product. Applying G on S is the first operation in the **Dual-Order Attention Module** (Figure 3). In the following next steps, two attention maps are obtained through this module: The 1st-order copy-move aware attention map A_1, and the 2nd-order co-occurrence attention map A_2. A_1 highlights copy-move candidate regions, and A_2 localizes co-occurrences between similar regions. In order to normalize the values in S' a SoftMax function is applied, and the given result is the matrix L of the same size of S', i.e., $hw \times hw$.

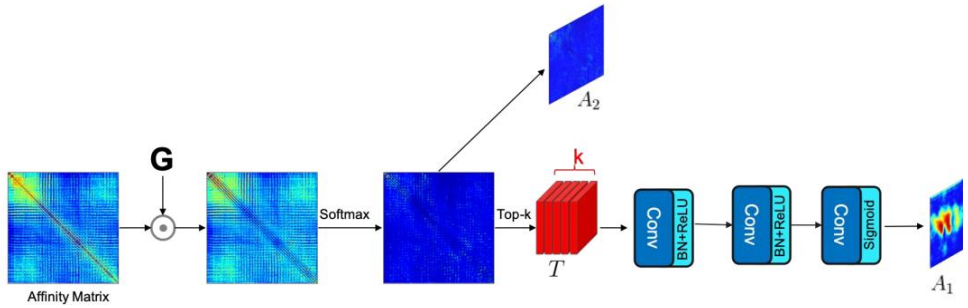

Figure 3 : The dual-order attention module to obtain the copy-move region attention map A1 and the co-occurrence attention map A2 [1]

It is agreed now that L contains the likelihood that a patch in the i-th row matches with a patch in the j-th column in S'.

The next step is to extract the top k values from each row in L, and reshape the result into a tensor T of shape $h \times w \times k$. This means that T contains, for each location, the values with the highest likelihood for a copy-move forgeries. Note that due to the max operation, T loses the localization, or co-occurrence information that exists in L (i.e., "patch at location *i* is similar to patch at location *j*"), therefore both are kept for later use. T is fed into an attention module which consists of 3 convolution blocks. The first two blocks contain convolution layers with 16 output channels and kernel size 3, followed by Batch-Normalization and ReLU. The final block contains two consecutive convolution layers with 16 output channels and kernel size 3, and 1 output channel and kernel size 1, respectively. The result is given by attention map A_1 as shown in Figure 4. The attention map A_2 is obtained by normalizing the matrix L such that the sum of each row equals to 1. Both attention maps will be used later to produce the final feature map, as described below.

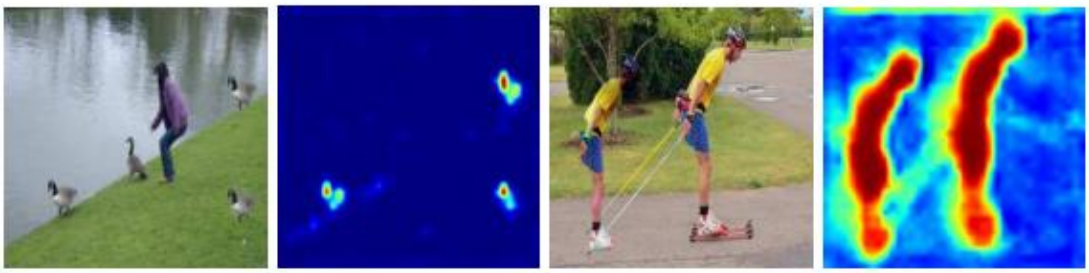

Figure 4 : Visualization of A1 on two copy-move forgery [1]

So far only one part of the generator was discussed – the branch of the dual-order attention module. As mentioned before, F_{cat} is also fed into two Atrous Spatial Pyramid Pooling (ASPP) blocks, ASPP-1, and ASPP-2. ASPP block is utilized in semantic segmentation networks to capture context at several scales for semantic image segmentation. ASPP works as follows: It takes the input feature map and computes 5 operations in parallel: (1) Standard convolution with 1x1 kernel; (2-4) 3 Atrous Convolutions with different rates; and (5) global image pooling.

In Atrous Convolution, the rate is defined as the distance (filled with zeros) between two kernel elements, as illustrated in Figure 5 (right). For example, for a 3x3 kernel with rate=4, three zeros are separating between every 2 elements of the original kernel [3].

The Global Image Pooling operation spatially averages all the input elements. This operation is done channel-wise, and the result is upscaled to a new image with the values of the average.

As mentioned before, two ASPP modules, with different parameters, are applied to extract contextual features F^1_{aspp} and F^2_{aspp} , both of size $h \times w \times d_s$. The second block has atrous rates 6, 12 and 24, as shown in Figure 5 (left)

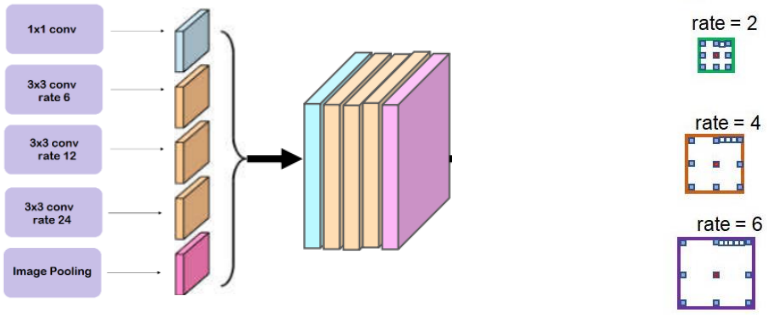

Figure 5: Left: ASPP-1 [1]; Right – Atrous Convolution illustration

The authors found through experiments that two ASPP blocks are useful to learn two different tasks, i.e., source and target detection. F^1_{aspp} and F^2_{aspp} are multiplied elementwise with $A_1$to get the possible copy-move attentive features F^1_{attn} and F^2_{attn} :

$$F^1_{attn} = F^1_{aspp} \odot A_1 \quad ; \quad F^2_{attn} = F^2_{aspp} \odot A_1.$$

A_2 is then used to obtain co-occurrence features F^1_{cooc} and F^2_{cooc} from F^1_{attn} and F^2_{attn}:

$F^1_{cooc} = A_2 \otimes {F'}^1_{attn}$ and $F^2_{cooc} = A_2 \otimes {F'}^2_{attn}$, where $\otimes$ is the matrix product operation. ${F'}_{attn}$ is the flattened version of F_{attn}, $h \times w \times d_s \rightarrow hw \times d_s$, so that F_{cooc} is of size $hw \times d_s$.

The region attentive features and co-occurrence features from both branches are fused via concatenation, to obtain the final feature representation:

$$F_{final} = Merge\ (F^1_{attn}, F^2_{attn}, F^1_{cooc}, F^2_{cooc}\ , A1)$$

The next step is to feed the feature vector F_{final} into the detection and localization branches, where the per-image detection score and per-pixel segmentation map are created, respectively. The detection branch outputs a detection score (forged/non-forged) via 2 fully connected convolutional layers. The localization branch consists of three convolution blocks (each followed by BatchNorm + ReLU), and a final convolution of 3 output channels to generate the segmentation mask of pristine (background), source and target regions.

