symmetry

MDPI

Article

Perceptual Hash of Neural Networks

Zhiying Zhu [1,†], Hang Zhou [2,†], Siyuan Xing [1], Zhenxing Qian [1,*], Sheng Li [1] and Xinpeng Zhang [1]

[1] School of Computer Science, Fudan University, Shanghai 200433, China; zyzhu19@fudan.edu.cn (Z.Z.); syxing19@fudan.edu.cn (S.X.); lisheng@fudan.edu.cn (S.L.); zhangxinpeng@fudan.edu.cn (X.Z.)
[2] School of Computing Science, Simon Fraser University, Burnaby, BC V5A 1S6, Canada; zhouhang2991@gmail.com
* Correspondence: zxqian@fudan.edu.cn
† These authors contributed equally to this work.

Abstract: In recent years, advances in deep learning have boosted the practical development, distribution and implementation of deep neural networks (DNNs). The concept of symmetry is often adopted in a deep neural network to construct an efficient network structure tailored for a specific task, such as the classic encoder-decoder structure. Massive DNN models are diverse in category, quantity and open source frameworks for implementation. Therefore, the retrieval of DNN models has become a problem worthy of attention. To this end, we propose a new idea of generating perceptual hashes of DNN models, named HNN-Net (Hash Neural Network), to index similar DNN models by similar hash codes. The proposed HNN-Net is based on neural graph networks consisting of two stages: the graph generator and the graph hashing. In the graph generator stage, the target DNN model is first converted and optimized into a graph. Then, it is assigned with additional information extracted from the execution of the original model. In the graph hashing stage, it learns to construct a compact binary hash code. The constructed hash function can well preserve the features of both the topology structure and the semantics information of a neural network model. Experimental results demonstrate that the proposed scheme is effective to represent a neural network with a short hash code, and it is generalizable and efficient on different models.

Keywords: perceptual hash; DNN; model retrieval; graph hash; HNN-Net

Citation: Zhu, Z.; Zhou, H.; Xing, S.; Qian, Z.; Li, S.; Zhang, X. Perceptual Hash of Neural Networks. *Symmetry* **2022**, *14*, 810. https://doi.org/10.3390/sym14040810

Academic Editor: Jeng-Shyang Pan

Received: 24 March 2022
Accepted: 11 April 2022
Published: 13 April 2022

1. Introduction

Of late, deep learning [1] has attracted the greatest attention in both academia and industry. It is increasingly applied to a variety of fields [2], such as image processing [3,4], natural language processing [5], audio processing [6,7], biometrics [8], etc. The core task is designing deep neural network (DNN) models. Researchers are developing new DNN models and their variants for specific tasks using frameworks such as TensorFlow, PyTorch, MXNet, Keras, Caffe, etc.

Obtaining a preferable pre-trained model requires a large amount of training data, the delicate designing of neural networks and expensive computing resources. It is no doubt true that each pre-trained model consumes a lot of comprehensive costs. However, these models are at risk of being stolen in direct or indirect ways. For example, pirates can directly steal by illegally copying deep neural network model files or use APIs opened by manufacturers to users to achieve indirect stealing by technical means such as knowledge distillation [9]. With the widespread use of cloud computing, DNN models are usually deployed on cloud server platforms. Pirated models may also be deployed, and the platform needs to provide users with discovery services of infringements. Due to the huge number of DNN models, we need technology to discover whether the uploaded model is suspected of piracy; therefore, we propose a new method: perceptual hash for neural networks. The flow chart is shown in Figure 1. We create a hash code for each model. Once

a new model is uploaded, we compare the hash code of the new model and the existing models to retrieve similar models.

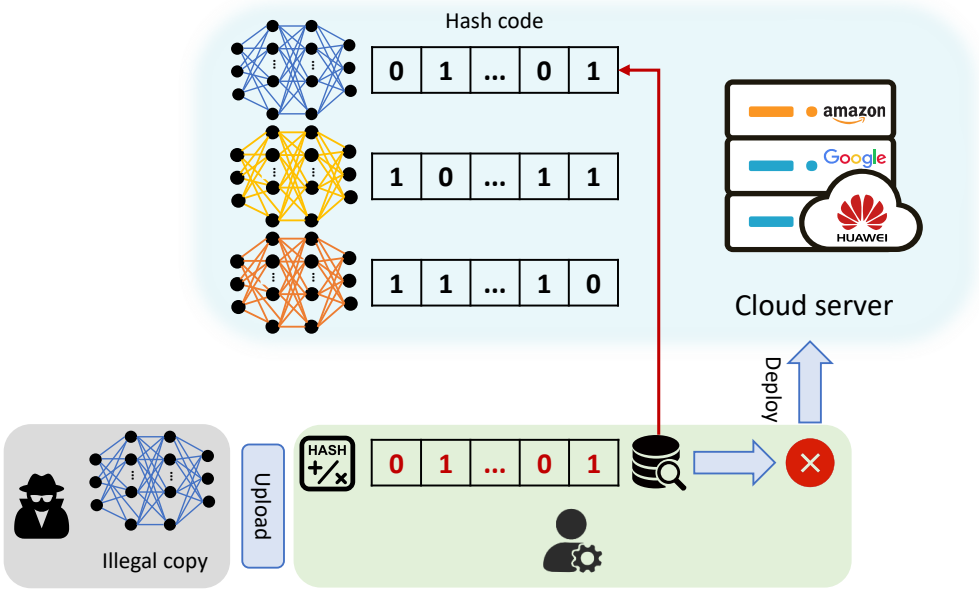


Figure 1. Flowchart of preventing pirated deployment. We create a hash code for each model on the cloud server. When a DNN model is uploaded, the administrator will compare the new hash code with the existing hash codes. The uploaded model will be further checked if a similar hash code is retrieved.

