Watermarking DNNs applied

From paper to code

Fabian Greavu, Giovanna Scaramuzzino

January 2021

Contents

1 Abstract

In June 2020 a new paper [6] involving watermarking DNNs was released. We are going to first introduce its main concepts regarding the watermarking idea, its working principles and implementation steps. Then we will show what implementation approaches did we follow and what parts needed to be tested. Finally, we show how the paper results can be achieved in practice and we propose a full code for that task.

2 Introduction

Machine learning has gone beyond expectations in recent years in terms of image recognition, speech recognition, natural language processing. Deep Neural Networks (DNNs) are used to perform these tasks, but to be trained they need a lot of data and a lot of computing resources. Some large models are trained on millions of images and may need hundreds of GPUs running for entire weeks for training tasks. Therefore, to reduce execution times, many companies have started sharing pre-trained models, which include costs for commercial use.
Considering all these elements (costs of recourse computations, data necessary for training), the pre-trained networks are considered the *intelligent properties* (IPs) of the owners, which must be protected against copyright infringement or breaking of license agreements. Furthermore, the models could be used in a malicious way.
For this purpose **DNN Watermarking** was introduced, which conceals watermark information in a published model for ownership identification and copyright protection.
Various methods have been developed to watermark DNNs for the **classification tasks**, but they cannot be directly applied to DNNs for **image processing tasks** because of different output sizes, different processing approaches (finding decision boundaries vs finding low-dimensional manifold over inputs) and overparameterization.

2.1 Watermarking paper description

With that in mind, the authors of [6] have created a new **black-box framework for image processing** that allows encoding the watermark trough specific model inputs (called *trigger keys*) and the expected model outputs (called *verification keys*).
Therefore, the basic idea for watermarking was to fine-tune the DNN model to ensure that, giving as input the trigger image, the output image would approximate verification images.
Before explaining the proposed method, some definitions are needed:

- $\mathrm{M}(\cdot;\theta)$: DNN model of image processing parameterized by θ
- M: space of all DNNs solving the same task M

- X: set of images
- θ_o: parameters of M trained on X
- θ^*: paremeters of M after watermarking
- K: space of all trigger keys
- $\mu(\cdot)$: a visual quality measure for images (es: PSNR)

2.2 Black-box modules

The proposed paper on black-box method for watermarking image processing DNNs is made of 3 modules: generation, embedding and verification.

1. **Watermark generation**
   This module deals with the generation of *trigger key* image $\mathbf{K} \in R^{\mathrm{MxN}}$ and *verification key* image $\mathbf{S} = \mathrm{G}(\mathbf{K}) \in R^{\mathrm{MxN}}$.
   Let U(a, b) the uniform distribution on the interval [a,b].
   The *trigger key* image $\mathbf{K} \in R^{\mathrm{MxN}}$ is sampled from the uniform distribution: $\mathbf{K}(\mathrm{i},\mathrm{j}) \sim \mathrm{U}(0,1)$ and the *verification key* image $\mathbf{S} = \mathrm{G}(\mathbf{K}) \in R^{\mathrm{MxN}}$ is generated from the operation $\mathrm{G}(\cdot)$. This operator, for denoising DNN, is defined as $\mathrm{G}(\mathbf{K}) = \mathbf{K} - \nabla\mathbf{K}$, where ∇ denotes the gradient operator.
   The key pair is known only by the owner of the model, and when M, N are sufficiently large, two owners are unlikely to get very similar trigger images.

Fig. 1: Trigger image (left) - Verification image (right)

2. **Watermark embedding**
   This module deals with training the host DNN model in order to embed the watermark information. This is done by fine-tuning the model with training data and the trigger image $\mathbf{K}$ such that $\mathrm{M}(\mathbf{K})$ well approximates the expected verification image $\mathbf{S}$ (shown in Figure 2).

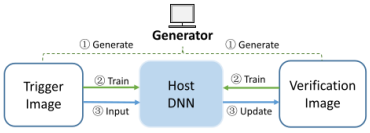

Fig. 2: Block diagram of the proposed framework: watermark embedding

The following loss function is used:

$$\ell(\theta) = \ell_{\mathrm{d}}(\theta) + \lambda \ell_{\mathrm{w}}(\theta) \tag{1}$$

with:

- $\ell_{\mathrm{d}}(\theta)$: loss regarding the original denoising
$$\ell_{\mathrm{d}}(\theta) = \frac{1}{2\mathrm{K}} \sum_{i=1}^{\mathbf{K}} \|(\mathrm{M}(\mathbf{X}_{\mathrm{i}}; \theta) - \mathbf{Y}_{\mathrm{i}})\|_2^2$$
$\mathbf{X}_{\mathrm{i}}$: i-th input image in the original training set; $\mathbf{Y}_{\mathrm{i}}$: the ground truth of $\mathbf{X}_{\mathrm{i}}$
- $\ell_{\mathrm{w}}(\theta)$: loss regarding watermark embedding
$$\ell_{\mathrm{w}}(\theta) = \|(\mathrm{M}(\mathbf{K}; \theta) - \mathbf{S})\|_2^2$$
- λ: strength of embedding

The training was stopped until $\ell_{\mathrm{w}}(\theta)$ was sufficiently small and the verification key $\mathbf{S}$ was updated by $\mathrm{M}(\mathbf{K}; \theta^*)$ after training the DNN.

3. **Watermark verification**
This module deals with how to check watermark presence. Using *trigger image* $\mathbf{K} \in R^{\mathrm{MxN}}$ and *verification image* $\mathbf{S} \in R^{\mathrm{MxN}}$ ownership of a model $\mathrm{A}(\cdot)$ can be demonstrated inserting $\mathbf{K}$ into $\mathrm{A}(\cdot)$ and calculating the distance in (2) between $\mathbf{S'} = \mathrm{A}(\mathbf{K})$ and $\mathbf{S}$. If the distance is lower than the threshold $\eta = 6,07 * 10^{-3}$, the ownership is verified. The threshold was determinated by probabilistic approach (more details are given in [6]).
Before the calculation, $\mathbf{S'}$ and $\mathbf{S}$ are normalized to [0,1].

$$\mathrm{d}(\mathbf{S}, \mathbf{S'}) = \frac{1}{MN} \|\mathbf{S'} - \mathbf{S}\|_2 \leq \eta \quad , \qquad \mathrm{d}(\mathbf{S}, \mathbf{S'}) \in [0, 1] \tag{2}$$

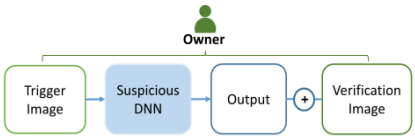

Fig. 3: Block diagram of the proposed framework: watermark verification

In addition to these three models, an **Auxiliary Copyright Visualizer** was proposed for visualizing the watermark information because the verification image has no visual implication due to the need of randomness.
The proposed method is a generative DNN as a plug-and-play module $\mathbf{R}(\cdot; \phi) : \mathrm{R}^{\mathrm{MxN}} \rightarrow \mathrm{R}^{\mathrm{PxQ}}$ with parameter vector ϕ. This module was trained to map the verification key $\mathbf{S}$ to a recognizable copyright image $\mathbf{I}$ (example: fig. 4) by minimizing:

$$\mathbf{r}(\phi) = \|\mathbf{R}(\mathbf{S}; \phi) - \mathbf{I}\|_2^2 \tag{3}$$

The module R will output the copyright image when model has watermark embedded. The flow is to give the trigger image as input to M, then use the output and the verification image (fig 1) in order to visually display the watermark sign (fig 4). If watermark is degraded, so it will be the output.

Fig. 4: Watermarking sign for visual check

2.3 Watermarking process principles

There are three principles for DNN watermarking about image processing that need to be kept:

1. **Fidelity**: don't degrade the processing performance of the host model
$$\mu(\mathrm{M}(\mathbf{X};\theta^*)) \approx \mu(\mathrm{M}(\mathbf{X};\theta_\mathrm{o})), \text{ s.t. } \forall \mathbf{X} \in X$$

2. **Uniqueness**: any DNN model for the same task cannot map the *trigger image* to the known *verification image* without related knowledge
$$\forall \mathbf{K} \in K,\ \mathrm{A}(\mathbf{K};\psi) = \mathrm{M}(\mathbf{K};\theta^*) \text{ iff } \mathrm{A} = \mathrm{M}, \psi = \theta^*$$
where $\mathrm{A} \in M$ and ψ encodes the parameters that have not been tuned on $(\mathbf{K}, \mathrm{M}(\mathbf{K}, \psi))$

3. **Robustness**: preserve the watermark information under attacks (i.e., small perturbation ϵ on θ^*)
$$\mathrm{M}(\mathrm{K};\theta^* + \epsilon) \approx \mathrm{M}(\mathbf{K};\theta^*)$$

3 Recreating results

The paper [6] had lots of visual results and tables showing how the watermarking was a success for RED and DnCNN. We will only be focusing on the **denoising** network (DnCNN) to do all the tests, also using the same **dataset** from the paper (Img12 and BSD68) with proposed image size (40x40).
As first step we asked for the original **code** from Watermarking Dnns paper [6] and the authors kindly released it even if not complete and poorly documented. It can be found in this github repository [7].
All the code generated by there can be found in **our repository** [3] and we hope that in future the authors will incorporate our work because we have introduced the documentation and it is fully working.

