Unsupervised Domain Adaptation Approaches for Chessboard Recognition

Wassim Jabbour
McGill University
wassim.jabbour@mail.mcgill.ca

Enzo Benoit-Jeannin
McGill University
enzo.benoit-jeannin@mail.mcgill.ca

Oscar Bedford
McGill University
oscar.bedford@mail.mcgill.ca

Saif Shahin
McGill University
saif.shahin@mail.mcgill.ca

Abstract

Chess involves extensive study and requires players to keep manual records of their matches, a process which is time-consuming and distracting. The lack of high-quality labelled photographs of chess boards, and the tediousness of manual labelling, have hindered the wide application of Deep Learning (DL) to automating this record-keeping process. This paper proposes and end-to-end pipeline that employs domain adaptation (DA) to predict the labels of chess pieces in 500 unlabelled top-view images of chess boards (Target domain), using 288,000 3D images of individual chessboard squares that were generated (Source domain). The pipeline is composed of a pre-processing phase which detects the board, crops the individual squares, and feeds them one at a time to a DL model. The model then predicts the labels of the squares and passes the ordered predictions to a post-processing pipeline which generates the Forsyth–Edwards Notation (FEN) (1) of the position, followed by a 2D reconstruction of the image. Because the pre-processing and post-processing phases do not involve learning weights, this paper focuses on how the DL model was optimized to predict unlabelled target domain squares given labelled source domain squares. The three approaches considered are the following: A VGG16 (2) pre-trained on ImageNet (3), defined here as the "Base model", fine-tuned to predict source domain squares and then used to predict target domain squares without any domain adaptation; an upgraded version of the Base model which applied CORAL loss (4) to some of the final fully connected layers of the VGG16 to implement DA; and a Domain Adversarial Neural Network (DANN) (5) which used the adversarial training of a domain discriminator to perform the DA. Because a realistic chess position mostly constitutes of empty squares which are easy to classify, target domain data was oversampled to balance out the differing class proportions, ensuring that the testing results presented do not actually reflect the full potential of the pipeline in a real-life situation. Nonetheless, the testing metrics achieved by the different models on the balanced target domain data can be found in the table below:

Metric	DANN	CORAL	Base
Accuracy	83.80%	83.68%	61.97%
Weighted F1 score	0.847	0.835	0.592
Average AUPRC	0.858	0.847	0.641

Table 1: Summary of the main results

1 Introduction

Chess, a centuries-old strategic board game, remains popular among enthusiasts and masters alike. In long-format competitive play, every move executed by each side must be manually recorded using pen and paper. One reason for this is post-game analysis, where chess players will review their games in order to improve their strategies and skills. Another reason is to resolve conflict about illegal moves or to determine draws by number of repetitions, among others. The need for an automatic record of chess moves on physical boards emerges from the time-consuming nature of manual record keeping for both long-format and short-format games. While this is an area where Deep Learning can be applied, there has not been much progress. This can be attributed to the difficulty of gathering enough labelled data, as there are up to 32 pieces and associated locations to label per image. This paper proposes an unsupervised approach to automatic annotation of chessboard photographs. The approach centers around the concept of unsupervised domain adaptation, a technique used to improve the performance of a model on a target domain containing no labelled data by using the knowledge learned by the model from a related source domain with an abundance of labelled data with a slightly different distribution (6). The source domain data employed to perform the domain adaptation consists of 3D images of chessboards rendered using Blender (7), as these are simple to generate in large numbers and can be designed to match the distribution of the target domain. The target domain data employed was unlabelled top-view photographs of chess positions.

From a broader perspective, the proposed solution consists of 3 components: A pre-processing pipeline which takes a full target domain photograph as input, detects the board, and crops out the individual squares. Then, the individual squares are passed one at a time to a Deep Learning model trained using domain adaptation which can classify the labels of the chess pieces on each square. Finally, the ordered predictions of the model are passed to a post-processing pipeline which generates a FEN string representing the position that can be fed to a chess engine to generate a complete 2D representation of the input.