To deal with the DNN retrieval task, we propose HNN-Net, a supervised two-stage scheme based on graph neural network (GNN) [10]. In the first stage, we use a graph generator to convert the DNN model to an undirected graph assigned with additional information we extracted after a series of operations. We use the graph hashing network to generate compact hash codes in the second stage. As the hamming distance computation is fast for the CPU, the proposed DNN model hashing can be applied to a large-scale database for retrieval. The proposed method is evaluated on a DNN model benchmark dataset we collected. The experiment results demonstrate that the proposed method is effective for neural network retrieval. The details of the two stages are shown in Figure 2, and we will further discuss the deep neural network hashing in Section 3.

Our contributions are three-fold, as follows:

- We propose a new idea of generating perceptual hash for neural networks, which can be used in model protection;
- The proposed deep hashing scheme based on neural graph work is capable of all kinds of deep learning frameworks;
- The proposed method is effective that has a good retrieval performance.

The rest of this article is organized as follows. In Section 2, we present the related works of the perceptual hash of neural networks. In Section 3, we define the problem and describe the details of the proposed DNN hash method. We evaluate the capacity of our scheme and set up some ablation experiments in Section 4. Section 5 concludes the article by recapitulating our work.

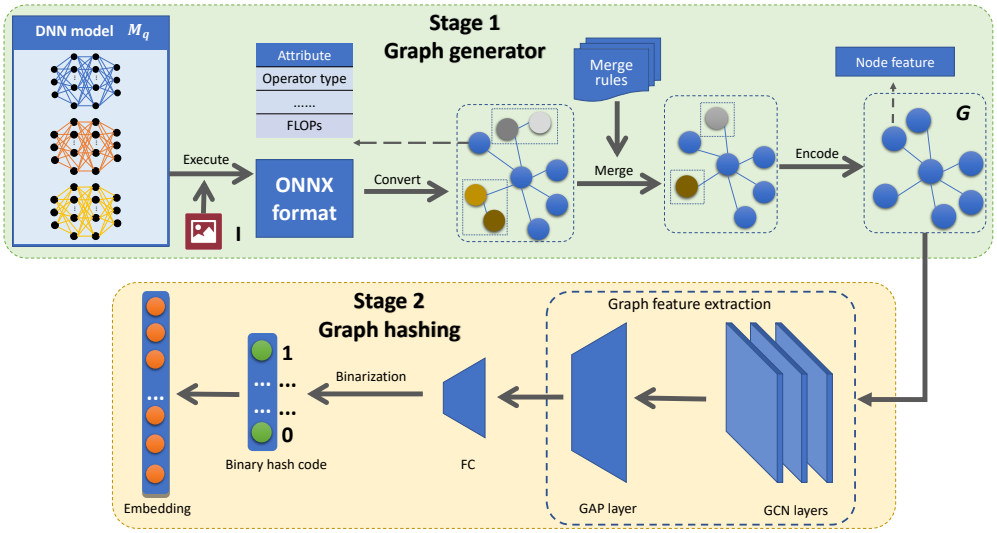


Figure 2. The details of our two-stage scheme. In the first stage, a raw DNN model is converted into a graph with an initial node feature after a series of operations. Afterward, the graph, which is generated by Stage 1, gets its graph-level feature embedding. Then the embedding is used to generate Q-bit hash codes after binarization and predict the classification of the input model.

2. Related Works

2.1. Computational Graph

A neural network model can be regarded as a series of complex computations with trainable parameters organized in a specific structure. One of the tools we use to describe the DNN model is a computational graph [11,12], which is a directed graph used to express a set of structured computations whose nodes and edges represent operators and directions of dataflow, respectively. The operator stands for a calculation function for input, and data flow goes along the graph's edges. When a data flow is passed through a node, it is dealt with by a node operator and generates a new value. Two examples of computational graphs for some simple functions are shown in Figure 3. Note that computational graphs are widely used for back-propagation on neural networks. Apparently, any DNN model can be represented as a computational graph.

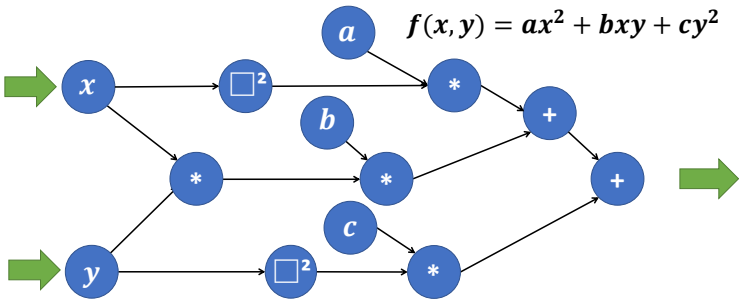


Figure 3. The computational graph for function $f(x,y) = ax^2 + bxy + cy^2$. Each node stands for a specific operator such as addition and multiplication. Each edge stands for the direction of the dataflow. The graph is fed with x and y as input and produce an output as the final result.

2.2. Deep Image Hashing

Traditional cryptographic hashing algorithms have an "avalanche effect", which implies that a minute difference between two input data will cause a significant difference in the output hash codes. On the contrary, perceptual hashing generates similar hash codes for two similar images and diverse hash codes for different images, which has blazed a trail for researchers to measure image similarity.

Deep image hashing makes use of deep learning to generate perceptual hash codes for images. CNNH [13] is an early work to learn image representation and image hashing.

To unify representation learning and hashing learning, DPSH [14] and HashNet [15] both propose end-to-end hashing learning approaches based on pair-wise loss. Moreover, DNNH [16] utilizes the triplet ranking loss to guide the learning for the hash function. In addition, SDH [17] makes use of an objective function to minimize the intra-class variations and maximize the inter-class variations.

SSDH [18] proposes an assumption that the semantic labels are governed by several latent attributes with each attribute on or off, and classification relies on these attributes. Based on this assumption, SSDH employs a softmax classifier to guide deep hashing learning to better exploit the semantic information of images. This idea inspired us to adopt a classification layer to make better use of the semantic information of DNN models and unifies DNN model classification and retrieval in a single learning model, which is trained with pair-wise loss.