The discriminator is designed to check whether the predicted mask is identical to the ground-truth or not i.e., to predict whether each N × N patch in the image is real or fake. The discriminator is fully convolutional. It consists of five convolution blocks, each followed by Batch Norm and Leaky ReLU, and a final convolution layer. The output channels of the consecutive convolution layers are 32, 64, 128, 256, 512, and 1, respectively, and the kernel size for all the convolution layers is 4 × 4. The stride of the convolution layers is 2 except the last one, which has a stride of 1. Therefore, as the input image is passed through each convolution block, the spatial dimension is decreased by a factor of two, and finally we get an output feature of size $\frac{H}{2^5} x \frac{W}{2^5} x 1$, where the spatial size of the input is HxW. The input to the discriminator network is the concatenation of the image I ($HxWx3$) and mask M ($HxWx3$).

2.2 Training

DOA-GAN is trained with 80,000 copy-move forged images and their ground-truth masks from USC-ISI CMFD dataset (from BusterNet), and 80,000 pristine images used as negative examples to train the Detection Branch. It is evaluated on the 10,000 testing forged images from USC-ISI CMFD and 10,000 pristine images. All the pristine images were collected from COCO dataset.

The loss function used to train DOA-GAN consists of 3 components:

$$L = L_{adv} + \alpha L_{ce} + \beta L_{det}$$

The Adversarial Loss L_{adv}, is the standard formulation of GAN loss, where the discriminator is trained to distinguish the ground-truth mask from the predicted mask, while the generator tries to fool the discriminator:

$$L_{adv}(G, D) = E_{(I,M)}[\log\big(D(I, M)\big) + \log\big(1 - D\big(I, G(I)\big)\big]]$$

The discriminator D tries to maximize the loss, and the generator G tries to minimize it.

Cross-Entropy Loss L_{ce} is applied on the Localization Branch output mask, and is expressed by:

$$L_{ce} = \frac{1}{HxWx3} \sum_{k=1}^{3} \sum_{i=1}^{H} \sum_{j=0}^{W} M(i,j,k) \log \widehat{M}(i,j,k),$$

where $\widehat{M}$ is the predicted mask of the generator network, and M is the ground-truth mask. This formulation is a simple averaging of 3-class per-pixel cross entropy, where the 3 classes are source, target and pristine.

Detection Loss L_{det} is the binary cross-entropy loss between the image-level detection score from the detection branch and ground truth label:

$$L_{det} = y_{im} \log(\hat{y}_{im}) + (1 - y_{im}) \log(1 - \hat{y}_{im}),$$

where $\hat{y}_{im}$ is the output of the detection branch and y_{im} is 1 if the image contains copy-move forgery and 0 otherwise.

2.3 Evaluation and results

To test the effectiveness of DOA-GAN approach and compare its performance to other copy-move detection/localization methods (such as BusterNet and ManTra-Net, both discussed earlier), evaluation experiments were conducted on three benchmark datasets: USC-ISI CMFD dataset (whose train set was used to train DOA-GAN), CASIA CMFD dataset, and CoMoFoD dataset.

To evaluate detection and localization performance, the authors report image-level (for detection) and pixel-level (for localization) precision, recall, and F1 score metrics. For the pixel-level evaluation, those metrics are computed for 3 classes: Pristine (background), Source, and Target, by averaging the scores of each image. As F1 score is ill-defined for pristine images, the testing images for pixel-level evaluation include only the forged images

To assess the effectiveness of different parts of the DOA-GAN system, several ablation models were evaluated in addition to the final DOA-GAN model: DOA-GAN without any attention (denoted as NA-GAN), baselines using the 1st-order (FOA-GAN) or 2nd-order attention (SOA-GAN) only, "DOA-GAN w/o L_{adv}", and "DOA-GAN w/o L_{det}", by removing the adversarial and image-level loss components L_{adv}, and L_{det}, respectively.

Table 1 summarizes the localization results of the various models, on USC-ISI CMFD test set.

Methods	Precision			Recall			F1		
	P	S	T	P	S	T	P	S	T
BusterNet [46]	93.71	55.85	53.84	99.01	38.26	48.73	96.15	40.84	48.33
ManTra-Net [47]	93.50	8.66	48.53	**99.22**	2.28	28.43	96.08	2.97	30.58
U-Net [38]	91.66	32.67	47.16	97.16	19.06	40.90	94.88	23.09	44.15
NA-GAN	95.87	35.30	59.32	96.91	41.64	52.32	95.40	33.25	55.94
FOA-GAN	95.06	52.82	71.17	97.24	43.32	62.06	96.04	43.43	65.90
SOA-GAN	95.53	50.94	70.20	98.17	40.86	66.58	**97.80**	42.50	67.19
DOA-GAN w/o ASPP-1	96.71	61.04	70.94	98.84	43.13	66.69	97.67	45.04	67.23
DOA-GAN w/o ASPP-2	96.08	60.70	65.20	99.43	39.18	68.76	97.62	44.13	65.41
DOA-GAN w/o $\mathcal{L}_{\text{adv}}$	95.80	72.30	83.60	96.27	60.32	79.10	96.01	63.25	80.45
DOA-GAN w/o $\mathcal{L}_{\text{det}}$	**97.35**	75.58	83.96	97.98	**64.19**	80.31	97.51	65.21	81.08
DOA-GAN	96.99	**76.30**	**85.60**	98.87	63.57	**80.45**	97.69	**66.58**	**81.72**

Table 1: The copy-move forgery localization results on the USC-ISI CMFD dataset using pixel-level precision, recall, and F1 score metrics for 3 classes: P, S, and T referring to Pristine, Source and Target, respectively [1]

First, we note that DOA-GAN clearly has the best results in most the metrics. Specifically, on the Source and Target classes, DOA-GAN shows superior performance by large margin over BusterNet and ManTra-Net.

Also, let us note that the inclusion of adversarial loss component (and Discriminator network) has only small contribution to the overall performance, as "DOA-GAN w/o L_{adv}" nearly matches the DOA-GAN final model. Apparently, the major boost in performance is due to the dual order attention module (NA-GAN vs DOA-GAN).

Image level evaluation results (for detection) are shown in Table 2 below:

Methods	Precision	Recall	F1
BusterNet [46]	89.26	80.14	84.45
ManTra-Net [47]	68.72	85.82	76.32
DOA-GAN	**96.83**	**96.14**	**96.48**

Table 2:Detection results on the USC-ISI CMFD dataset [1]

Again, it is clear that DOA GAN performs much better also for image-level detection.

Evaluation experiments also conducted on the popular datasets CASIA CMFD and CoMoFoD.