Following section will explain all the tools and libraries used for the repository [3].

3.1 Python and Tensorflow env

All the infos can be checked out in the repository's README file.
For our setup it is needed a **python 3.7** with **tensorflow 1.x** environment (tested on tf 1.15 and it works perfectly, similar versions can be also checked out but not 2.x). It is advised to use a **virtual environment** or **conda** in order to keep a separate environment and not corrupt other projects.
Other necessary libs can be installed using pip:

- **Pillow**, **Matplotlib**, **Numpy**
- **OpenCV** (no need for the contrib, the official one is enough).

With all this installed we can begin to use our scripts.

3.2 Project Tree

In order to keep track of source files, code and dataset, the repository is organized as following:

- **Datasets**: inside **dataset** folder, directory **Train400** contains 400 images from Berkeley segmentation dataset [1] and directory **Train_KTH_TIPS** contains 400 square images from KTH-TIPS dataset [2].
- **Starting model**: inside **DnCNN_weight** there is the last checkpoint (of epoch 45) of the dncnn model already trained. We use it as starting model e.g. the proprietary model we want to watermark
- Folder **test_img**: contains logo and sign images used for visual verification
- Various **python files** used as executiong scripts and libs

Other infos can be found in repository's README file [3].

4 Approaches tested

After cleaning the repo, set some appropriate name for the variables and changed some parameters in the script, we used all the scripts to test if watermarking was done.
In order to prove the results, we choosed to implement the following results from the paper:

- **Basic Watermarking**: train the original DnCNN model unwatermarked to imprint the watermark on it; also train the Auxiliary Visualizer to show results

- **Fidelity test**: compare the performance (in terms of PSNR) of the unwatermarked model with the watermarked one to check that watermarked model still performs well in his job (denoising as DnCNN)
- **Uniqueness test**: from the watermarked DnCNN, test with different trigger/verification key pairs to see if it changes behaviour
- **Robustness under Fine Tuning**: from the watermarked DnCNN, execute a fine-tuning (last layers) and try to check the watermark presence
- **Robustness under pruning**: from the watermarked DnCNN, execute pruning (at k%) and try to check the watermark presence

4.1 Simple Watermarking view

The basic DnCNN trained model can be found in **DnCNN_weight folder** and can be reloaded and trained.
After generating the key pair (which we will call *real* trigger and verification images) by our key generator (GeneratorTriggerVerificationImg class), we used them along with the dataset images in order to train the model for 8 epochs (fixed number from [6]). The resulting model (with saved checkpoints) is then saved in **overwriting** folder.
From here we will refer to **last epoch 8** as the **watermarked model** (WM). This imply that, for a deployed application, it acts as the network to be deployed.
When giving as input to the WM model the trigger image, it is possible to check watermark presence using the formula in (2) whit $\eta = 6,07 * 10^{-3}$. The script **ExecuteVerification** does this check and returns as values the distance and a boolean for watermark presence or absence.
After training the Auxiliary Visualizer (script AuxVisualizer_train) it is possible to visually see the watermark (Mr. Vision sign, fig. 4). This will act as the base output for our watermark and will be used in next tests as baseline for visual comparison.
In figure 5 we can see the watermark output from the Auxiliary Visualizer model using all checkpoints, saved per epoch during the train, in overwriting folder. As expected **8 epochs are enough** to embed the WM.

Fig. 5: Watermark output per epochs during train

4.2 Fidelity analysis

In order to keep fidelity principle, we compared the **performances** of the unwatermarked model with the watermarked one on **BSD68** and **Set12** test datasets (available in the repository provided by the authors). Dataset Set12 corresponds to what is called Img12 in the paper [6].
The **PSNR** was used as a comparison metric. For each dataset, each image was given as input to both models, and PSNR was calculated on each of them. Then the average of all PSNRs of each dataset was calculated. A higher λ value means a higher strength of the watermark, but a lower PSNR due to some degradation in model performance. So we are searching for the highest λ that minimize the PSNR difference. Results are visible in table 1. For the watermarked model we used $\lambda = \mathbf{10^{-3}}$ (unwatermarked model corresponds to $\lambda = 0$).