2.3. Graph Hashing

A graph (Graph) is a data structure modeling of a set of objects (nodes) and relationships (edges) between objects, and is widely used in various fields such as knowledge graphs [19], natural sciences [20] and many other fields [21,22]. The graph neural network (GNN) [10] is a widely used technique to process graph structure data based on deep neural networks. Graph convolution network (GCN) [22] defines a specific mechanism about how every node gathers information from its neighboring nodes and learns to generate its representation. Compared with early hand-crafted features, it performs much better in extracting effective features from graphs and representing nodes, edges or graphs in low-dimensional vectors. Additionally, attention mechanisms such as [23–27] have been proposed for graph-level or node-level tasks.

To solve the problem of graph similarity search, many GNN-based methods [22,26,28–30] have been proposed to generate a graph-level embedding to minimize the distance between two graphs [31]. However, they are not effective enough to search large-scale databases in real-time.

To achieve a fast graph similarity search, GHashing [31] is the latest attempt that uses GNN to generate binary hash codes and graph-level embedding for fast graph similarity search. It uses graph attention pooling (GAP) [26] as its attention mechanism. However, GHashing can not convert DNN models into hash code directly. This is not suitable for practical application. In our task, it is taken as a basic foundation of our network for neural network hashing. Different from Ghashing, in our network, the DNN models can be directly used as input to get hash codes.

3. Deep Neural Network Hashing

3.1. Problem Definition

We denote $\{M_i\}_{i=1}^{N}$ as a DNN model database associated with its classification labels, and M_q as an arbitrary irrelevant DNN model to query. DNN model retrieval is the task of retrieving the most similar model from the database with the query M_q. A DNN model refers to a concrete executable DNN model implemented by a specific type of deep learning framework. Take PyTorch [32] as an instance. A Python class inherited from "torch.nn.Module" with its parameters loaded consists of a complete DNN model.

As we introducted in Section 1, the model can be converted into a graph. In this procedure, we trace the data flow until the model terminates to get a trace with a full record of operations that occurred in the data flow. From the trace, we can obtain rich information in addition to the topology structure information of the graph, such as the number of FLOPs (floating number operations of a layer) and the number of trainable parameters for each layer in various ways.

To handle the task, our proposed scheme learns a hash function that maps a DNN model M_q to a Q-bit binary hash code $\mathbf{h}$.

Given M_q as the query, we computed hamming distances between the hash code for $\mathbf{h}$ and those for models in the database. Hamming distance measures the similarity between two binary hash codes, calculated as:

$$\text{HAMMING}(\mathbf{h}_1, \mathbf{h}_2) = \|\mathbf{h}_1 \oplus \mathbf{h}_2\|_1, \tag{1}$$

where $\oplus$ is the exclusive OR operation and $\mathbf{h}_1$ and $\mathbf{h}_2$ are two hash codes.

The details of the two-stage scheme are shown in Figure 2, and the overall training procedure is shown in Figure 4. We first converted a DNN model M_q into an optimized computational graph which is assigned with additional information captured from the execution of the model, and then fed it into a GNN-based neural network, which consisted of graph feature extraction layers and fully-connected layers. The output was a Q-bit binary hash code.

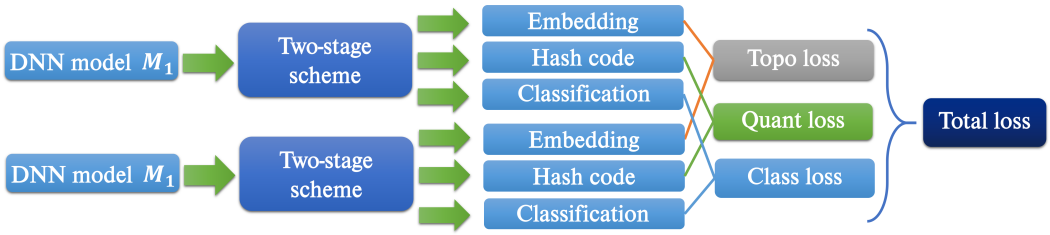


Figure 4. Flowchart of our proposed scheme in the training stage, consisting of 3 losses in a pair-wise manner. $\mathcal{L}_{topo}$ and $\mathcal{L}_{class}$ are, respectively, used for preserving topology structure similarity and semantic similarity between models. $\mathcal{L}_{quant}$ reduces quantization loss caused by the conversion from continuous activation values to discrete binary hash code.

3.2. Stage 1: Graph Generator

A DNN model M_q was first converted to an undirected acyclic graph $(\mathcal{V}, \mathcal{E}, \mathcal{F})$ with a trigger image $\mathbf{I}$, where $\mathcal{V}$ is the set of nodes, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges, and $\mathcal{F}$ is a set of functions that map a vertex to multiple attributes. The computational graph would be too large to handle if we defined basic operators like addition or multiplication as nodes. Instead, we merged the diverse operators into a single graph node, which was represented as a network layer in deep learning.

To make our scheme compatible with all kinds of existing deep learning frameworks, these merges followed a standard open-source format called ONNX (Open Neural Network Exchange) [33].

3.2.1. ONNX Operation

We denoted the function that maps a DNN model to the computational graph whose operators and data types obey the ONNX [33] standard as f_{onnx}. ONNX provides the definition of its built-in operators and standard data types, where each computation dataflow graph is structured as a list of nodes that form a graph [33]. The set of built-in operators is portable across frameworks, and every framework supporting ONNX provides implementations of these operators on the data types. Hence, our scheme is compatible with most existing frameworks, including TensorFlow, PyTorch, Mxnet, Keras, Caffe, etc. [33].