	Methods	Year	Precision	Recall	F1
Det	Block-ZM	2010	68.97	53.69	60.38
	DCT-Match	2012	63.74	46.31	53.46
	Adaptive-Seg	2015	93.07	25.59	40.14
	DenseFiled	2015	**99.51**	30.61	46.82
	BusterNet	2018	48.34	75.12	58.82
	DOA-GAN	2019	63.39	**77.00**	**69.53**
Loc	Block-ZM	2010	10.09	3.01	3.30
	DCT-Match	2012	8.80	1.90	2.40
	Adaptive-Seg	2015	23.17	5.14	7.42
	DenseField	2015	20.55	20.91	20.36
	BusterNet	2018	42.15	30.54	33.72
	DOA-GAN	2019	**54.70**	**39.67**	**41.44**

Table 3: The performance on the CASIA CMFD dataset [1]

Table 3 compared DOA-GAN's performance vs. other baselines on CASIA CMFD dataset. In contrary to USC-ISI, this dataset doesn't have source/target separation masks (treating both classes as 'forged'). Therefore DOA-GAN adapted to this scenario by modifying its localization head to predict only 1 output channel (indicating forged/pristine) instead of 3, and retraining.

As can be seen, DOA-GAN performs the best in terms of all metrics, for both localization and detection, except the precision in detection. It also better than BusterNet on all measures. Note that the results on BusterNet are different from the results reported in [4], as in the original BusterNet, the manipulation branch was trained on external image manipulation datasets, however, for fair comparison, both BusterNet and DOA-GAN were trained only on the USC-ISI CMFD dataset and MS COCO dataset.

	Methods	Year	Precision	Recall	F1
Det	Block-ZM	2010	51.72	20.87	29.74
	DCT-Match	2012	50.48	29.77	37.46
	Adaptive-Seg	2015	65.66	43.37	52.24
	DenseField	2015	**80.34**	20.10	32.15
	BusterNet	2018	53.20	57.41	55.22
	DOA-GAN	2019	60.38	**65.98**	**63.05**
Loc	Block-ZM	2010	2.90	2.50	1.73
	DCT-Match	2012	3.53	3.41	2.03
	Adaptive-Seg	2015	23.02	13.27	13.46
	DenseField	2015	22.23	23.63	22.60
	BusterNet	2018	**51.25**	28.20	35.34
	DOA-GAN	2019	48.42	**37.84**	**36.92**

Table 4: The performance on the CoMoFoD dataset [1]

Table 4 shows the evaluated performance on the CoMoFoD dataset. Again, DOA-GAN achieves the best performance except the precision in detection and localization. Note that different types of transformations are applied in this dataset to create copy-move manipulated images, e.g., translation, rotation, scaling, combination, and distortion. Various post-processing methods, such as JPEG compression, blurring, noise adding, and color reduction, are also applied to all forged and original images.

However, DOA-GAN is not perfect – it can fail in some cases. For example, when the copy region is extracted from the uniform background and pasted on the same background. It also might fail when the scale of the copied object has changed significantly. As shown in Figure 6 below, the backgrounds for the first example are uniform, and the scale of the copy-move regions are very small in the second example.

Figure 6: From left to right – input image, DOA-GAN result, and ground truth [1]

3. My Implementation

In general, I used pytorch [5] framework.

3.1 Data

To train, validate I used USC-ISI and COCO datasets same as [1].

USC-ISI has 10k images to test on. These are forged images used to evaluate localization. It is composed of MIT SUN2012 Database and MS COCO Dataset by copy paste random objects. To evaluate detection together with USC-ISI, COCO used with 10k images which are non-forged, called authentic.

To evaluate, I used CoMoFoD CMFD [6] and CASIA v2.0 CMFD [7]. CASIA v2.0 CMFD was taken from BusterNet repo [7] - it has hd5 file dataset of 1313 positive CASIA-CMFD samples, Both forged images and masks are included. To include the authentic images, I downloaded the data itself from Kaggle. The forged part is used for localization, and together with the authentic part I evaluate the detection.

The CoMoFoD CMFD dataset contains only copy-move forgeries.

These forged images and their masks appear in CoMoFoD-CMFD.hd5 file, and to evaluate detection on the non-forged images I used the original images (marked with "_O").

Every forged image (denoted as "_F_") is made of one of the following manipulations [7]:

- Translation - a copied region is only translated to the new location without performing any transformation (Images 1 to 40)
- Rotation - a copied region is rotated and translated to the new location (Images 41 to 80)
- Scaling - a copied region is scaled and translated to the new location (Images 81 to 120)
- Distortion - a copied region is distorted and translated to the new location (Images 121 to 159)
- Combination - two or more transformations are applied on a copied region before moving it to the new location (Images 160-200)

Also, it originally contains binary masks, but BusterNet created RGB masks by:

- Obtain the target copy by thresholding the difference between the manipulated image and its original
- Obtain the source copy by matching the target copy on the manipulated image using SIFT/ORB/SURF features
- Manually verify all obtained masks

Postprocessing methods applied on the images:

"JC" - JPEG compression, quality factor = [20, 30, 40, 50, 60, 70, 80, 90, 100] (noted as JC1, …, JC9 respectively)

"IB" - image blurring, μ = 0, σ2 = [0.009, 0.005, 0.0005]) (noted as IB1, IB2, IB3)

"NA" - noise adding, averaging filter = [3x3, 5x5, 7x7] (noted as NA1, NA2, NA3)

"BC" - brightness change, (lower bound, upper bound) = [(0.01, 0.95), (0.01, 0.9), (0.01, 0.8)],

"CR" - color reduction, intensity levels per each color channel = [32, 64, 128]

"CA" - contrast adjustments, (lower bound, upper bound) = [(0.01, 0.95), (0.01, 0.9), (0.01, 0.8)].

Therefore, for each fake image (we have 200 base, 40 for each manipulation (there are 5))– there are 25 post processing = 200*25=5000. Same for non-copy move images - 5000 images. 200 colored masks + 200 binary masks =400

The data sets are described in the following table:

Set	Training	Validation	Testing	Notes
USC-ISI CMFD **[7]**	80k	10K	Localization & detection -10k	Composed of MIT SUN2012 Database and MS COCO Dataset by copy paste objects.
MS COCO **[8]**	80k	10k	Detection - 10k	Used as pristine images counterparts for USC-ISI.
CoMoFoD CMFD [**[6], [7]**]	x	x	Localization – 5000 Detection 5000	Manipulations: translation, rotation, scaling, combination, and distortion. Contains RGB masks made by BusterNet
CASIA v2.0 CMFD**[7]**	x	x	Localization - 1313 Detection - 2626	

Table 5: Data sets summary

General transformation for all data used depicted in file “pairwise_transforms.py”. I created an API that transforms pairs of image and mask or single image only. Transformations used are convert to tensor, resize to 320X320, random horizontal flip for training set only, and normalization for images only. “datasets_api.py” contains API for USC-ISI, CASIA, CoMoFoD.

3.2 Model training configurations

Learning rates - `learning_rate_G=1e-3`; `learning_rate_D=1e-4`; `learning_rate_VGG =1e-4`. Following [1], different learning rates are used to train different parts in the model. The feature extraction module is based on the first three blocks of the VGG-19 network pretrained on the ImageNet dataset. The learning rate of the feature extractor (VGG-19) is set to 0.0001, i.e., is being trained additionally to use its pre trained weights (transfer-learning). The authors used two different learning rates for the generator and the discriminator networks, 0.001 and 0.0001, respectively.