In order for the proposed Domain Adaptation model to be successful, there are several crucial constraints that must be addressed. Firstly, the 3D generated data must closely resemble target domain photographs in terms of camera angle, lighting, board colors, and piece textures, ensuring that the extent of domain adaptation is minimized. Secondly, this project is circumscribed to automating the annotation of individual images, rather than full videos. Finally, on a higher level, a goal-oriented constraint is that the model is expected to perform significantly better in classifying the target domain squares than a model trained on the source data and directly tested on the target data without any domain adaptation.

This project's synthetic 3D chess images and the domain adaptation model are freely available for non-commercial use. Source code is provided under an open-source license at https://github.com/WassimJabz/RecogniChess, implemented in Python, and supported on major platforms. Additional figures, data, and methodology can be accessed in the same repository.

2 Methodology

2.1 Datasets

2.1.1 Target dataset

The target dataset comprises 500 photographs of actual chess positions displayed on a chessboard, as demonstrated in Figure 1. These images were made available thanks to an open-source effort by fellow chess enthusiasts (8). The images purposefully include a noisy background around the board to simulate real-life situations where getting clear images of the board might not always be possible, especially in short format games. A pre-processing pipeline using Computer Vision was designed to detect the board in the image, warp it to top-view, and crop out individual squares. This process will be explained in Section 2.2.1.

It is to be noted that, while labels of the positions were provided with this dataset, they were discarded during model training in order to simulate an unsupervised domain adaptation situation. However, the labels were utilized during validation and testing in order to successfully tune the hyperparameters and evaluate the performance of the model, as will be expanded on in Section 2.5.

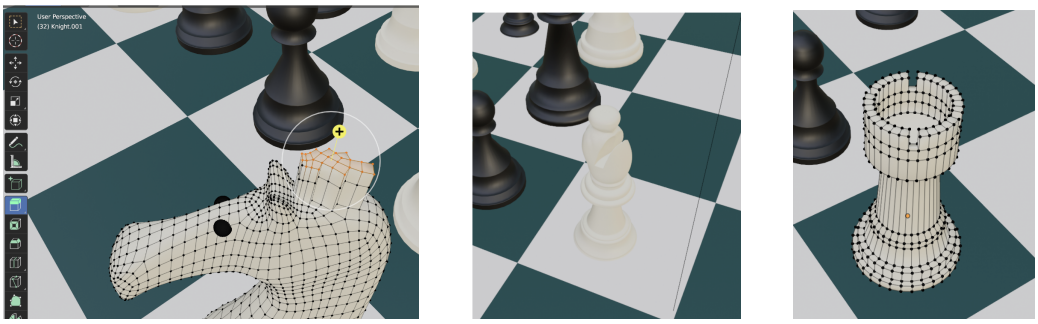
Figure 1: Example of an image taken from the target dataset.

We would like to thank the creators, Kurt Convey, Michael Deng, Samuel Ryan and Mukund Venkateswaran, for their contribution to chess position recognition development.

2.1.2 Source dataset generation

2.1.2.1 Blender modeling and Python Script

The source dataset is comprised of 288,000 distinct images of chess squares extracted from a singular virtual chessboard. These images were generated using the open-source 3D-rendering software, Blender (7). The pre-designed set of 3D chess pieces, including the board, were obtained from an online source (9) and imported into the software. Using Blender, the texture of both the chessboard and the chess pieces was customized to replicate the real-life dataset. This involved transforming the white pieces to a subtle beige hue, the black pieces to a darker shade, and the chessboard to a combination of green and white tones. Certain chess pieces from the imported pre-designed set exhibited noticeable visual differences when compared to their real-life counterparts from a top-view perspective. In order to rectify this aesthetic, more suitable pieces were imported from other online sources (10; 11), which were further refined using the 3D tools available in Blender. This is shown for a knight piece in Figure 2a. Similarly, a separate bishop piece was imported from another online source (11), and used after re-scaling and re-shaping its top form as shown on Figure 2b. Finally, the original pre-designed rook was modified as depicted on Figure 2c.