3.2.2. Merge Operation

Many DNN structures contain duplicate sub-structure blocks that perform a similar role across different neural networks. Merging specific layers into one block, representing a node in the graph, can improve the hashing accuracy. Therefore, we applied hard-coded merge rules, i.e., Conv-ReLU, Conv-BatchNorm, Conv-BatchNorm-ReLU, and Conv-Conv-BatchNorm-ReLU, to the previous graph, and denote the merge operation by f_{merg}. After ONNX and merge operations, the model M_q is converted to a graph by:

$$G \leftarrow f_{merg}(f_{onnx}(M_q(\mathbf{I}))). \tag{2}$$

3.2.3. Node Feature Embedding

To make full use of model characters for model hashing, we first extracted the number of FLOPs and parameters from the trace of the data flow. Each node $v_i \in \mathcal{V}$ from the established graph G has 3 additional information mapping functions $\mathcal{F} = \{f_1, f_2, f_3\}$ which, respectively, map a vertex to its operator type, the number of FLOPs and the number of parameters on the corresponding layer. In the last encoding step, for any node v_i, its f_2, f_3 returns continuous values while f_1 returns discrete value. For continuous values $f_2(v_i), f_3(v_i)$, we designed an encoder function $f_{enc}(x) = \max([\log_{10}(x)] + 1, 10)$ to map the input to discrete values, where $[\log_{10}(x)] + 1$ is the number of digits in the the integer part of x, e.g., 3 for 123.45 and 2 for 97.76. To normalize the feature dimensions, we used the max function to ensure that they drop within $\{0, 1, 2, ..., 10\}$. The three attributes were thus all discrete values. Then we encoded them into three one-hot vectors by f_{oneh} and concatenated them into a unified one. The initial node representation $\mathbf{u}_i$ for vertex $v_i \in \mathcal{V}$ is

$$\mathbf{u}_i = [f_{oneh}(f_1(v_i)), f_{oneh}(f_{enc}(f_2(v_i))), f_{oneh}(f_{enc}(f_3(v_i)))], \tag{3}$$

where $[x, y]$ means concatenating x, y along the feature dimension.

3.3. Stage 2: Graph Hashing

The graph hashing network $\mathcal{H}$ can be divided into three sub-networks, focusing on how to extract graph-level features and how to generate a compact Q-bit binary code that preserves both the topology structure similarity and the semantic similarity among models. We used pair-wise loss between two different DNN models in each iteration for training.

The first sub-network, which is called graph feature extraction, contained three graph convolution network layers (GCN) [22] and one graph attention pooling layer (GAP) [26], and output a graph-level feature. We denoted $\mathbf{u}_i$ as the representation of node v_i, and $\mathbf{u}'_i$ as the next representation of node v_i processed by a GCN layer.

GCN [22] is a neighbor aggregation method that defines how a node aggregates embeddings of all its neighbors and learns to generate an output as its own subsequent embedding, as defined by

$$\mathbf{u}'_i = \mathrm{GCN}(\mathbf{u}_i). \tag{4}$$

The final output of GCNs $\mathbf{U} \in \mathbb{R}^{N \times D}$ is the feature matrix whose i-th row is the feature vector $\mathbf{u}'_i$. GAP [26] defines how we produced a graph-level embedding from a graph whose every node has its own node representation, as defined by

$$\mathbf{U}' = \mathrm{GAP}(\mathbf{U}). \tag{5}$$

Then, the fully-connected layers contain 5 layers denoted by $\mathrm{FC}_1 \sim \mathrm{FC}_5$. FC_1 receives the output features from GAP as its input and $\mathrm{FC}_1 \sim \mathrm{FC}_3$ generates graph features. The binarization layers generate a Q-bit binary code based on the output of FC_5. Finally, it outputs an embedding in continuous values and a discrete binary hash code.

3.4. Objective Loss Function

The objective loss function between two models M_1 and M_2 consists of three parts: topology loss, classification loss and quantization loss, and formally:

$$\mathcal{L}(M_1, M_2) = \alpha \mathcal{L}_{topo}(M_1, M_2) + \beta \mathcal{L}_{class}(M_1, M_2) + \gamma \mathcal{L}_{quant}(M_1, M_2), \tag{6}$$

where α, β and γ are hyper-parameters.

$\mathcal{L}_{topo}$ encourages our model to preserve more topology structure similarity and a part of semantic similarity between the two models, as defined by

$$\mathcal{L}_{topo}(M_1, M_2) = (||\mathrm{FC}_3(M_1) - \mathrm{FC}_3(M_2)||_2^2 - \min(\mathrm{GED}(M_1, M_2), R))^2, \tag{7}$$

where GED [34] is the graph edit distance to measure topology structure similarity between two graphs. Since it makes no sense for our model when it becomes too large, we used R as an upper bound. Inspired by GHashing [31] that learning from an embedding function described above is better than learning a hash function directly, we used the output of FC_3 for constraint.

$\mathcal{L}_{class}$ is a classification loss that encourages neural networks to be discriminative among different model types, as defined by

$$\mathcal{L}_{class}(M_1, M_2) = -(t_1 \ln \mathcal{H}(M_1) + t_2 \ln \mathcal{H}(M_2)), \tag{8}$$

where the output of $\mathcal{H}$ is the classification output, and t_1, t_2 are the ground-truth model labels of M_1 and M_2, respectively.

$\mathcal{L}_{quant}$ encourages binary hash code to be more compact and carry more information in each bit and is defined as

$$\mathcal{L}_{quant}(M_1, M_2) = -(\|\mathrm{FC}_5(M_1) - \frac{1}{2}\mathbf{1}\|_1 + \|\mathrm{FC}_5(M_2) - \frac{1}{2}\mathbf{1}\|_1), \tag{9}$$

where $\mathbf{1}$ is a Q-dim all-ones vector.

Quantization loss is a common problem for hashing based on deep learning. There is an information loss in the conversion from the continuous activation value to discrete binary hash code by the binarization. It is a common technique to reduce this loss by encouraging each unit of the activation values to be close to either 0 or 1.

