Report of Class Project: Jigsaw Puzzles

Letian Yang
Shanghai Jiao Tong University

1. Introduction

In this class project, the Jigsaw puzzles are introduced as a *self-supervised learning* problem. The problem definition of a Jigsaw puzzle reassembly problem is that: given an image divided into evenly-sized but shuffled pieces, we try to figure out the order of those pieces, or equivalently, restore the images from those pieces.

2. DeepPermNet

In our class *DeepPermNet* [5] is given as a reference. As an end-to-end model, *DeepPermNet* is applicable to most visual permutation learning problems where the permutation might be based on semantic information. In the paper, the permutation of images according to the extent of smiling and narrow eyes (Fig.1) are shown as two examples of visual permutation.

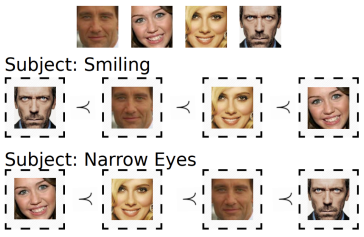

Figure 1. An example of visual permutation

In fact it is a topic much harder and wider than jigsaw puzzles. Permutation by semantic information requires much more background information, *e.g.* what is "smile" and what are "eyes", while extracting those knowledge from the images (where there exists too many other irrelevant features) is no easy task. Although DeepPermNet is surely one solution to the problem of jigsaw puzzles, it's expected that DeepPermNet won't yield the best possible results, which are more likely to be achieved by other solutions specialized at jigsaw puzzles. But we will still begin from reproducing and experimenting on DeepPermNet.

2.1. Architecture

In the paper the authors demonstrated their network architecture as is shown in Fig.2. I imitated such an architecture and designed a model whose "shared weights" part is based on the VGG blocks [6] featuring 3×3 convolution kernels that minimizes the parameters needed. The final architecture selected was shown in Fig.3.

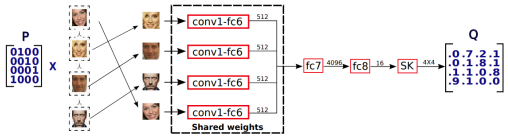

Figure 2. Architecture of DeepPermNet

As is shown in Fig.2, DeepPermNet uses *sinkhorn* normalization, and in the paper it has been proven by experiments that sinkhorn helps to get better results. However, I found that sinkhorn could make calculation extremely slow. On my laptop, training one epoch using my architecture without sinkhorn costs about 20 seconds, but it took about 30 minutes to train an epoch with sinkhorn. Despite the effect of sinkhorn confirmed by DeepPermNet, from my perspective, the spent computation is not worth the performance gain in the context of my class project. Therefore I choose to discard sinkhorn in training.

However, I tried using sinkhorn on already-trained models and found that it brings about steady performance gain. Therefore, I would still like to use sinkhorn when testing the model performance.

2.2. "Tricky" Pretraining

In almost all the deep-learning-based jigsaw puzzle solvers, the term "pretraining" is mentioned. Some works use self-supervised jigsaw solver to pretraining image classifiers, while the others operate in the reverse direction: pretrain image classifiers to solve jigsaw puzzles.

I have been tricked into thinking that pretraining should be performed, and then I started using pre-trained CIFAR-10 classifiers to solve the puzzle. It was a long time before I finally realized that what I'm doing was not pretraining.

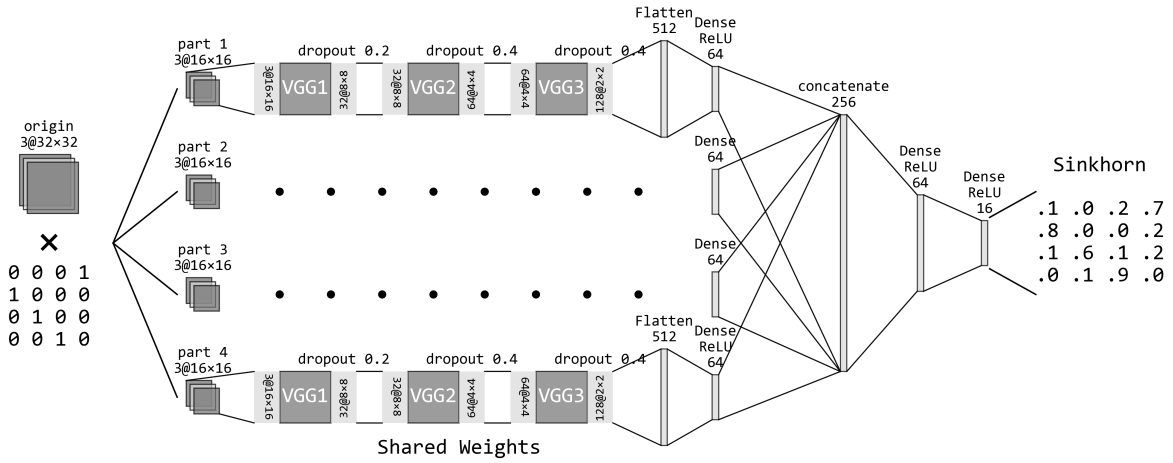

Figure 3. My Architecture used in this project.