(a) Modification of the knight's crest. (b) Final Bishop 3D model. (c) Final Rook 3D model.

Figure 2: 3D Chess Piece Modifications in Blender.

The final 3D chess board and pieces are shown in Figure 3.

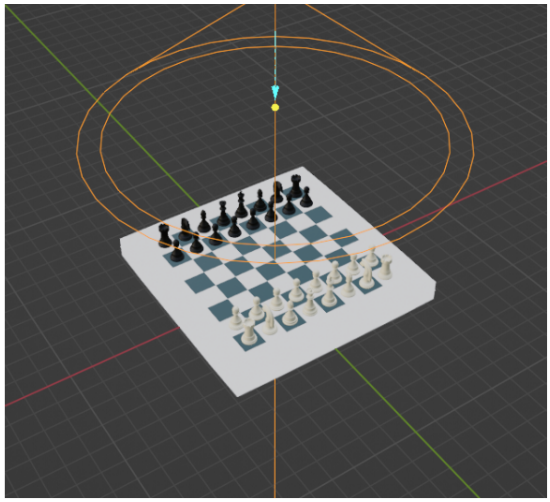


Figure 3: Blender model after texturing and modeling the chess pieces.

2.1.2.2 Blender rendering

After texturing and modeling each of the chess pieces, a Python script was written to automate the rendering process. This script randomly assigns chess pieces to the square positions, ranging from A1 to H8. Each chess piece is randomly placed with varying degrees of centering and rotation within its assigned square. Additionally, the generated images include a proportion of each chess piece that mimics a realistic chess position, with every image incorporating both white and black kings. This guarantees a comparable class distribution to that of the real-life dataset. Further, because a continuation of this work could include using the rules of chess to apply constraints to the predictions of the model, for instance the impossibility of having more than 2 kings on the board, the generated positions were constrained to satisfy every chess rule. Finally, it is to be noted that three distinct lighting scenes observed in the original dataset were replicated within Blender. Prior to rendering, the Python (12) script selects a lighting scene by randomly adding or removing light objects, thereby simulating the varying lighting conditions. All images were rendered from a camera object placed at the top of the board in order to maintain a consistent top-view perspective. Finally, a crop was applied during the rendering process to remove the rim of the board and exclusively display the 64 squares. A total of 4500 images were generated using this Python script.

2.1.2.3 Cropping individual squares

Following the generation of the chessboard images, an external Python script was defined to divide these images into individual squares. Given that all images were rendered from a stationary camera perspective in Blender, the coordinates and dimensions of each square (A1 through H8) remained consistent across all examples. As such, a representative image was divided into the 64 squares by employing the Grid tool in Adobe Photoshop (13).

Each square was then resized to be 100x100 pixels and saved in a folder dedicated to its parent position, along with a global label array. Empty squares were assigned the label 0. Additionally, the encoding used to represent the different pieces can be found in Table 2.

Color	Pawn	Knight	Bishop	Rook	Queen	King
White	1	2	3	4	5	6
Black	7	8	9	10	11	12

Table 2: Table showing the conversion for converting chess labels to numerical values.

2.2 Pre-processing

2.2.1 Target data

2.2.1.1 Contextualizing the pre-processing phase

Ideally, the final pipeline would include pre-processing the full, unprocessed, target domain image in real-time before passing the individual squares to the trained model. However, using this approach during model training has multiple downsides, such as not being able to reduce the number of empty squares beforehand, as well as, more importantly, the much slower training time due to having to detect the board and crop the individual squares for every image of every batch as training progresses. Therefore, the pre-processing pipeline was run on every single target domain image beforehand, which simplified training the model significantly. That being said, the reader should keep in mind that in a realistic testing situation the pre-processing would be run as part of the full pipeline, and the outputs of this phase would be passed to the model in real-time.

