Solving the Jigsaw Puzzle via Various Learning Approaches

Kyra Chow, Ray Coden Mercurius , Anna Xu
Department of Computer Science
University of Toronto
Toronto, ON M5S2E4
csc413-2022-01@cs.toronto.edu

Abstract

The jigsaw puzzle is a game dating back to the 1760s and despite years of developments in both game and technology, there is a lack of literature utilizing various machine learning (ML) techniques to solve this game. To tackle this issue, we developed a learning-based solver which draws inspiration from multiple existing ML methods involving computer vision. Through analyses of each model's performance of solving the puzzle, we then determine whether certain model architectures are preferred over others for this task.

1 Introduction

A square jigsaw is a subset of jigsaw puzzles where each puzzle tile is a square and the correct configuration of the tiles displays an n-by-n grid. To solve a square jigsaw, the user must first determine the correct tile configuration (position and orientation). In this project, we focus on using computer vision techniques in machine learning (ie. convolutional neural networks (CNNs)) to identify the correct tile configuration. We will then compare the solving performance of various Deep CNN (DCNN) architectures. The motivation for this project follows from the lack of diverse machine learning-based literature surrounding jigsaw DCNN solvers. We hope that our conclusions may be used to better select models for similar tasks and increase model explainability.

2 Related Works

There are 2 general approaches to solving square jigsaws.

Firstly, one can solve the board in an iterative process. This is done by computing an edge compatibility metric between sub-image pairs, and using this information to prune possibilities. Zanoci and Andress computed an edge compatibility metric using non-ML neighboring pixel information and the color gradients [1]. This was done between all possible sub-image pairs ($n^2 * 16$ total comparisons, where n = number of sub-images). Then, treating pieces as vertices and compatibility scores as edges, they utilized the minimum spanning tree algorithm to recreate the original board using the most likely edges. Only 80% of edges were correctly matched, using board sizes of hundreds of pieces. A downfall of non-ML methods is the deterministic nature of the solver, and its difficulty to adapt to simple perturbations such as eroded edges. A simpler method proposed by Bhaskhar, uses a fine-tuned ResNet to compute edge compatabilities between puzzle tiles. Then the grid is iteratively filled in, using the most probable tile at each time step. As a result, over 80% of pieces were rearranged into their correct position/orientation [2].

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

The second approach is for one to attempt to solve the entire board at one time. Noroozi separately fed each sub-piece into a siamese-ennead CNN, then combined these encodings in a fully connected layer to predict the original permutation [3]. Accuracy in terms of solving entire board was low at only 67% with a board size of 3x3. Given that 9! permutations exist, solving at once may be too complex for current architectures. A similar method by Kulharia [4] used a many to one long short-term memory (LSTM) to output a one-hot encoding of the permutation. Input at each time-step was a collapsed sub-image. It only achieved 80% accuracy per complete board solve despite a small board size of 2x2. We will expand on this approach in our project.

3 Method

3.1 Image Data Source

We will create the dataset for our report using the Caltech-UCSD Birds 200 (CUB_200) 2011 dataset [5]. CUB-200 2011 is an image dataset containing 11,788 images of 200 different bird species. Using a 70/30 split, we will partition these images into training and validation datasets, with 8,252 images in the training set and 3,536 in the validation set.

3.2 Edge Compatibility Train Data

Our DCNN's input is a square concatenation of 2 subimages, and the output is whether they are left-right adjacent, which we will now refer to as adjacent. To generate these input output pairings we make use of Bhaskhar's `Checking_adjacency_dataset` repository [2], which will generate edge concatenations between the images' various edges (See Fig 4). For each n-by-n grid, there are n^2 puzzle pieces and C(n^2, 2) possible pairs of square concatenations. The repository will then randomly choose from these pairs to create our train data that has a roughly equal number of adjacent labels, and non-adjacent labels, and will output our newly formed data.