4. Experiments

4.1. Implementation Details

4.1.1. Network Architecture

The output dimensions of $GCN_1 \sim GCN_3$ were 256, 128 and 64, respectively. The input dimension of FC_1 was 448. The output dimensions of $FC_1 \sim FC_3$ were 348, 256 and 256, respectively. The output dimension of FC_4 was 128. The number of binary code $Q = \{16, 32, 64\}$. The trigger image $\mathbf{I}$ was a $[3, 256, 256, 256]$-shaped zero tensor as default.

4.1.2. Training Parameters

The hyperparameters were defined as follows: $\alpha = 10, \beta = 10, \gamma = 0.2$. The implementation was based on TensorFlow. For the optimization, we trained the network for 200 epochs using the Adam optimizer with a minibatch size of 10, and the learning rate is 0.001. The whole training process took about 16 h on the NVIDIA RTX 1080 Ti GPU and Intel® Xeon® Silver 4210 CPU @ 2.20 GHz. Our dataset was generated with pre-trained neural network models built in PyTorch 1.8.1.

4.1.3. Dataset

Our new DNN model dataset consists of 22 models belonging to 10 categories extracted from the PyTorch package "torchvision.models", as shown in Table 1. We randomly delete 10% nodes and their related edges in computational graphs of models by 10 times for data augmentation. Finally, we pick the first 80% of data as the training dataset and the remaining 20% as the test dataset.

Table 1. Model list of our dataset includes 10 categories and 22 models.

#	Category	Model Name in PyTorch
1	VGG	vgg 11, 13, 16
2	Resnet	resnet 18, 34, 101
3	mnaset	mnasnet 0_75, 1_0, 1_3
4	DenseNet	densenet 121, 161, 201
5	SqueezeNet	squeezenet 1_0, 1_1, v2_x0_5
6	ShuffleNet	shufflenet v2_x0_5, v2_x1_5, v2_x2_0
7	AlexNet	alexnet
8	Inception	inception_v3
9	MobileNet	mobilenet_v2
10	GoogLeNet	GoogLeNet

4.1.4. Evaluation Metrics

We conducted traditional retrieval experiments and used common performance metrics in the retrieval task. We picked top-k models that have the smallest hamming distance with the query as the retrieval result. Evaluation metrics included precision, recall and F_1 score.

4.2. Comparison with State-of-the-Art Methods

4.2.1. Methods in Comparison

We compared our method with GHashing [31] and a simplified version of our method denoted as HNN-Net w/o $\mathcal{L}_{class}$. The code of GHashing [31] is open-source without a license. Raw data stands for data that were not processed with hard-coded merge rules, and merged data stands for data that were processed with merge rules.

As introduced in Section 3.2, "merge" is a technique adopted by our Graph Generator to optimize the computations we generated. For example, if our merge rules contained a single rule "CONV - RELU" and we had a graph which only contains 2 two nodes (0:CONV,1:RELU), 1 edge "CONV -> RELU". After "merge", this graph would be converted to a graph with only 1 node (0:CONV-RELU) and 0 edges. "Raw data" means that all graphs were simply converted from ONNX format and would not be processed by the "merge" operation. If we apply "merge" to "Raw Data", we get "Merged Data".

4.2.2. Results

Figure 5 shows evaluation curves evaluated with varying numbers of hash codes Q. Figure 5a,b are evaluated on raw data and merged data, respectively. We adopt $\alpha = 10$, $\beta = 10$, and $\gamma = 0.2$ for HNN-Net. HNN-Net performs much better than HNN-Net without $\mathcal{L}_{class}$ and GHashing when Q is larger than 16 regardless of which data we use. Table 2 provides comparison results among varying methods, and the best ones are in bold. From the table, we can conclude that our method outperforms baselines in most cases. The improvement comes from the better information extraction from the deep neural networks according to the execution of the DNN model and the classification loss, which can preserve the semantic similarity between DNN models. Additionally, merge operation makes better use of transcendental knowledge in deep learning and helps us learn a better hashing function.

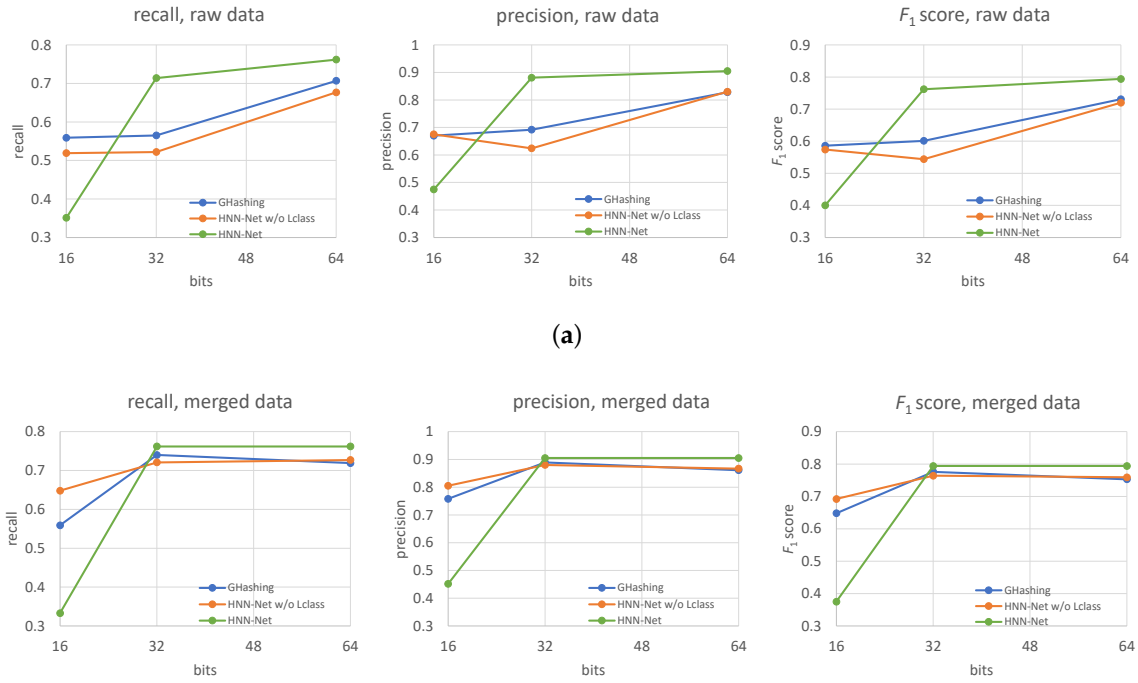


Figure 5. Comparative evaluation metrics curves of different methods on two datasets. (**a**) is evaluated on the raw data with respect to varying Q. (**b**) is evaluated on the merged data with respect to varying Q.