Set Adam optimizer for the three models of G, D and VGG19 with `betas = [0.9,0.999]`. The hyper-parameters betas of Adam are initial decay rates used when estimating the first and second moments of the gradient, which are multiplied by themselves (exponentially) at the end of each training step (batch).

I used k = 20 for the top-k value in the 1st attention block, same as the authors described. The ASPP blocks are based on those used in DeepLabV3. Pytorch provides these in torch vision segmentation models, the relevant code copied to “aspp.py”

Patience - The learning rates are decreased by half (`factor=0.5`) when the training loss plateaus after 5*8=40 epochs (in the original paper they train on 80000 samples, but I train on 10000 random samples each time. To implement this, I used `torch.optim.lr_scheduler` that imports `ReduceLROnPlateau` function.

g_warmup_epochs - In order to prevent the discriminator over-train, there are `g_warmup_epochs=26` epochs to let net_g warm up. The authors in [1] trained 3 warmup epochs and because I train 10k each epoch instead of 80k, I needed to train net_g at least 3*8=24 epochs. Even if we consider the adversarial loss, we need to let G plot some reasonable masks before start training the discriminator.

`calc_dist_tform` - If using boundary loss, a precomputed distance map should be computed in the training dataset.

Localization weights -as a part of the training I tried different losses to improve results. These parameters control weather I use a specific loss in a training experiment (=1), or not (=0). `weight_focal(default=1); weight_gdl(default=0); weight_dice(default=0); weight_boundary(default=0).`

3.3 Loss Functions

When using cross entropy loss for the localization masks, the statistical distributions of labels (masks or detection scores) are very important in determining training accuracy. Particularly, cross-entropy loss does not work well with highly imbalanced classes. The more unbalanced the label distributions are, the more difficult the training will be. In our case, predicting the CMFD masks is a highly imbalanced task, where a forged image has many more pristine pixels than source and target pixels, as the copied areas are usually small.

Why is class imbalance a problem? Because most of the ML algorithms assume that the data is balanced, i.e., the data is equally distributed among all its classes. When training a model on an imbalanced dataset, the learning becomes biased towards the majority classes (pristine pixels). The model learns to perform well on the majority classes, but due to the lack of enough examples the model fails to learn meaningful patterns that could aid it in learning the minority classes. In case of CMFD, this might push the training to predict all the pixels as pristine in order to minimize the cross-entropy loss.

In order to improve the cross-entropy loss, I added class weights that give more emphasis to source and target classes. The general formula for weighted cross-entropy loss is:

$$CE = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c\in C} w_{i,c} * y_{i,c}\log(p_{i,c}) \underbrace{=}_{**} -\frac{1}{N}\sum_{\substack{i=1\\ c\in C}}^{N} w_{i,c}\log(p_{i,c})$$

(**) since $y_{i,c}$ is 1 if item i belongs to class c and 0 for others.

In particular, I used $w_{source} = w_{target} = 10$, $w_{pristine} = 1$ and $y_{i,c}, p_{i,c}$ represent tensors with labels for each pixel in real computation.

However, the improvement was not significant, and the core issue of cross entropy loss is not solved. In cross entropy loss, the loss is calculated as the average of per-pixel loss, and the per-pixel loss is calculated discretely, without knowing whether its adjacent pixels are boundaries or not. As a result, cross entropy loss doesn't see the global context, which is not enough for image level prediction.

3.3.1 Focal loss

$$FL(p_t) = -(1-p_t)^{\gamma}\log(p_t)$$

The Focal Loss [9] aims to help with imbalanced classes of pristine source and target, as there are much more pristine pixels the source and target. The Focal Loss adds a factor $(1-p_t)^{\gamma}$ to the standard cross entropy criterion, where p_t represents the predicted probability. Setting $\gamma > 0$ reduces the relative loss for well-classified examples $(p_t > 0.5)$, putting more focus on hard, misclassified examples (e.g., target and source). Here, I used $\gamma = 2$. Therefore, easily predictable pristine pixels (e.g., large background areas) will contribute much less to the overall loss, thus effectively reducing the class imbalance.

Additionally, the focal loss function can also have per-class weights (similarly to weighted cross-entropy).

3.3.2 Dice loss

Dice Loss [10] is a popular loss in semantic segmentation problems, which claims to improve the class imbalance problem. The Dice Coefficient measures the similarity between two samples – predicted and ground-truth:

$$D = \frac{2\sum_i^N p_i * g_i}{\sum_i^N p_i^2 + \sum_i^N g_i^2}$$

Dice coefficient can also be seen as a form of IoU (Intersection-over-Union) overlap between two sets and in terms of sets it is

$$\frac{2 * intersection}{union}$$

An observation dice coefficient is the same as F1-score (harmonic mean of precision and recall)

$$F1 = \frac{1}{\frac{1}{2}\left(\frac{1}{recall} + \frac{1}{precision}\right)} = \frac{TP}{2TP + FN + FP}$$

We know that union equals to TP + FN + FP, and intersection equals to TP.

Dice Loss aims to maximize the Dice Coefficient (or minimize 1-DiceCoefficient).

A potential problem with dice, is that it can have high variance. Getting a single pixel wrong in a tiny object can have the same effect as missing a whole large object, thus the loss becomes highly dependent on the current batch. Originally, Dice coefficient is defined for a binary classification problem (e.g., foreground/background). The extension to multi-class problem (which is our case) is done by calculating Dice Coefficient per class (P/S/T), then averaging between classes.

3.3.3 Generalized dice loss (GDL)

As another experiment to improve results, I used the Generalized Dice Loss [11] , which is an improvement for Dice Loss for coping with extremely imbalanced data.

Its form is:

$$GDL = 1 - 2\frac{\sum_{l=1}^{C} w_l \sum_i g_{li} p_{li}}{\sum_{l=1}^{C} w_l \sum_i g_{li} + p_{li}}, w_l = \frac{1}{\left(\sum_i^N g_{li}\right)^2}$$

Where the class weighting w_l is determined on the fly such that the contribution of each label is corrected by the inverse of its volume, thereby reducing the well-known correlation between region size and the standard Dice Coefficient.

3.3.4 Surface (Boundary) loss

As the resulted masks didn't exceed my expectations, I tried to add the Boundary Loss [12] additionally to GDL. It aims to reduce issues related to regional losses in highly unbalanced segmentation problems. It uses the distance metric on the contours or shapes, not regions.