2.2.1.2 Overview

The pre-processing phase consisted of applying a number of filters to each input target domain image in order to sequentially detect the board, warp it to the top-view, and crop the individual squares. The algorithm used was inspired by (14), and uses a number of filters to perform the board detection and warping, as can be shown in the overview of Figure 4. Please note that the computer vision part was not the main focus of this paper, and thus the mathematical details of filters will be omitted for conciseness.

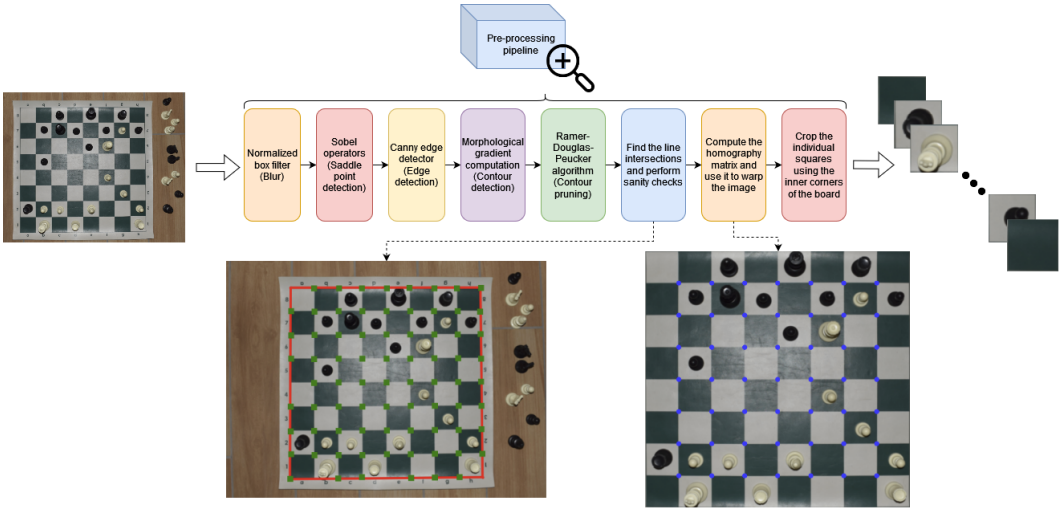


Figure 4: Full overview of the pre-processing pipeline

The open-sourced library used to implement the algorithm can be found in (15). We would like to thank the authors for making this possible.

2.2.1.3 Detecting the board

The board was detected using a combination of Sobel operators (16) to find the saddle points, Canny edge detection (17) to find the edges, and a mix of morphological gradient computations (18), as well as the Ramer-Douglas-Peucker algorithm (19) to detect and prune the contours, only keeping the straight lines of the image. The intersections of the straight lines are then used to find the reference points of the board, where a reference point is either defined as one of the board's corners (4 outermost points), or as the intersection between green and white squares. The outer contour of the board can then be located by linking the reference points that are located around the outer edges of the board.

2.2.1.4 Warping the board and cropping the squares

Using the 4 outer corners located using the method outlined in the previous section, we can compute a 3x3 homography matrix (20) which will help in warping the board to the top view. For a warped image of size $H \times W$, we then find $L = min(\frac{H}{8}, \frac{W}{8})$, and subsequently crop the board into 64 squares of side length L. The reason behind using a fixed value instead of utilizing the inner warped reference points of the board is that this approach was simpler and led to the same visual results, while also yielding squares of equal size. These squares were then resized to be 100x100 pixels to match the generated source domain data.