In Figure 5b, 32-bit HNN-Net slightly outperforms 64-bit HNN-Net. 64 bits make a single hash code carry much more information than 32 bits. However, if a single hash code can carry too much information from the training dataset, overfitting can result, leading to a decrease in performance in the test dataset.

Table 2. The comparison of recall, precision and F_1 score among GHashing [31], HNN-Net without $\mathcal{L}_{class}$ and HNN-Net in different hash code bits evaluated on the raw data and the merged data.

Methods	Metrics	Raw Data			Merged Data		
		16 Bits	32 Bits	64 Bits	16 Bits	32 Bits	64 Bits
GHashing [31]	Recall	**0.559**	0.565	0.707	0.559	0.74	0.719
	Precision	0.67	0.692	0.828	0.758	0.889	0.861
	F_1 score	**0.586**	0.601	0.731	0.648	0.776	0.753
HNN-Net w/o $\mathcal{L}_{class}$	Recall	0.519	0.522	0.677	**0.648**	0.721	0.727
	Precision	**0.675**	0.624	0.83	**0.805**	0.88	0.867
	F_1 score	0.574	0.544	0.72	**0.692**	0.764	0.759
HNN-Net	Recall	0.351	**0.714**	**0.762**	0.333	**0.762**	**0.762**
	Precision	0.475	**0.881**	**0.905**	0.452	**0.905**	**0.905**
	F_1 score	0.4	**0.762**	**0.794**	0.375	**0.794**	**0.794**

4.3. Ablation Analysis

4.3.1. The Role of Merge Operation

Figure 6 shows the results of using merged data or not. Figure 6a,b demonstrate clearly that using merged data brings huge performance improvements, while for our method shown in Figure 6c, the improvement brought by a merge operation is much smaller than others.

The essence of the merge operation is to utilize transcendental knowledge of neural networks, namely their semantic information. Thus, more improvements could be brought by a merge operation to an approach. The approach effectively extracts less semantic information. On the other hand, HNN-Net introduces semantic loss to its loss function, while GHashing and HNN-Net without $\mathcal{L}_{class}$ do not use it and aim to extract more semantic information from data and generate better hash codes. Accordingly, experiment results in Figure 6c imply that the merge operation brings the least improvement to HNN-Net compared with the other two. It proves semantic loss in HNN-Net, and merge operations extract the same kind of information inside the data as expected. In other words, both of them extract semantic information from data effectively.

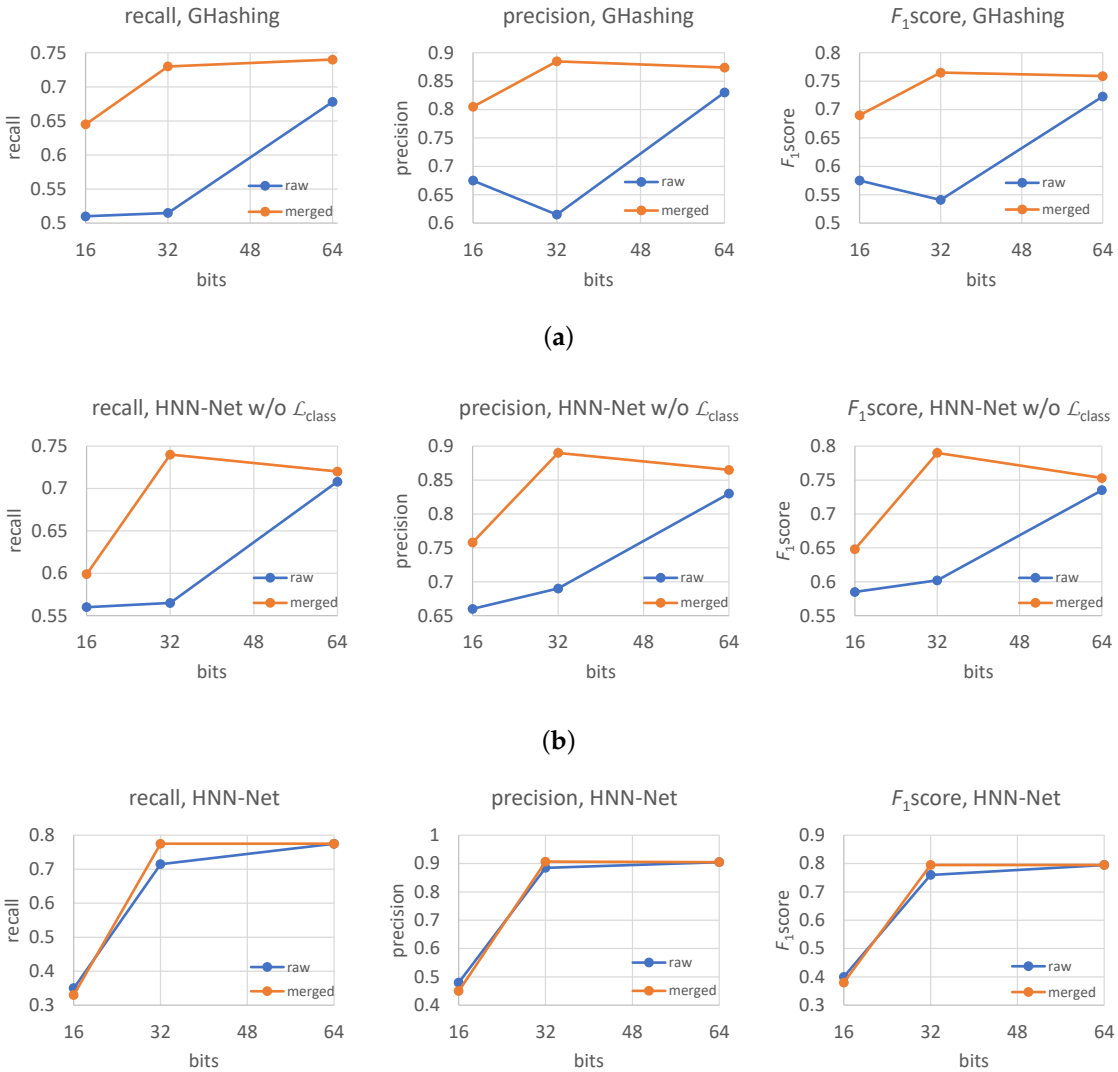
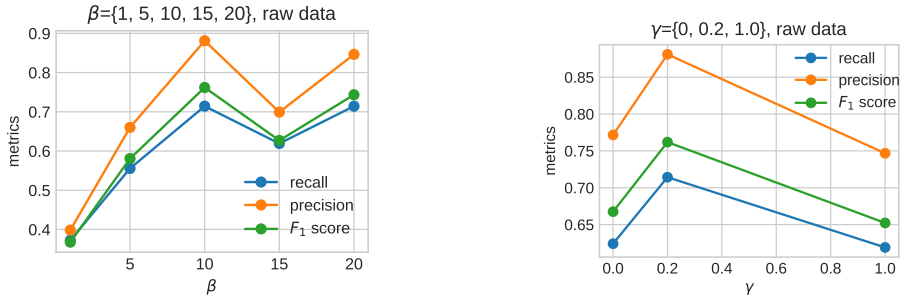