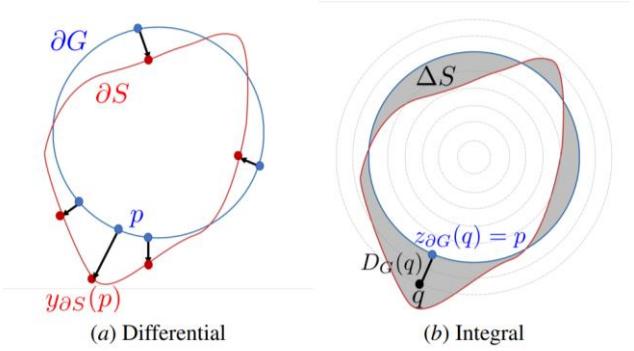

Figure 7: S is the predicted region G is the ground truth. ΔS denotes the region between the two contours ∂G, ∂S, the loss is actually the subtraction (union-intersection)[12]

The following equation describes boundary loss [12]

$$\boldsymbol{L_B(\theta)} = \int_{\boldsymbol{\Omega}} \boldsymbol{\phi_G(q) s_\theta(q)}\, \mathrm{d}\boldsymbol{q}$$

Where $s_\theta(q)$ is the predicted SoftMax probability output of the network for the relevant class at pixel q, and $\boldsymbol{\phi_G}: \Omega \rightarrow R$ denotes the level set representation of boundary ∂G:

$$\boldsymbol{\phi_G(q)} = \begin{cases} -D_G(q) & if\ \ q \in G \\ D_G(q) & otherwise \end{cases}$$

D_G is a distance map with respect to boundary ∂G, i.e., $D_G(q)$ is the Euclidean distance between pixel $q \in \Omega$ and its nearest point on contour ∂G. $\boldsymbol{\phi_G}$ is computed per class {P, S, T} on the fly, during the batch fetching stage (since it depends only on the ground-truth labels).

In practice, the loss $\boldsymbol{L_B(\theta)}$ is computed for each class as a multiplication between output probabilities $s_\theta(q)$ and distance maps $\phi_G(q)$, and then averaged across locations and classes.

For each class (pristine, source, target), the level-set distance map $\boldsymbol{\phi_G}$ is computed from the given ground-truth masks as:

```
distance(negmask)*negmask - (distance (posmask) - 1) * posmask
```

where posmask is 1 if for pixels belonging to the class, and 0 otherwise. negmask is the binary complement of posmask. distance is the 2D Euclidean distance transform function. i.e., for each pixel x[i] it gives values of the Euclidean distance:

$$y_i = \sqrt{\sum_i^n (x[i] - b[i])^2}$$

Where $b[i]$ is the background point (value 0) with the smallest Euclidean distance to input points $x[i]$, and n is the number of dimensions, in our case `n=2`.
Note that the (-1) in the posmask formula above, means that the distance is zero for pixels that are right at the boundary (i.e., adjacent to a non-object pixel).

As illustrated in **שגיאה! מקור ההפניה לא נמצא.** , `(distance(posmask) - 1) * posmask` is the distance map from inside the object to its boundaries, and `distance(negmask)*negmask` is the distance between the background around the object and its boundaries.

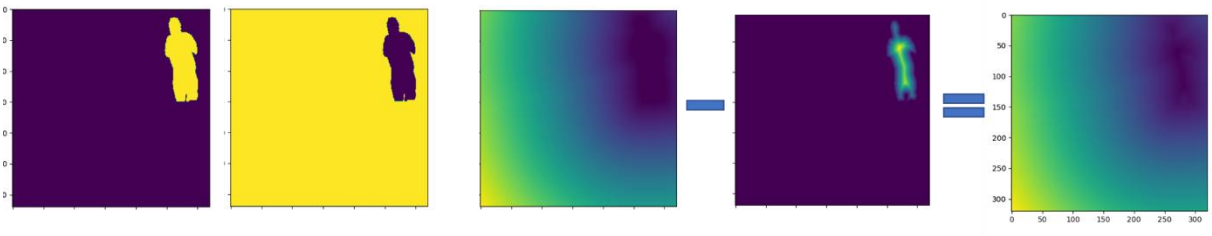

*Figure 8: Distance map for class 0 (target class). From left to right – positive mask, negative mask, distance(negmask)*negmask, (distance (posmask) - 1) * posmask, resulted map. Yellow in the distmap means very high values as it is very far from the object.*

3.3.5 Summary table

A summary table inspired by [13]

Loss Type	Loss Function	Use cases
Distribution-based	Binary Cross-Entropy	Works best in equal data distribution among classes scenarios
	Weighted Cross-Entropy	Widely used with skewed dataset Weighs positive examples by β coefficient
	Focal Loss	works best with highly imbalanced dataset down-weight the contribution of easy examples, enabling model to learn hard examples
Region-based	Dice Loss	Inspired from Dice Coefficient, a metric to evaluate segmentation results
	GDL	Weighted dice loss
Boundary	Boundary loss	Aims to minimize the distance between ground truth and predicted segmentation. Usually, to make the training more robust, boundary-based loss functions are used with region-based loss.

3.4 Training

In this part the training methodology is presented. Same as DOA-GAN, the training set includes the USC-ISI dataset with 80000 forged images and the COCO-2014 dataset with 80k non forged images. As a starter, the model was trained without the GAN components from few reasons: The first reason is to get a stable model that outputs reasonable masks and detection scores, since GAN is harder to train. The second reason is that the discriminator should help to train the generator and according to the authors results it didn't improve much. The relevant parameter to indicate that is called `adversarial_loss`. If it is set to true, then `net_d` is created, and the relevant loss is considered in the general loss. As an experiment I tried to use ls-GAN loss which uses MSE (mean square error) between prediction and target its training is better converges then vanilla GAN and I got more stable training. Let us first focus on the training without the network D.

In previous section (3.3) different losses were presented, which the authors didn't use, but I decided to use because the results at the beginning with original loss (as described in section 2.2) didn't perform well.

Back to the training function. It starts with defining the relevant loss as configured. These losses above are intended to train the localization branch.

The detection loss is the same (almost) as DOA-GAN's author defined - `BCEWithLogitsLoss` binary cross entropy with logits. Why with logits? Well instead of adding sigmoid layer (or `SoftMax` if the output was more than one class, but in this case, it is a detection score) in the model we do it in the loss function itself. This version is more numerically stable than using the sigmoid layer followed by a `BCEloss`.

During training the collected metrics are accuracy for detection/localization, and localization recall for each class (source, target, pristine) named respectively as `acc_det`, `acc_loc`, `rec_t`, `rec_s`, `rec_p` etc. For the binary case: `rec_f`, `rec_p` represents forged and pristine.

The final loss formulas:

(1) $L_{loc} = weight_{focal} * L_{focal} + weight_{gdl} * L_{gdl} + weight_{dice} * L_{dice} + weight_{boundary} * L_{boundary}$

(2) $L_{tot} = L_{adv} + \alpha * L_{loc} + \frac{1}{2} * \beta * L_{det} + \frac{1}{2} * \beta * L_{det\_pristine}$

Main training loop

For each epoch:

Zero running metrics

Set `net_g` to train mode – layers such as `BatchNorm` and Dropout acts differently than in eval mode. In training we do use Dropout, in eval we don't. During training, `BatchNorm` layer keeps a running mean and variance. The running stats are updated with a default momentum of 0.1 during training. Mathematically, the update rule for running statistics here is $\widehat{x_{new}} = (1 - momentum) * \widehat{x_{old}} + momentum * x_{current}$, where $\widehat{x_{old}}$ is the estimated statistics so far and $x_{current}$ is the new observed value.
During evaluation, this running mean/variance is used for normalization. A note `no_runing_stats (track_running_stats=False)` flag for batch normalization is used in my training. In this case in training mode the running mean/variance won't be kept and thus won't be used in eval mode. When I used the default BatchNorm, there was a big gap between training and validation performance. The validation set had worse results and didn't converge. And with no running stats I got similar performance for train and validation.