2.2.1.5 Partial Invariance to camera angle

It is important to highlight that the approach used for detecting and cropping the board means that, as long as the top of the pieces is visible enough, any image can be warped into full top-view angle and subsequently have its squares passed to the model for predictions. This applies even if the original camera angle is not entirely top-view, or even if the board is rotated, as shown in Figure 5.

Figure 5: Board detection on a side view image

However, the downside of this approach is that, as the angle approaches the side-view, the pieces that are on the opposite side of the board relative to the camera get cut in half by the warping algorithm. Even for the example in Figure 5, there will result a partial cut of the green king as well as the two green rooks, as shown in the warped result in Figure 6.

This issue will be further worsened by the square cropping algorithm, because some pieces such as the green bishop now span two squares. Taken together, these issues are the reason why the current project utilized images with a top-view camera angle. That said, an interesting continuation of this project would be to expand to other views by modifying the warping and cropping algorithms.

2.2.2 Imbalance reduction and dataset distributions

The histograms in Figure 7 display the distributions of both the target data after its pre-processing and the source data after its generation. It is clear that the class distributions are imbalanced due to a high percentage of empty squares, accounting for approximately 70% of both datasets. Given that this class is relatively easy to predict, and to ensure that model performance was not over-inflated due to predicting the majority class correctly, some examples of empty squares were removed from the two datasets. However, a higher proportion of empty squares was still maintained in order to reflect a real-life scenario where there are typically more empty than occupied squares on a chessboard.

The final distribution of the real-life dataset is depicted in Figure 8a. The same process of removing empty squares was applied to the generated source data, as depicted in Figure 8b. Additionally, an

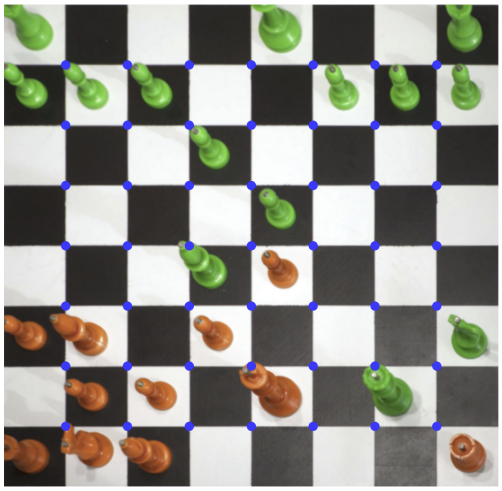
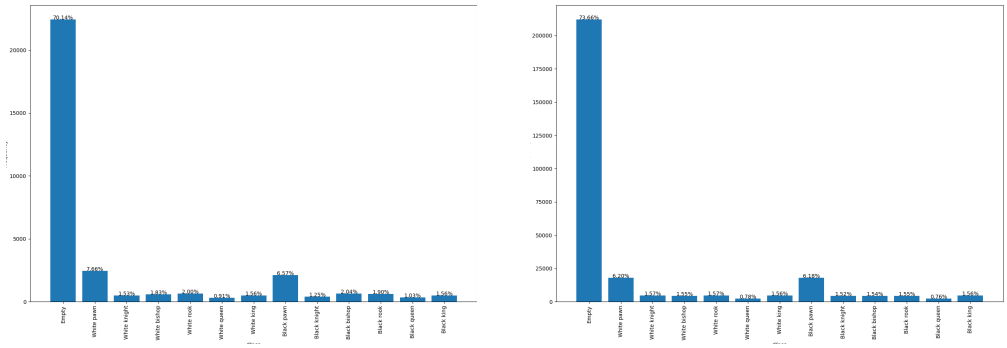
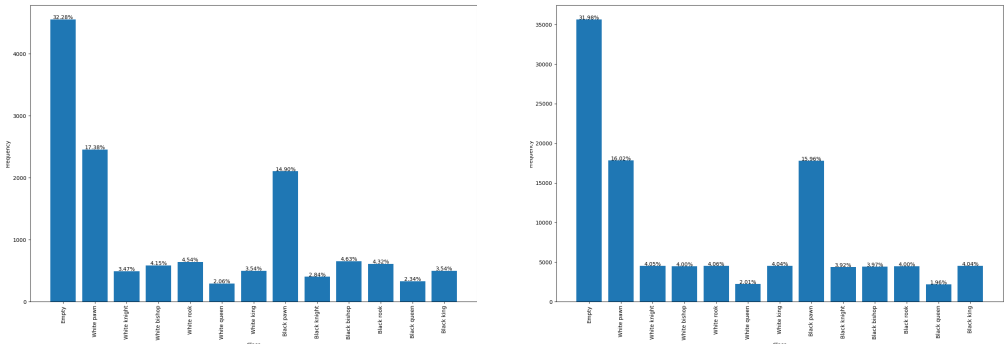