Pretraining involves first training the model using another dataset, *e.g*. ImageNet, and then doing surpervised training on another dataset. The generlization abilities of DNN models makes up the essence of transfer learning. Such process won't make sense on the same dataset. By training the CIFAR-10 classifier we are doing nothing more than training jigsaw puzzle solver for some more iterations.

Consequently, the whole model with Fig.3 architecture is trained at the same time.

2.3. The Loss

The exact difficulty of generalizing a CNN image classifier to predict permutation lies in the hardness to represent $n!$ kinds of permutations efficiently. In DeepPermNet, *doubly stochastic matrices* are introduced to solve such difficulty. The authors said that they used *cross entropy loss* to train their models.

However, generating doubly stochastic matrices involves the very costly sinkhorn, which is what we would like to avoid. Therefore I tried several simpler ways of calculating losses from the outputs of neural network.

2.3.1 Mean Square Error Loss

Whatever data we get, we can always use *mean square error* (MSE) to represent how close we are to the original distribution statistically. Due to the difficulty of measuring the "distance" of a generated 4×4 vector to a permutation matrix, I chose MSE loss as my first trial.

2.3.2 Cross Entropy Loss

As is described in the paper, I tried *cross entropy loss* too. To be specific, for predicted matrix $\boldsymbol{P} \in \mathbb{R}^{4\times4}$ and the ground truth $\boldsymbol{A} \in \mathbb{R}^{4\times4}$, each row $\boldsymbol{p}_i = \boldsymbol{P}[i,:]$ and $\boldsymbol{a}_i = \boldsymbol{A}[i,:]$ are sent into `jt.nn.CrossEntropyLoss` as inputs.

It turned out that simply using cross entropy loss won't help the model converge. After running several epoches, the loss maintained a large value and the error rate on training set was still higher than 95%. I tried using columns instead of rows, but the problem persisted. The attempt to use cross entropy loss ended in failure.

From my perspective, the reason why the model failed to converge was that all the gradients were exactly the same except for the last layer. Then the changes of previously layers might cancel out each other, preventing the model from stepping closer to convergence.

2.3.3 Bi-directional Cross Entropy Loss

I don't know whether there already exists a same loss function, but I would briefly explain it. The function was very simple: adding the cross entropy loss calculated using columns and that using rows. Mathematically, given prediction $\boldsymbol{P} \in \mathbb{R}^{n\times n}$ and ground truth $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, the loss is

$$\ell(\boldsymbol{P}, \boldsymbol{A}) = \sum_{i=1}^{n} \mathrm{CE}(\boldsymbol{P}[i,:], \boldsymbol{A}[i,:]) + \sum_{j=1}^{n} \mathrm{CE}(\boldsymbol{P}[:,j], \boldsymbol{A}[:,j]) \tag{1}$$

where $\mathrm{CE}(\cdot,\cdot)$ accepting two vectors is the cross entropy loss function. In `jittor` one-hot encoding is not supported so $\arg\max_i \boldsymbol{A}[i,:]$ and $\arg\max_j \boldsymbol{A}[:,j]$ are sent instead.

I thought of such a function as an approximate of *sinkhorn + cross entropy loss*. Since ensuring both rows

and columns are summed up to one at the same time was hard, we could simply change rows and columns to sum up to one respectively. Then the vertical CE loss and horizontal CE loss works together to reduce probabilities given by wrong entries while increasing those by correct entries. This attempt turned out to work well.

2.4. Training Details and Experiments

In training, the optimizer was `jt.nn.Adam` with learning rate 10^{-3} and weight decay 10^{-4} by default. During tests, the sequence is predicted by taking the maximum of each row, and we calculate accuracy for our models. We take a sample as being predicted right if and only if all four items in the sequence is correct.

At first I chose to stack two VGG blocks and use MSE loss. I got 16.10% error rate on the training set and 20.97% on the test set (shown in Fig.4), and *sinkhorn* couldn't help much here. Therefore the most serious problem was underfitting: our model fails to capture the key information and utilize those features to predict the permutation.