For each mini- batch:

Assigning the relevant parts of `data` variable into `forged_images` and `gt_masks` variable names.

Zero the gradient of `net_g` - accumulation of gradients happens when backward is called on the loss tensor, i.e., performing update to weights and biases computed. Otherwise, the gradient would be a combination of the old gradient, which has already been used to update the model parameters, and the newly computed gradient. Therefore, it would point in some other direction than the intended direction towards the minimum.

Forward pass of the **forged images** in `net_g`. The results are prediction masks and scores for each image in the mini batch represented `as pred_mask, score_forged`.
Both of shape `(batch_size,3,320,320)` and (`batch_size,1`) respectively.
SoftMax was not applied on `pred_mask` yet.

Calculating localization loss `L_loc` which is the weighted sum of `L_boundary, L_focal, L_dice, L_gdl` (if they are configured of course) between `pred_mask` and `gt_mask`.

Calculating the detection loss

$$L_{det} = detection\_loss(score\_forged, torch.ones\_like(score\_forged))$$

Train `net_d` if `run_gan_in_epoch` is true – will elaborate later.

Calculating the total loss as in Eq. (2)

Forward pass on the pristine batch, to train the detection branch.
I run `net_g(pristine_images)` and get their detection scores (the predicted masks are not relevant), and computing $L\_det\_pristine$.

`L_det_pristine.backward()` –

since `L_det_both = 0.5 * (L_det + L_det_pristine)` is the detection loss for both pristine and forged and `L_det` is part of `L_total` which has already done backprop, the one that left is `L_det_pristine`. I couldn't do backward at the same time since it is another batch of images, and it might raise some out of GPU memory exception.

Calling `optimizer_g.step()` which means that after computing the gradients for all tensors in the model, the optimizer iterate over all parameters (tensors) it is supposed to update (`requires_grad=True` and `is_leaf=True`) and use their internally stored grad to update their values.

Update running metrics that are showed in tensorboard graphs [see Figure 9]

So, this is basically the training loop. After that there is the validation loop which looks almost the same, except that no gradients are computed, and no optimization is done.

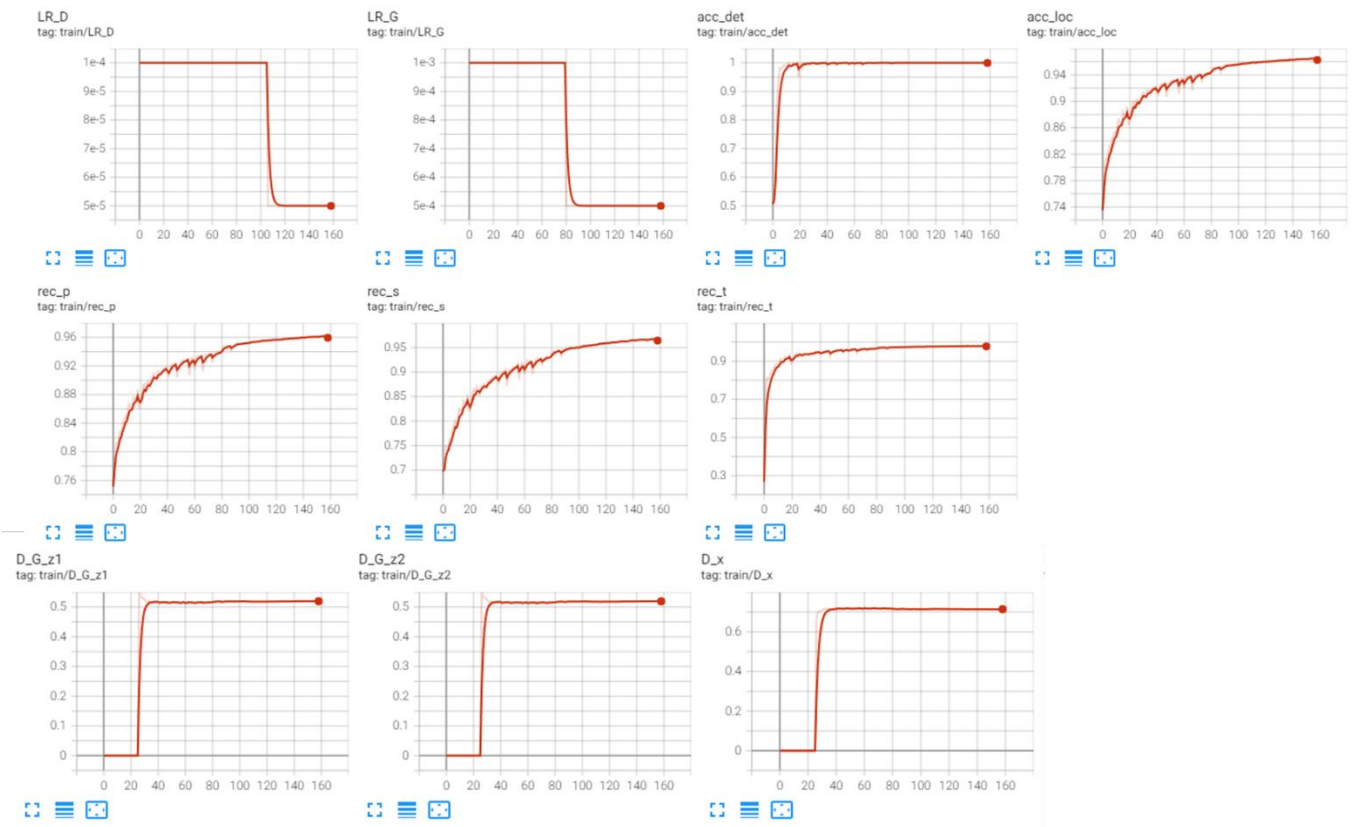

Figure 9: Tensorboard of LS-GAN training. Running metrics are updated for each epoch.

Let us explain how network D is calculated:

Network D is trained with real masks and generated masks, both with the forged images.

I modified the discriminator by adding spectral norm[[14], [15]] after each convolution layer, which stabilizes the GAN training. The spectral norm is the maximum singular value of a matrix.

Train `net_d`:

Set `requires_grad=True` for each of `net_d`'s parameters – in the end of this loop (starting from the second iteration) I set those to false, because I update `net_g` and make another forward pass with `net_d`, and in this forward pass I don't need to compute the gradients of D again – it saves time.

Zero the gradient of `net_d`

Forward pass of `net_d` – forged images and their "real" (ground truth) masks are passed through the Patch-GAN [16] based discriminator. The output is a tensor of shape `(batch_size, 1, 10, 10)` – for every given image the output is a $10x10$ patch. Every pixel in the output describes a patch in the original image, and the loss of netD looks at the average of all this patches. The score of the pixel tells if the patch is real or fake.

Computing the loss `errD_real` between the output of the discriminator and "real" labels. label 1 means that the mask is ground truth.

Calculate gradients for D in backward pass - `errD_real.backward`

Computing the average output (across the batch) of the discriminator for all batch with `gt_masks` - `D(x)` – it should start close to 1 then theoretically converge to 0.5 when G gets better, because at the beginning G outputs garbage and it is easy for D to guess "correctly".