Figure 6: Board warping on a side view image



(a) Target dataset (b) Source dataset

Figure 7: Class distributions before re-balancing.



(a) Target dataset (b) Source dataset

Figure 8: Final class distributions.

important item to highlight is that the distribution of individual chess pieces is remarkably similar across distributions, underlining the accuracy of the synthetic data generation.

Both datasets were then split into training, testing and validation sets after being randomly shuffled with the following proportions: 80% for the training set and 10% for both the validation and testing set.

2.2.3 Oversampling

In order to further resolve the challenge of class imbalances in the datasets, the minority classes were oversampled. Indeed, class imbalance can negatively impact the performance of models, as they may struggle to effectively learn from under-represented classes. To address this issue, the "WeightedRandomSampler" provided by PyTorch (21) was leveraged during data loading. This sampler oversamples the minority classes, ensuring that each batch used for training received a balanced representation of samples from all 13 classes.

2.2.4 Data augmentation

Because oversampling entails that some training examples will be used much more frequently than others, and to further improve the model's ability to handle diverse lighting and orientation conditions, additional data augmentation techniques were applied to both datasets. This was achieved by utilizing the "transforms" module provided by PyTorch (21) during the data loading process. The applied transformations included random horizontal and vertical flips, as well as adjustments to the brightness of the image following a uniform distribution. The aim of the augmentations is to help the model in recognizing chess pieces under varying lighting conditions and orientations, thereby increasing its robustness.

2.2.5 Data loading

Due to the heavy size of the images, loading the full datasets into memory at once was not feasible because it required more RAM than the available computing devices could offer. To overcome this issue, a lazy data loading approach was implemented, whereby instead of loading the whole dataset at a time, the individual minibatches of images were loaded from the hard disk drive (HDD) as needed, and were subsequently discarded after the training iteration. This approach ensured that the training process could be initiated without concern for memory limitations.

2.3 Prediction post-processing

During testing, the 64 individual labels predicted by the model for a given chess position are passed to a post-processing pipeline which generates the Forsyth–Edwards Notation (1) of the position, which is just the standard notation used to efficiently represent the contents of a chess position. The generated string can then be used to synthesize a 2D representation of the position using open-source software such as the Fen2Image service offered by ChessVision (22). This process is summarized in Figure 9.