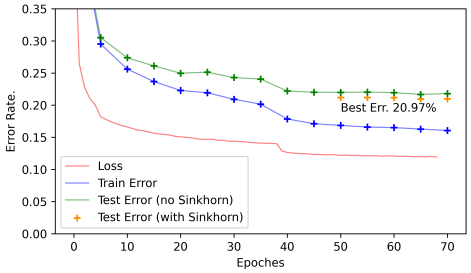

Figure 4. Performance got using mean squared error (2 VGG blocks).

Then I resorted to the *bi-directional cross entropy loss* defined as before. I found that using such a loss yielded a great improvement. Moreover, I found that after training for some epoches using learning rate of 10^{-3}, it would be of help to reduce the learning rate to 10^{-4} and train for another several epoches.

What's more, to solve the problem of underfitting, I stacked another VGG block to the model (then the model became what it looks like in Fig.3). For the model with two stacked VGG block, the model got 17.89% error rate with *sinkhorn* used in testing and 20.23% without (shown in Fig.5); for that with three, the corresponding error rates are 16.77% with *sinkhorn* and 18.71% without (shown in Fig.6).

2.5. Problems with DeepPermNet

I noticed that if we view the "shared weights" part as extraction of features and exclude sinkhorn, the remaining parts are simply fully-connected linear layers performing the role of predicting labels using extracted features. The

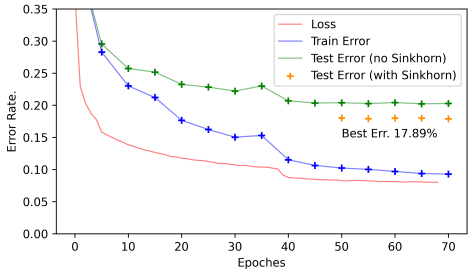

Figure 5. Performance got using bi-directiona cross entropy (2 VGG blocks).

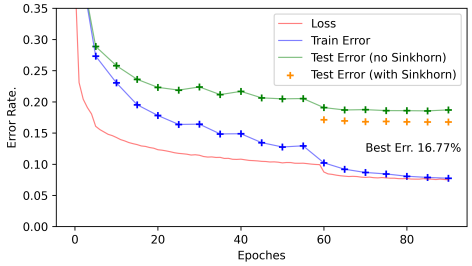

Figure 6. Performance got using bi-directiona cross entropy (3 VGG blocks).

sinkhorn layer is the only component related to "permutation".

In this sense, we are doing almost nothing more than treating the permutation problem (or jigsaw puzzle problem) as a variant of classification problem. This viewpoint is illustrated as follows. For a simple n-class classification, the desired model predicts the probability of item belonging to each class. However, in reality the work of mapping any $\mathbb{R}^n$ vector to a probability $(n-1)$-simplex is left to *softmax*. Symmetrically, in DeepPermNet, the desired doubly stochastic matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n} = [a_{ij}]_{n\times n}$ with a_{ij} representing the probability of item i belonging to position j is obtained from any $\mathbb{R}^{n\times n}$ matrix by *sinkhorn*.

However, the problem with treating jigsaw puzzles as classification is that there are $n!$ valid answers encoded in the $n \times n$ bool-value matrix (which has $2^{n\times n}$ value space). Not only is $n!$ too large a number for classification, we are also faced with illegal outputs. Although sinkhorn combined with the optimization problem mapping doubly-stochastic matrices to one of the $n!$ valid answers ensured validity, we have shown in previous experiments that such a routine is too computationally expensive.

Therefore, our following efforts are paid to jigsaw-puzzle-special solutions based on deep neural networks.

3. Other trials

It is straight-forward that matching the edges of all items is a solution to jigsaw puzzles, and in fact item edge matching is a basis of almost all traditional techniques. In a survey [2] it is stated that deep-learning-based techniques generally performs worse than traiditional techniques except one which introduced GAN [1]. I would prefer solving it using DL-based methods, and GAN requires more computation than I could afford. Therefore my methods won't include traditional techniques and the "JigsawGAN".

3.1. Trial 1: Position Prediction

Inspired by [3] and [4], I would like to test how can CNNs tell how to position two given pieces. Consider the process of us humans doing jigsaw puzzles. We see several pieces and our brains tell us that we can put these two pieces together. Then gradually we solve the puzzle.

Therefore I changed the problem to a 5-class classification problem. Given two pieces p_1 and p_2, there are five possible positions in total:

- p_1 is on top of p_2.
- p_2 is on top of p_1.
- p_1 is at left of p_2.
- p_2 is at left of p_1.
- p_1 is not a neighbour of p_2.