Figure 6. Comparative evaluation curves among different hashing algorithms. (**a**) GHashing [31], (**b**) HNN-Net w/o $\mathcal{L}_{class}$, (**c**) HNN-Net.

4.3.2. Effect of the Classification Loss $\mathcal{L}_{class}$ and the Quantization Loss $\mathcal{L}_{quant}$

Figure 7 shows the performance of HNN-Net evaluated on raw data with varying β when $\alpha = 10, \gamma = 0.2, Q = 32$. As we can see, the three criteria all perform the best when $\beta = 10$. A natural explanation is that a much larger β prevents the model from paying sufficient attention to the topology structure similarity among models, and a much smaller β restrains the model from preserving semantic similarities. Figure 7b is evaluated on raw

data with varying γ when $\alpha = 10, \beta = 10$ and the optimal weight $\gamma = 0.2$. Additionally, an appropriate $\mathcal{L}_{quant}$ can reduce the information loss caused by the conversion from the feature embedding in continuous values to discrete binary hash code.



(**a**) (**b**)

Figure 7. Comparative evaluation curves of HNN-Net on varying β and γ. (**a**) Evaluation on raw data with respect to varying β. (**b**) Evaluation on raw data with respect to varying γ.

5. Conclusions

We issue the new problem of DNN model retrieval in the face of deep learning security threats. We propose a two-stage hashing scheme based on GNNs, and it is compatible with models implemented by most existing deep learning frameworks. HNN-Net generates hash codes that preserve both the topology structure and the semantic similarity of models, and then learns a classifier for discriminating DNN models. Results verify that our scheme is effective and performs much better than others. HNN-Net makes full use of the shallow information in DNN, such as topology information, network node information, and network operation intermediate result information. However, the use of deep information in DNN, such as the functional information of the network itself, is very limited. In the future, we will combine the shallow and deep information of the DNN model to generate better quality perceptual hash codes.

Author Contributions: Z.Z.: Conceptualization, Methodology, Writing—original draft. H.Z.: Writing—review & editing. S.X.: Software. Z.Q.: Visualization, Investigation. S.L.: Data curation. X.Z.: Supervision. All authors have read and agreed to the published version of the manuscript.

Funding: National Natural Science Foundation of China: 62072114; National Natural Science Foundation of China: U20A20178; National Natural Science Foundation of China: U20B2051; National Natural Science Foundation of China: U1936214.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: Thanks for the support from the National Natural Science Foundation of China under Grant U20B2051, Grant U20A20178, Grant 62072114 and Grant U1936214.

Conflicts of Interest: The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

1. LeCun, Y.; Bengio, Y.; Hinton, G. Deep learning. *Nature* **2015**, *521*, 436–444. [CrossRef] [PubMed]
2. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press: Cambridge, MA, USA, 2016.
3. Voulodimos, A.; Doulamis, N.; Doulamis, A.; Protopapadakis, E. Deep learning for computer vision: A brief review. *Comput. Intell. Neurosci.* **2018**, *2018*, 7068349. [CrossRef] [PubMed]
4. Hemanth, D.J.; Estrela, V.V. *Deep Learning for Image Processing Applications*; IOS Press: Amsterdam, The Netherlands, 2017; Volume 31.