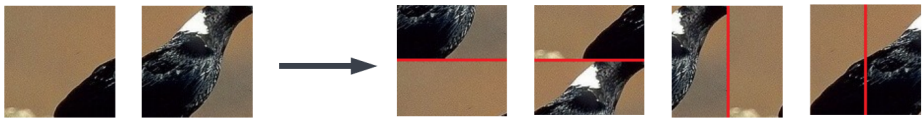

Figure 1: 2 sub-images can generate 4 different concatenations.

3.3 Adjacency Classifier

The goal of this classifier is to determine whether an input image, consisting of two puzzle pieces, (P,Q), is left-right adjacent. Drawing inspiration from the work of [2], we will finetune different image classification models, such as ResNet, on a custom adjacency dataset to create the adjacency classifier. Furthermore, we want to compare the performance of these models both in terms of their prediction accuracy as adjacency classifiers, and how well they work in combination with the downstream solver to complete the puzzle. In the following sections we will briefly describe the architecture of each CNN used, and how we have adapted the model to fit our finetuning process.

3.3.1 ResNet Architecture

The ResNet models were first proposed in [6]. There are currently five versions of ResNet (ResNet18, ResNet34, ResNet50, ResNet101, ResNet152) trained on the ImageNet dataset available for fine tuning in the `torchvision` library in PyTorch. In general, the ResNet models contain convolutional kernels of size 3x3, and the number of filter layers increases along with the depth of the network. Thus, different versions of the ResNets correspond to the total number of filter layers in the network (see Fig. 5 for architecture detail). The defining feature for ResNets are the residual blocks which exist between pairs/triplets of the convolutional layers with residual connections that propagate the

unaltered input to the output (see Fig. 6). This innovation was introduced in order to prevent the vanishing gradient problem.

To adapt ResNets for our finetuning process, we modified the final fully connected layer originally meant for a larger classification task to one meant for binary classification. Note that when we finetuned on all the ResNets, the weights/bias parameters that are affected belong to this final classification layer.

3.3.2 VGG Architecture

The VGG models were first proposed in [7]. There are 4 versions of VGG (VGG11, VGG13, VGG16, VGG19) with batch normalization implemented in the `torchvision` library in PyTorch. Similar to ResNets, the VGGs contain only 3x3 convolution filters; however, they are considerably more shallow (the deepest VGG has 19 total layers, while the deepest ResNet has 152) (see. Fig. 7). The limitation on the VGGs' depth is due to the vanishing gradient problem, where more layers lead to issues with weight updates during training.

To adapt the VGGs for our finetuning process, we have to again modify the final classification layer to fit our binary classification task.

3.3.3 ResNeXt Architecture

The ResNeXt models were first proposed in [8](See Fig. 8). What sets the ResNeXts apart from ResNets are the structure of the residual blocks. Instead of applying convolution filters to the block's entire input feature map, a series of convolutions are applied to parts of the the input and then the output is then grouped back together(see Fig. 9). This parallel structure allows specialization to occur among groups where different characteristics are learned (ie. edges, corners etc).

To adapt the ResNeXts for our finetuning process, we modified the final fully connected layer exactly like how we had done for the ResNets.

3.3.4 Setup

For the sake of fair comparison, all CNN models were tested on control hyper parameters (see Table 1).

3.4 Configuration Solver

The configuration solver takes as input a randomly scrambled n-by-n board (see Fig. 10 - 12). Due to time constraints, this portion of the implementation leveraged existing functions from [2]. At each time step the solver receives a board that is partially filled in, and outputs a new board with 1 additional piece adjacent to a piece already filled in. To decide on the next piece, the solver will try every single combination of (open spot, remaining piece, orientation). That is if there are S open spots, R remaining pieces, and 4 orientations, our solver will attempt S*R*4 combinations, and pick the most probable one according to the average edge compatibility scores of all its edges. The validity of an edge of 2 pieces stacked vertically can be checked by simply cutting out the edge, rotating the edge 90 degrees, and then passing it in to the left-right adjacent CNN classifier. The solver process is visualized in Figure 2 of the appendix.