These 5 types of position relationships construct 5 labels. We crop the CIFAR samples to sample pairs and generate corresponding labels. Then we use a similar architecture to Fig.3 (exactly the same except that only two 64 vectors are concatenated and the final output is a 5-dimensional vector.

I had expected that such a model would generate much better results than DeepPermNet, but it didn't. Whatever techniques I used, the model could only achieve 83% accuracy, which was only similar to that achieved by the end-to-end permutation model. Then there was no reason to continue to try on this route since the accuracy obtained by solving a part of the problem was not higher than solving the whole problem itself.

Consequently, my first trial ended in failure.

3.2. Trial 2: Judge Whether a Picture is Natural

After the first failure, I focused on a simpler question that: how much can the neural network predict whether a picture is cropped and merged by wrong pieces or is naturally shot. Whatever method we might use, this task was undoubtedly the most basic problem in the jigsaw puzzle. I used the same model as in Sec.3.1 except that the output is a scalar value followed by a *sigmoid* function.

I generated positive-class samples by simply selecting any 16×16 part from a 32×32 CIFAR image. Meanwhile, the negative samples were generated by selecting two 8×16 or 16×8 parts at the boundary of a CIFAR image and merged those two pieces together. An example of negative example is given in Fig.7.

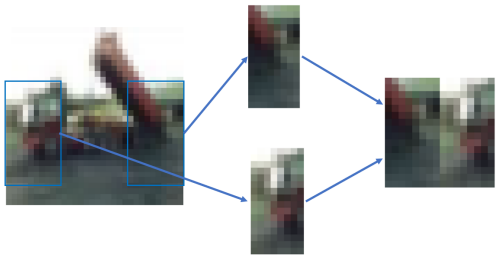

Figure 7. Example of negative sample

After training, I found that the model could achieve about 1.5% error rate on test set. Although the value seemed high enough, but in fact it is not satisfactory. After solving the boundary matching problem, we are faced with 8 horizontal boundaries (two for each piece) and 8 vertical boundaries, and there is in total 24 pairs of edges to match. Supposing all the judging results of the edge pairs were independent, we would have about $(1 - 1.5\%)^{24} \approx 69.6\%$ accuracy. This is far lower than what we have achieved by DeepPermNet.

Although I tried to learn the correlation between outputs of judging model and the label matrix using a MLP with the architecture shown in Fig.8, the model failed to converge. After several epoches there is no drop in loss and the error rate on training set is still higher than 95%.

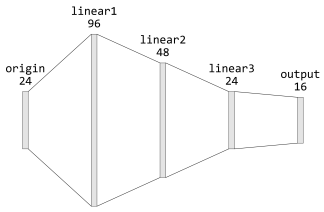

Figure 8. MLP Architecture.

Then my second trial also ended in failure.

4. Conclusion

In the above discussion, I have depicted a rough thinking routine that gradually arose during the class project. Now I think that DeepPermNet was still a best deep-learning based solution to jigsaw puzzles without introducing GAN.

Despite the defects described in Sec.2.5, "end-to-end" was the art of deep neural networks. Although people don't

know how each layer of the DeepPermNet was working, figuring out the meaning was the work of the network itself. Instead, humans only care about the performance of the final model. Human interference doesn't work as well as the model itself does naturally.

My final results are reported as: using the architecture shown in Fig.3, I achieved a minimum error rate of 16.77% by introducing the *bi-directional cross entropy*. To be frank, it is a disappointing result for me. However, for lack of time, my trials to solve jigsaw puzzles can only end here.

References

[1] Ru Li, Shuaicheng Liu, Guangfu Wang, Guanghui Liu, and Bing Zeng. Jigsawgan: Auxiliary learning for solving jigsaw puzzles with generative adversarial networks. *IEEE Transactions on Image Processing*, 31:513–524, 2021. 4

[2] Smaragda Markaki and Costas Panagiotakis. Jigsaw puzzle solving techniques and applications: a survey. *The Visual Computer*, pages 1–17, 2022. 4

[3] Marie-Morgane Paumard, David Picard, and Hedi Tabia. Jigsaw puzzle solving using local feature co-occurrences in deep neural networks. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 1018–1022. IEEE, 2018. 4

[4] Marie-Morgane Paumard, David Picard, and Hedi Tabia. Deepzzle: Solving visual jigsaw puzzles with deep learning and shortest path optimization. *IEEE Transactions on Image Processing*, 29:3569–3581, 2020. 4

[5] Rodrigo Santa Cruz, Basura Fernando, Anoop Cherian, and Stephen Gould. Deeppermnet: Visual permutation learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3949–3957, 2017. 1

[6] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 1