Fidelity test		
models (λ)	**BSD68**	**Set12**
Watermarked (10^{-2})	28.68	29.81
Watermarked ($\mathbf{10^{-3}}$)	29.02	30.08
Watermarked (10^{-4})	29.13	30.28
Unwatermarked (0)	29.14	30.32

Table 1: Fidelity test: Average PSNR(dB)of unwatermarked model (basic DnCNN model) and watermarked model

We confirmed that an embedding strength of $\lambda = 10^{-3}$ is enough to embed watermark and not degrade substantially the unwatermarked model performance (according to PSNR results); but results are lower concerning those in [6]. We can see in table 1 that an embedding strength of $\lambda = 10^{-3}$ degrades by 0.12 the PSNR, but using $\lambda = 10^{-}2$ degrades by 0.39 and it is a substantial degradation (paper shows a degradation of 0.06 instead).

4.3 Uniqueness analysis: trigger key in unwatermarked and watermarked model

When giving as input the real trigger image to the unwatermarked DnCNN model it is straightforward that watermark will not be recognized because it wasn't encoded. When using the watermarked model we can recognize it perfectly. Auxiliary visualizer output can be checked in figure 6. Table 2 shows how watermarked psnr value is similar to the original model one and only degrades by 0.15. As for the watermark distance the original model has a value of $7 * 10^{-3}$, bigger than $6.07 * 10^{-3}$ whereas watermarked model is substantially lower ($2.75 * 10^{-3}$).

Unwatermarked vs Watermarked model			
	PSNR	**distance**	**watermarked**
Unwatermarked	29.99	0.00741	false
Watermarked	29.84	0.00275	true

Table 2: Comparison of results between unwatermarked model (basic DnCNN model) and watermarked model

Fig. 6: Output of Auxiliary Visualizer, using the basic (unwatermarked) DnCNN model (left) and the watermarked model (right)

4.4 Uniqueness analysis: multiple keys

For this principle we generated **P-pairs** of triggers (**K**) and verification (**S**) (using our random key generator). After that, the generated trigger images were given as input to the watermarked model separately (already trained with the *real* trigger and verification image), obtaining the corresponding optimal $\mathbf{S}^*$ verification images. Each one of them was subsequently **compared** with the corresponding verification image S using the formula in (2) in order to verify if they encoded the watermark information. A distance greater than *eta* indicates that there is no watermark. Given all the watermarking distances generated by the P pairs, the minimum of all distances was then calculated as a result.
For this experiment we used $P = 200,400,600,800,1000$ pairs of keys. In all the generated keys there is no watermark and the **minimum watermark distance remains stable** as P increases (results in table 3).
Uniqueness is therefore respected since watermarked model does not encode other watermark information.
Figure 7 shows how, by inserting a trigger image (not the real used to train the DnCNN model) in the watermarked model, the visual output does not show the sign MR. Vision Watermarked (fig. 4) and it is completely unrecognizable.

Fig. 7: Example of output using false trigger image

Uniqueness test: watermark distances					
pairs of keys	P=200	P=400	P=600	P=800	P=1000
min watermark distances	0.0095	0.0094	0.0094	0.0094	0.0094
for all keys: watermark presence	False	False	False	False	False

Table 3: Uniqueness results - Minimal distances, Computed on Marked Model

4.5 Robustness analysis: Fine Tuning

In order to perform the fine-tuning we used all layers from watermarked model. Only the **last layer** (conv layer of block 17) has been retrained. As train set we started using original train data (on which original model was trained) and secondarily the Texture images from **KTH_TIPS** dataset (to achieve similar results, we selected 400 images from it, same size as original dataset).
In table 4 we can see results according to epochs 10,25,50,75,100. For all of them, the **watermark was successfully detected**, and the PSNR out of 100 epochs was degrade for less than 0.2 db. Comparing the two dataset resulted in:

- Mean PSNR of **29.46** when fine-tuning with **original** dataset
- Mean PSNR of **29.90** when fine-tuning with **Texture** dataset

This is a normal result given that model started changing slightly its behaviour on different data. But watermark is still visible.
As a visual comparison, by executing the Auxiliary Visualizer model on outputs by epoch we can see the results in figure 8. Therefore the **model is robust to fine-tuning attacks**. This is due to the choice of trigger images that are unlikely to have contents overlapped with the often-used training data.