4 Results and Discussion

4.1 Adjacency Classifier Performance

Overall all, we see that all models achieved their highest validation accuracy between 0.90 and 0.94, as their lowest validation loss between <0.15 and 0.36. In general, we see that deeper, more complex

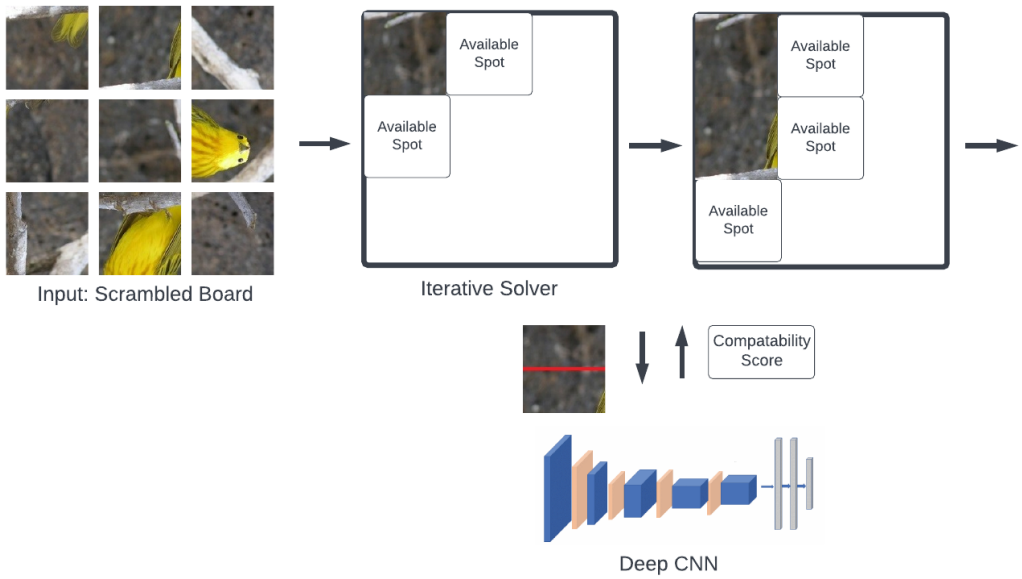

Figure 2: Architecture of Solving Process

models do not offer better generalizability for our problem, instead, shallower, simpler models are more capable.

In terms of individual model performances, we achieved the highest training accuracy of 0.99 and lowest training loss of 0.05 on VGG13, and highest validation accuracy of 0.94 and lowest validation loss of <0.15 on VGG16.

4.2 Final Solver

To test how well our adjacency classifier works in combination with the downstream configuration solver, we run the entire pipeline (ie. adjacency classifier + configuration solver) on two test images (see Fig. 10-12 for sample input/output). In the interest of time we did not run a complete performance study with more samples. However, what we observe from our limited test set is that VGG16 performed the best (ie. it gave a correct complete solution) in terms of solving a 3x3 board, and a 2x2 board. This result corroborate with our above observation as the adjacency classifier created using VGG16 gave the best validation loss/accuracy (see Table. 4 -5).

4.3 Discussion

Looking at our training and validation curves, we observe that the bigger ResNet models (ie. ResNet101 and ResNet152) suffers overfitting - we see the training accuracy improving whilst the validation accuracy worsens (the same can be seen for the loss curves). This is reasonable as the size of our entire dataset is rather small which means that more complex models will easy overfit to the training data and perform suboptimally in validation.

For the VGG curves, however, the validation curves are very noisy and have significant jumps in accuracy and loss. This suggest to us that the learning rate for the model is set too high - the gradient descent algorithm has trouble stabilizing near the local minimum because the step size for updates are too large which causes large oscillations. In the interest of time we did not remedy this issue but trained all models on the same control settings.