Another forward pass, but this time with the predicted masks generated by `net_g`.

Compute the loss `errD_fake`, this time with "fake" labels which are 0. Finally, backward.

The final loss of netD is the sum `errD = errD_real + errD_fake`.

Computing the average output (across the batch) of the discriminator for all batch with masks generated by G – $D\big(G(z1)\big)$ – it should start near 0 and converge to 0.5 as G gets better.

Update D according to the authors policy - When the discriminator loss decreases to 0.3, we freeze the discriminator until the loss increases, therefore:

```
if errD > gan_disc_loss_thr:
        optimizerD.step()
```

Set `requires_grad=False` for each of `net_d` parameters. No need to save gradient for D because we don't do any step for D

Since we just updated D, perform another forward pass of all-fake batch through D. Then Calculate G's loss based on this output (`adv_loss`)

Computing the average output (across the batch) of the discriminator for all batch with masks generated by G – $D\big(G(z2)\big)$ – it should start near 0 and converge to 0.5 as G gets better.

A note – why computing backward separately on `errD_real` and `errD_fake` and not on `errD`? Well, it is a trade-off between time and GPU memory. Time – it might be slower performing backward twice. Memory – keeping memory consumption lower by 2 "small" auto grad graphs instead of one large, so we can double the batch size for example. Backward clears the graph.

3.5 Testing

This part consists of testing each dataset separately for localization and detection: For localization task each dataset was tested separately. The metrics used were F1, Precision and Recall from torch metrics [17]. The computed metrics are defined as follows:

- Localization (pixel-level evaluation) – 3 or 2 classes (target, source and pristine or target +source and pristine) to evaluate localization, and the applied reduction specified by the "Average" parameter calculates the metric for each class separately and returns the metric for every class. On top of this parameter, there's a way to average multi-dimensional multi class inputs called `mdmc_average` and its value is 'samplewise', which means that the statistics are computed separately for each sample in the batch (N) axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes, as the N dimension within the sample, and computing the metric for the sample based on that. The localization evaluation is in the `test_localizaition.py` script.

- Detection (image-level evaluation) – here the number of classes equals to 1, from few reasons: first, the output score in this case is $(N, 1)$. Second, we have positive samples (forged) and negative samples (non-forged). The average parameter is none – it calculates the metric for each class separately and returns the metric for every class.

All images and masks are being transformed before testing - transform to tensor representation, resize to $320\ x\ 320$, normalization

3.6 Results and future work

As part of my implementation, I conducted many experiments with different hyper-parameters and training configurations but chose the best ones to demonstrate. Table 6 below shows 4 experiments: 2 experiments without the adversarial loss component (`without` L_{adv}) and 2 that do use it (`Full GAN`), according to the author in [1].

The columns of this table are described here in detail. The first one, "No running stats", means that `BatchNorm` layers run with `track_running_stats=False`, unlike the default `BatchNorm` behavior (as described in training section 3.4). This is a key hyper-parameter that made the model get better results (much better than vanilla `BatchNorm`), and the 4 experiments shown use it. It is worth to mention that I also tried using `LayerNorm` normalization layer instead of `BatchNorm`, and although it gave good results, the best results were obtained with the modified `BatchNorm`.

The second column is the weight that was given to the detection loss L_{det} – alpha detection (αdet), which controls the balance between the localization and detection loss terms. The third one is the usage of the vanilla Cross Entropy loss for the pixel-level localization, as described in [1]. The 4th is the usage of Focal Loss, which is discussed in detail in section 3.3.1. The 5th and 6th column describe the usage of Generalized Dice Loss (GDL) and Boundary Loss (BL), respectively. Both GDL and BL are described widely in sections 3.3.3 and 3.3.4 respectively. 0.1 is the given weight for Boundary Loss.

Those experiments that are not using the adversarial loss were conducted earlier than others, to get stable training, as it is well known that GAN is more difficult to train and more sensitive to hyper-parameters compared to standard CNNs. The experiment called `no_runing_stats` is a combination of `track_running_stats=False`, Focal Loss and αdet=0.5. The second experiment `gdl_boundary_nrs` stands for combined GDL and BL (`nrs=no running stats`).

The experiments in "Full GAN" section are `lsgan_gdl_bd_nrs` and `nrs_ce_lsgan`. The former uses GDL & BL for localization loss, while the latter uses vanilla CE.

Both configurations are trained with Least-Squares GAN (LS-GAN) instead of the vanilla GAN:

$$L_D^{GAN} = E\left[\log\left(D(x)\right)\right] + E\left[\log\left(1 - D(G(z))\right)\right]$$
$$L_G^{GAN} = E\left[\log\left(D(G(z))\right)\right]$$

$$L_D^{LSGAN} = E[(D(x) - 1)^2] + E[D(G(z))^2]$$
$$L_G^{LSGAN} = E[(D(G(z)) - 1)^2]$$

Figure 10: LS-GAN vs vanilla GAN [18]

Vanilla GAN was used in some experiments (not shown here), but eventually I preferred LS-GAN which turned up to be more stable to train and had better convergence.

	Methodology/ Experiment name	No running stats	Alpha detection	Cross entropy	Focal loss	GDL	Boundary Loss
Without L_{adv}	`no_runing_stats`	✓	0.5		✓		
	`gdl_boundary_nrs`	✓	0.3			✓	✓(0.1)
Full GAN	`lsgan_gdl_bd_nrs`	✓	0.3			✓	✓(0.1)
	`nrs_ce_lsgan`	✓	0.3	✓			

Table 6: Experiments that were conducted

3.6.1 Results

In this section the results tables are described and compared to [1]. Table 7 below shows results on USC-ISI test dataset, whose training and validation sets used to train my model same as [1]. As a whole, the model of `gdl_boundary_nrs` got better results. And when used the adversarial loss, the model of `lsgan_gdl_bd_nrs` seems to be better as a whole. My models got significantly better results than [1].

Experiment	Precision			Recall			F1		
	P	S	T	P	S	T	P	S	T
`no_runing_stats`	**99.56**	61.01	75.49	92.90	**91.44**	99.12	96.06	71.96	84.32
`gdl_boundary_nrs`	99.15	**88.59**	**95.23**	99.25	89.71	96.62	**99.20**	**89.04**	**95.89**
`DOA-GAN without` L_{adv} `[1]`	95.80	72.30	83.60	96.27	60.32	79.10	96.01	63.25	80.45
`lsgan_gdl_bd_nrs`	95.89	77.52	94.19	**99.50**	55.67	94.68	97.62	61.69	94.36
`nrs_ce_lsgan`	99.15	71.22	82.59	95.45	88.62	**99.22**	97.17	76.40	89.87
`DOA-GAN[1]`	96.99	76.30	85.60	98.87	63.57	80.45	97.69	66.58	81.72

Table 7: Localization results for USC-ISI test set

In detection results (shown in Table 8) almost no difference between the models I trained, besides it has better results than [1].