5. Young, T.; Hazarika, D.; Poria, S.; Cambria, E. Recent trends in deep learning based natural language processing. *IEEE Comput. Intell. Mag.* **2018**, *13*, 55–75. [CrossRef]
6. Purwins, H.; Li, B.; Virtanen, T.; Schlüter, J.; Chang, S.Y.; Sainath, T. Deep learning for audio signal processing. *IEEE J. Sel. Top. Signal Process.* **2019**, *13*, 206–219. [CrossRef]
7. Lee, H.; Pham, P.; Largman, Y.; Ng, A. Unsupervised feature learning for audio classification using convolutional deep belief networks. *Adv. Neural Inf. Process. Syst.* **2009**, *22*, 1096–1104.
8. Sundararajan, K.; Woodard, D.L. Deep learning for biometrics: A survey. *ACM Comput. Surv.* **2018**, *51*, 1–34. [CrossRef]
9. Hinton, G.; Vinyals, O.; Dean, J. Distilling the Knowledge in a Neural Network. *Comput. Sci.* **2015**, *14*, 38–39.
10. Zhou, J.; Cui, G.; Zhang, Z.; Yang, C.; Liu, Z.; Wang, L.; Li, C.; Sun, M. Graph neural networks: A review of methods and applications. *arXiv* **2018**, arXiv:1812.08434.
11. Kantorovich, L.V. On a mathematical symbolism convenient for performing machine calculations. *Dokl. Akad. Nauk SSSR* **1957**, *113*, 738–741.
12. Bauer, F.L. Computational graphs and rounding error. *SIAM J. Numer. Anal.* **1974**, *11*, 87–96. [CrossRef]
13. Xia, R.; Pan, Y.; Lai, H.; Liu, C.; Yan, S. Supervised Hashing for Image Retrieval via Image Representation Learning. In Proceedings of the AAAI Conference on Artificial Intelligence, Québec City, QC, Canada, 27–31 July 2014; Volume 28.
14. Li, W.J.; Wang, S.; Kang, W.C. Feature learning based deep supervised hashing with pairwise labels. *arXiv* **2015**, arXiv:1511.03855.
15. Cao, Z.; Long, M.; Wang, J.; Yu, P.S. Hashnet: Deep learning to hash by continuation. In Proceedings of the IEEE International Conference on Computer Vision, Venice, Italy, 22–29 October 2017; pp. 5608–5617.
16. Lai, H.; Pan, Y.; Liu, Y.; Yan, S. Simultaneous feature learning and hash coding with deep neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, 7–12 June 2015; pp. 3270–3278.
17. Erin Liong, V.; Lu, J.; Wang, G.; Moulin, P.; Zhou, J. Deep hashing for compact binary codes learning. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, 7–12 June 2015; pp. 2475–2483.
18. Yang, H.F.; Lin, K.; Chen, C.S. Supervised learning of semantics-preserving hash via deep convolutional neural networks. *IEEE Trans. Pattern Anal. Mach. Intell.* **2017**, *40*, 437–451. [CrossRef] [PubMed]
19. Hamaguchi, T.; Oiwa, H.; Shimbo, M.; Matsumoto, Y. Knowledge Transfer for Out-of-Knowledge-Base Entities: A Graph Neural Network Approach. *arXiv* **2017**, arXiv:1706.05674.
20. Battaglia, P.W.; Pascanu, R.; Lai, M.; Rezende, D.; Kavukcuoglu, K. *Interaction Networks for Learning about Objects, Relations and Physics*; Curran Associates Inc.: Red Hook, NY, USA, 2016.
21. Hamilton, W.L.; Ying, R.; Leskovec, J. Inductive Representation Learning on Large Graphs. *arXiv* **2017**, arXiv:1706.02216.
22. Kipf, T.N.; Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv* **2016**, arXiv:1609.02907.
23. Lee, J.B.; Rossi, R.; Kong, X. Graph classification using structural attention. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, London, UK, 19–23 August 2018; pp. 1666–1674.
24. Thekumparampil, K.K.; Wang, C.; Oh, S.; Li, L.J. Attention-based graph neural network for semi-supervised learning. *arXiv* **2018**, arXiv:1803.03735.
25. Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Lio, P.; Bengio, Y. Graph attention networks. *arXiv* **2017**, arXiv:1710.10903.
26. Bai, Y.; Ding, H.; Qiao, Y.; Marinovic, A.; Gu, K.; Chen, T.; Sun, Y.; Wang, W. Unsupervised inductive graph-level representation learning via graph-graph proximity. *arXiv* **2019**, arXiv:1904.01098.
27. Li, Y.; Gu, C.; Dullien, T.; Vinyals, O.; Kohli, P. Graph matching networks for learning the similarity of graph structured objects. *arXiv* **2019**, arXiv:1904.12787.
28. Bai, Y.; Ding, H.; Bian, S.; Chen, T.; Sun, Y.; Wang, W. Simgnn: A neural network approach to fast graph similarity computation. In Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining, Melbourne, Australia, 11–15 February 2019; pp. 384–392.
29. Ying, Z.; You, J.; Morris, C.; Ren, X.; Hamilton, W.; Leskovec, J. Hierarchical graph representation learning with differentiable pooling. In Proceedings of the 32nd International Conference on Neural Information Processing Systems, Montreal, QC, Canada, 3–8 December 2018; pp. 4800–4810.
30. Bai, Y.; Ding, H.; Gu, K.; Sun, Y.; Wang, W. Learning-Based Efficient Graph Similarity Computation via Multi-Scale Convolutional Set Matching. In Proceedings of the AAAI, New York, NY, USA, 7–12 February 2020; pp. 3219–3226.
31. Qin, Z.; Bai, Y.; Sun, Y. GHashing: Semantic Graph Hashing for Approximate Similarity Search in Graph Databases. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, Virtual Event, CA, USA, 6–10 July 2020; pp. 2062–2072.
32. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. Pytorch: An imperative style, high-performance deep learning library. In Proceedings of the 33rd International Conference on Neural Information Processing Systems, Vancouver, BC, Canada, 8–14 December 2019; pp. 8026–8037.
33. Onnx: Open Neural Network Exchange. 2019. Available online: https://github.com/onnx/onnx (accessed on 10 April 2022).
34. Gao, X.; Xiao, B.; Tao, D.; Li, X. A survey of graph edit distance. *Pattern Anal. Appl.* **2010**, *13*, 113–129. [CrossRef]