Importantly, all Deep CNNs hit an adjacency classfication accuracy cap near 94%. We speculate that this issue may be due to the nature of the dataset itself. Our dataset contains images of birds

with natural landscapes that include the sky and water. Unlike in commercial puzzles, our pieces were never checked to maintain a good balance of distinguishing features across all sub images. It is possible that many sub images contained the same generic background (Refer to Figure 3 in Appendix) lacking in major features. Here, we require our CNN to pay strong attention to pixel level details. This might be challenging to Deep CNNs pretrained on object recognition, where localized pixel level information is lost in its deeper later layers, especially since we only fine tuned the final classification layer.

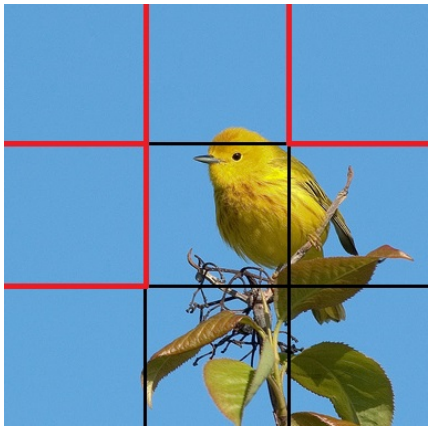

Figure 3: The edges marked in red do not have defining features. This imbalance across puzzle tiles could have yielded the accuracy cap of 95%.

Finally, we want to observe the effect of using different CNNs for adjacency classification has on the downstream solver. From our limited results, we see that the overall solver performance can be improved (only VGG16 + solver gave a completely correct solution for both test samples).

5 Conclusion and Next Steps

In this report, we expanded on existing methods by showing that various pretrained Deep CNNs can be successfully fine-tuned to generate working adjacency scoring metrics. Importantly, we concluded that smaller, wider CNNs (such as VGGs) obtain higher accuracies by paying finer attention to details than longer, narrower models (such as ResNets) that are pretrained for object classification.

5.1 Future Work - Train Data

In real life, jigsaw tile edges are eroded inwards anywhere from 1% to 5%, which would be a perfect challenge for machine learning. In future works, we can test and explore the effects of various degrees of erosion on edge adjacency classification performance.

5.2 Future Work - Solver

As well, in the future we can tackle the issue of the propagation of errors downstream. Incorrectly placed pieces cannot be readjusted, as well some pieces are placed with only 1 neighboring edge information which introduces significant uncertainty. To remedy this, we propose solving in batches. That is, we can simultaneously solve divisions of 2x2 at a time by brute forcing 8!/4! combinations, taking the most likely combination according to chained edge compatibility scores. Finally, we can have a top-down double check by having a CNN take in a subdivision and output its validity, then backtracking if necessary.

References

[1] C. Zanoci and J. Andress, "Making puzzles less puzzling: An automatic jigsaw puzzle solver.," 2016.

[2] N. Bhaskhar, *Nivbhaskhar/unpuzzled: A jigsaw puzzle solver using ai*. [Online]. Available: https://github.com/nivbhaskhar/UnPuzzled/.

[3] M. Noroozi and P. Favaro, "Unsupervised learning of visual representations by solving jigsaw puzzles," *Computer Vision – ECCV 2016*, pp. 69–84, 2016. DOI: 10.1007/978-3-319-46466-4_5.

[4] V. Kulharia, A. Ghosh, N. Patil, and P. K. Rai, "Neural perspective to jigsaw puzzle solving," 2016.

[5] C. I. of Technology, *Caltech-ucsd birds 200 2011 dataset*, 2011. [Online]. Available: https://www.tensorflow.org/datasets/catalog/caltech_birds2011.

[6] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, 2015. DOI: 10.48550/ARXIV.1512.03385. [Online]. Available: https://arxiv.org/abs/1512.03385.