Fine Tuning watermark distances				
dataset	epochs	**PSNR**	**distance**	**watermarked**
Original	10	29.53	0.00258	true
	25	29.52	0.00259	true
	52	29.49	0.00260	true
	75	29.41	0.00263	true
	100	29.36	0.00267	true
KTH-TIPS	10	29.94	0.00264	true
	25	29.89	0.00267	true
	50	29.88	0.00279	true
	75	29.88	0.00271	true
	100	29.93	0.00276	true

Table 4: Fine Tuning results

Fig. 8: Fine Tuning results on epochs 0,5,10,25,50,75,100

4.6 Robustness analysis: pruning

Pruning was easily done following pruning methods from [8] on TensorFlow (tf) Variables by creating masks and assigning operations to tensors and running one session.
After reloading the model, a simple call to prune passing TensorFlow interesting variables and the percentage of pruning did the trick. We tried to prune **all layers** but this resulted in immediately **losing the watermark**. Paper talks about **pruning lowest k% neurons** and then citing paper [4] where the compression process is done by **Pruning, Quantization and Huffman Encoding**. Applying pruning on all layers reduced too much the PSNR and lost the watermark immediately at $k = 0.05\%$. So instead of applying all the Compression method with the ambiguity described, **we choose to prune only the conv layers in the DnCNN**.
The watermarking sign from the Auxiliary Visualizer can be seen in Figure 9. It is visible that the watermark is kept very well in the first 8 figures and immediately changes after another step suggesting that there is an almost rigid threshold for pruning with keeping watermark. And an attacker would like to have less parameters to compute in order to speed up the forward step but keeping high accuracy (our case PSNR).
Numerical values can be seen in Table 5 suggesting that the threshold might be at $k = 0.55\%$. This **results differs** from the paper's $k = 0.35\%$ but we did prune only the conv layers, so it is a good result. By pruning all layers, the watermark would be lost after $k = 0.03\%$, so this would suggest that paper authors did a different type of pruning (they did not specify if they doing unit or weight pruning or if they doing full compression as in [5] and did not provide any code on TensorFlow 1.x).
One interesting result is that PSNR rapidly drops after $k = 0.35\%$ so even if watermark would be recognized, the DNN would have degraded processing performances so we doubt that eventual attackers would go further.
With state-of-the-art applications, pruning can be rapidly done on Keras models by using **TensorFlow-model-optimization** package and calling **prune_low_magnitude** method or other available. We kept their code so being stick with tf 1.x we did a manual pruning by applying a zero mask on the top k inverse absolute values (smallest k% values from a layer).

Fig. 9: Pruning results from k = 0 ... 0,55

Pruning watermark distances (only conv layers)			
pruned % k	**PSNR**	**distance**	**watermarked**
0.05	25.53	0.00291	true
0.10	25.24	0.00290	true
0.15	25.35	0.00285	true
0.20	17.63	0.00291	true
0.25	17.11	0.00292	true
0.30	16.66	0.00300	true
0.35	17.27	0.00302	true
0.40	12.98	0.00299	true
0.45	12.91	0.00289	true
0.50	11.16	0.00343	true
0.55	9.57	0.00544	true
0.60	9.55	0.00608	false

Table 5: Pruning results

5 Conclusions

Watermaking method applied to image processing DNNs for denoising purposes works well. It is possible to mark the model without losing performance in denoising task and without significantly degrading the image. Furthermore, this method is able to guarantee robustness against common model attacks such as fine-tuning and pruning.

In order to mark it, the model needs to be retrained and our code showed how to perform that in TensorFlow 1.x. When using a model trained with that version, to load a checkpoint we need to still use TensorFlow Variables and tensors, but when possible it is **advised to use Keras** and/or keep it on **TensorFlow 2.x** (because of code maintenance, community answers on questions and lots of other tools available).
As for the principles per se, our results shows almost same values from [6] and when they have a slight change, they still manage to prove that paper's results can be achieved. Future test could be done by:

- **compressing** the network for robustness analysis by using strategy in [4] (a sample code can be found in repository [5])
- testing all those results also on **RED** DNN
- testing watermarking on other case studies (e.g. Superresolution)
- use different frameworks like TensorFlow 2.x, Keras or PyTorch

References

[1] D. Tal D. Martin, C. Fowlkes and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. *ICCV 2001*.

[2] Mario Fritz, Eric Hayman, Barbara Caputo, and Jan olof Eklundh. The kth-tips database, 2004.

[3] Fabian Greavu Giovanna Scaramuzzino. Watermark-dncnn. https://github.com/ScaramuzzinoGiovanna/Watermark-DnCNN, 2020.

[4] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.

[5] Wojciech Mo. Deep compression. https://github.com/wojciechmo/deep-compression, 2018.

[6] Y. Quan, H. Teng, Y. Chen, and H. Ji. Watermarking deep neural networks in image processing. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–14, 2020.

[7] Yuhui Quan. Watermark-dncnn. https://github.com/painfulloop/Watermark-DnCNN, 2020.

[8] Gorjan Radevski. Prunning of deep neural networks. https://github.com/gorjanradevski/pruning_deep_nets, 2019.