Experiment	Precision	Recall	F1
no_runing_stats	100.00	100.00	100.00
gdl_boundary_nrs	100.00	100.00	100.00
DOA-GAN without L_{adv}[1]	95.45	93.09	94.25
lsgan_gdl_bd_nrs	99.00	100.00	99.00
nrs_ce_lsgan	100.00	100.00	100.00
DOA-GAN[1]	96.83	96.14	96.48

Table 8: Detection results for USC-ISI

The localization and detection results for CASIA and CoMoFoD datasets are described below in **שגיאה! מקור ההפניה לא נמצא.** I used the models that were trained only on USC-ISI, same as [1]. The scores in in Table 9 describe the union of source and target classes i.e., no source target differentiation. The model of [1] localized better, even though the gap is not significant. no_runing_stats detected better than [1]. Generally, the models that I trained without the adversarial loss got better results than those with L_{adv}.

Dataset	Experiment	Localization				Detection		
		Precision	Recall	F1		Precision	Recall	F1
CoMoFoD	no_runing_stats	29.9729	**49.6347**	27.4558		**99.98**	**99.94**	**99.96**
	nrs_ce_lsgan	31.5311	45.6249	25.5044		96.5655	24.1800	38.6756
	DOA-GAN [1]	**37.84**	48.42	**36.92**		65.98	60.38	63.05
CASIA	no_runing_stats	28.686	**46.2625**	30.231		92.9394	**92.2315**	**92.5841**
	nrs_ce_lsgan	25.6598	30.7978	20.797		**100**	0.1523	0.3042
	DOA-GAN [1]	**54.70**	39.67	**41.44**		63.39	77.00	69.53

Table 9: Localization and detection scores for CASIA and CoMoFoD

3.6.2 Visual results

Output masks of the Generator network alongside the ground truth mask and original image

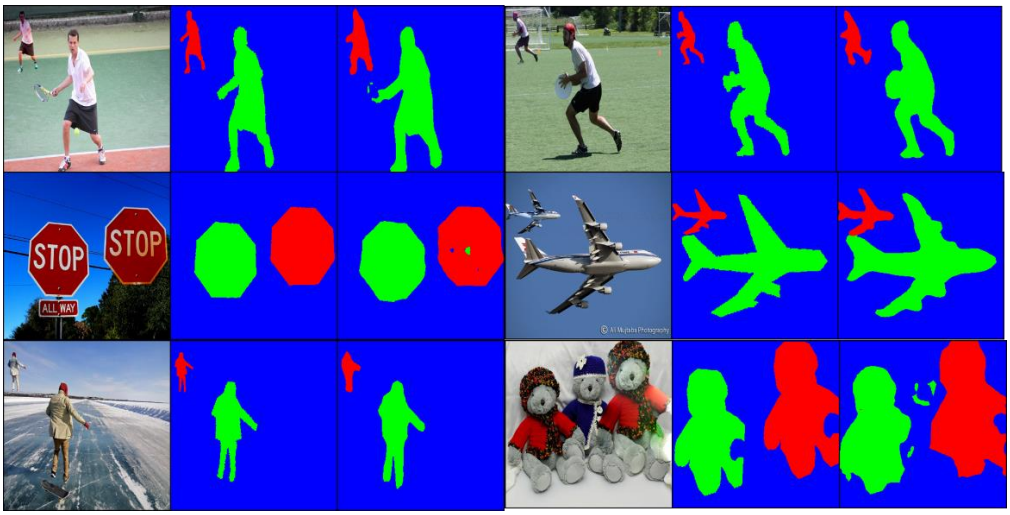

Figure 11: test USC-ISI no running_stats.pt from left original image, ground truth mask(middle) and prediction.

3.6.3 Future work

In my work I focused on two models that the authors of [1] showed the best results for- Full GAN and DOA GAN without discriminator.

There is room to conduct much additional research, specifically:

- Use a better GAN loss as WGAN[19]
- Train the model on other popular benchmark datasets like CASIA and CoMoFoD that may have used slightly different copy-move techniques which are not fully covered by the original training.
- Experiment with additional forgery techniques like splicing or video as described in a supplementary file of [1].

In summary, the research area of visual forgery detection is gaining popularity and provides demand for more research in the future.

4. References

[1] A. Islam, C. Long, A. Basharat, and A. Hoogs, “DOA-GAN: Dual-Order Attentive Generative Adversarial Network for Image Copy-Move Forgery Detection and Localization,” in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Seattle, WA, USA, Jun. 2020, pp. 4675–4684. doi: 10.1109/CVPR42600.2020.00473.

[2] “DOAGAN pytorch.” https://github.com/s1v1/DOAGAN.git

[3] L. Chen, D. Zhao, and C. Heipke, *Complementary Features Learning from RGB and Depth Information for Semantic Image Labelling*. 2019.

[4] Y. Wu, W. Abd-Almageed, and P. Natarajan, “BusterNet: Detecting Copy-Move Image Forgery with Source/Target Localization,” *Computer Vision – ECCV 2018*, vol. 11210, pp. 170–186, 2018, doi: 10.1007/978-3-030-01231-1_11.

[5] “PyTorch.” https://pytorch.org/

[6] “CoMoFoD dataset.” https://www.vcl.fer.hr/comofod

[7] “BusterNet repository.” https://github.com/isi-vista/BusterNet

[8] “COCO dataset.” https://cocodataset.org/#home

[9] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, “Focal loss for dense object detection,” 2017, pp. 2980–2988.

[10] F. Milletari, N. Navab, and S.-A. Ahmadi, “V-net: Fully convolutional neural networks for volumetric medical image segmentation,” 2016, pp. 565–571.

[11] C. H. Sudre, W. Li, T. Vercauteren, S. Ourselin, and M. J. Cardoso, “Generalised Dice overlap as a deep learning loss function for highly unbalanced segmentations,” *arXiv:1707.03237 [cs]*, vol. 10553, pp. 240–248, 2017, doi: 10.1007/978-3-319-67558-9_28.

[12] H. Kervadec, J. Bouchtiba, C. Desrosiers, E. Granger, J. Dolz, and I. B. Ayed, “Boundary loss for highly unbalanced segmentation,” *Medical Image Analysis*, vol. 67, p. 101851, Jan. 2021, doi: 10.1016/j.media.2020.101851.

[13] S. Jadon, “A survey of loss functions for semantic segmentation,” 2020, pp. 1–7.

[14] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida, “Spectral Normalization for Generative Adversarial Networks,” *arXiv:1802.05957 [cs, stat]*, Feb. 2018, Accessed: Dec. 09, 2021. [Online]. Available: http://arxiv.org/abs/1802.05957

[15] “Spectral Norm.” https://pytorch.org/docs/stable/generated/torch.nn.utils.spectral_norm.html

[16] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, “Image-to-Image Translation with Conditional Adversarial Networks,” *arXiv:1611.07004 [cs]*, Nov. 2018, Accessed: Feb. 26, 2022. [Online]. Available: http://arxiv.org/abs/1611.07004

[17] “TorchMetrics.” https://torchmetrics.readthedocs.io/en/latest/

[18] “LSGAN.” https://github.com/hwalsuklee/tensorflow-generative-model-collections

[19] M. Arjovsky, S. Chintala, and L. Bottou, “Wasserstein generative adversarial networks,” 2017, pp. 214–223.