[7] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, 2014. DOI: 10.48550/ARXIV.1409.1556. [Online]. Available: https://arxiv.org/abs/1409.1556.

[8] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, *Aggregated residual transformations for deep neural networks*, 2016. DOI: 10.48550/ARXIV.1611.05431. [Online]. Available: https://arxiv.org/abs/1611.05431.

Appendix

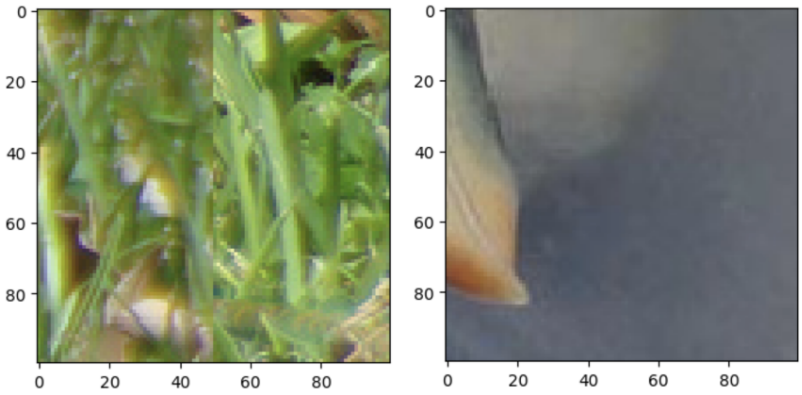

Figure 4: Two square concatenations of puzzle piece edges. On the left, we see that the edges are not left-right adjacent and thus they will have label 0. Whereas on the right we see that the edges are left-right adjacent, and thus they will have label 1.

Hyperparameter	Value
learning rate	0.001
momentum	0.9
examples per epoch	500
optimizer	optim.SGD

Table 1: Hyperparameters for all CNN Models

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

ures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of block

Figure 5: Model architecture comparison between the different ResNets

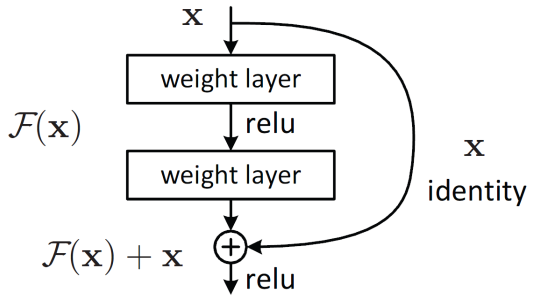

Figure 6: Illustration of a residual block structure

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 7: Model architecture comparison between the different VGGs

stage	output	ResNet-50	**ResNeXt-50 (32×4d)**
conv1	112×112	7×7, 64, stride 2	7×7, 64, stride 2
conv2	56×56	3×3 max pool, stride 2	3×3 max pool, stride 2
		$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128, C{=}32 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3	28×28	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256, C{=}32 \\ 1\times1, 512 \end{bmatrix}\times4$
conv4	14×14	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512, C{=}32 \\ 1\times1, 1024 \end{bmatrix}\times6$
conv5	7×7	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 1024 \\ 3\times3, 1024, C{=}32 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	global average pool 1000-d fc, softmax	global average pool 1000-d fc, softmax
# params.		**25.5**×10^6	**25.0**×10^6
FLOPs		**4.1**×10^9	**4.2**×10^9

Table 1. (**Left**) ResNet-50. (**Right**) ResNeXt-50 with a 32×4d template (using the reformulation in Fig. 3(c)). Inside the brackets are the shape of a residual block, and outside the brackets is the number of stacked blocks on a stage. "C=32" suggests grouped convolutions [24] with 32 groups. *The numbers of parameters and FLOPs are similar between these two models.*

Figure 8: Model architecture comparison between ResNet50 and ResNeXt50

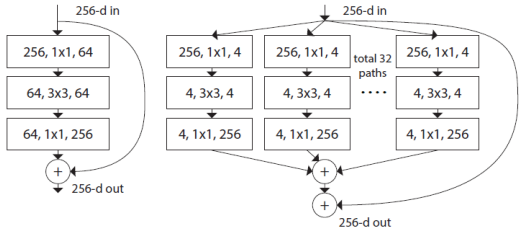

Figure 9: Illustration of a ResNeXt residual block structure compared to the a ResNet residual block structure

Model	Lowest Training Loss	Highest Training Accuracy
Resnet18	0.10	0.97
Resnet34	0.11	0.96
Resnet50	0.10	0.96
Resnet101	0.14	0.95
Resnet152	0.09	0.97
VGG11	0.05	0.98
VGG13	**0.05**	**0.99**
VGG16	0.06	0.98
VGG19	0.06	0.98
ResNeXt50	0.11	0.97
ResNeXt101	0.31	0.92

Table 2: The lowest training loss and highest training accuracy observed for each model

Model	Lowest Validation Loss	Highest Validation Accuracy
Resnet18	0.31	0.92
Resnet34	0.27	0.93
Resnet50	0.35	0.92
Resnet101	0.29	0.93
Resnet152	0.36	0.93
VGG11	0.23	0.92
VGG13	0.27	0.92
VGG16	**<0.15**	**0.94**
VGG19	0.20	0.91
ResNeXt50	0.26	0.92
ResNeXt101	0.21	0.92

Table 3: The lowest validation loss and highest validation accuracy observed for each model

Figure 10: Original test image

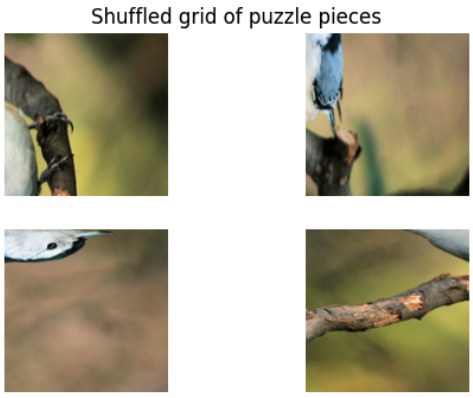

Figure 11: Sample input to solver

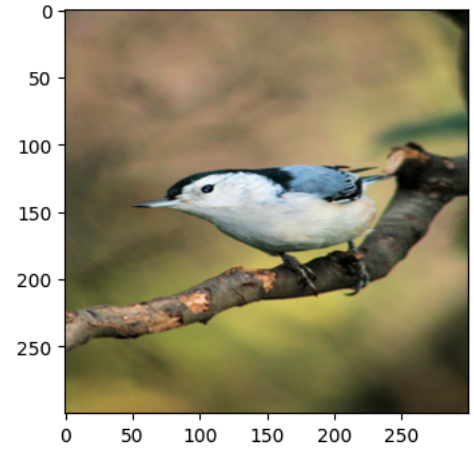

Figure 12: Final solver output

Model Name	# of correct pos.	# of correct pos. and rot.
ResNet18	3	3
ResNet34	1	1
ResNet50	1	1
ResNet101	3	3
ResNet152	1	1
VGG11	**9**	**9**
VGG13	4	4
VGG16	1	1
VGG19	2	1
ResNeXt50	6	6
ResNeXt101	3	3

Table 4: Solver result for a 3x3 board. The correct board should feature 9 correct positions and 9 correct rotations

Model Name	# of correct pos.	# of correct pos. and rot.
ResNet18	**4**	**4**
ResNet34	2	1
ResNet50	2	2
ResNet101	**4**	**4**
ResNet152	4	2
VGG11	**4**	**4**
VGG13	**4**	**4**
VGG16	2	2
VGG19	**4**	**4**
ResNeXt50	**4**	**4**
ResNeXt101	**4**	**4**

Table 5: Solver result for a 2x2 board. The correct board should feature 4 correct positions and 4 correct rotations