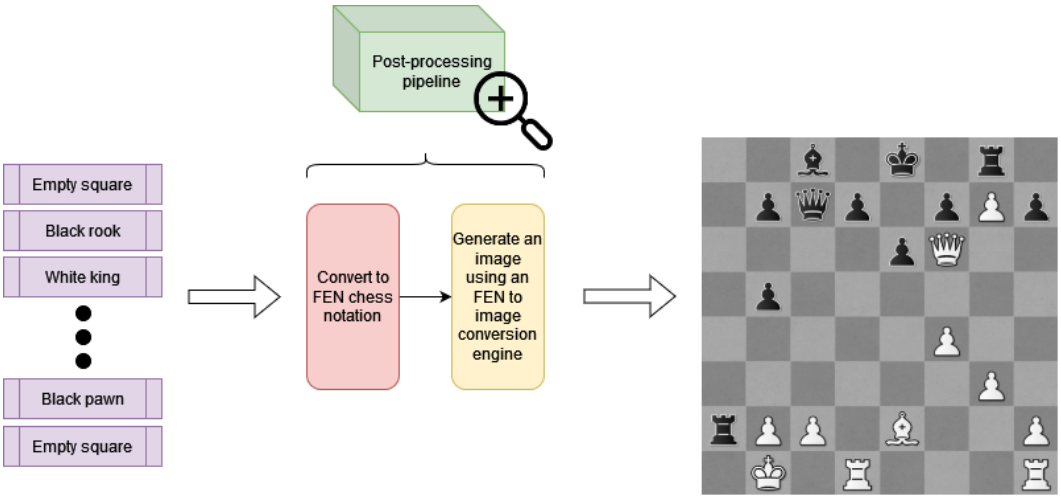


Figure 9: Overview of the post-processing pipeline

2.4 High-level pipeline overview

As mentioned, although model training, validation, and testing were carried out on individual chessboard squares, the pre-processing and post-processing pipelines were designed to be a part of an end-to-end pipeline. The idea is to allow users to upload top-view, uncropped, and unprocessed photographs of their chessboards into the full pipeline and obtain a reconstruction of their chess position from it. Considering the DL model as a black box for now, an overview of the full pipeline is summarized in Figure 10.

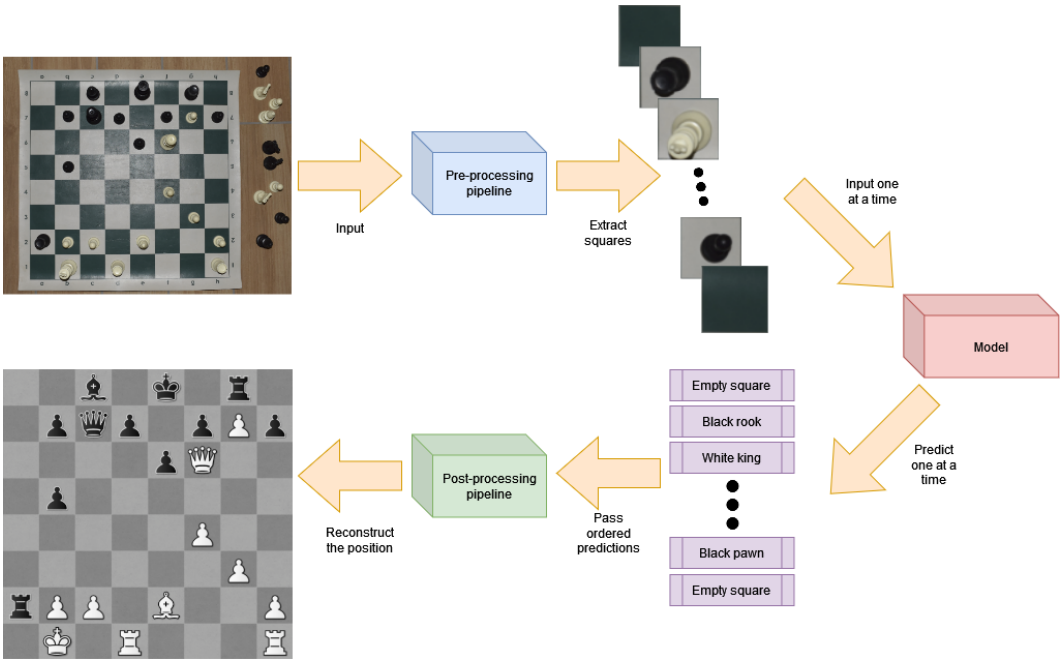


Figure 10: Overview of the full pipeline

2.5 Model

2.5.1 Proposed approach: DANN

2.5.1.1 Architecture

1. Overview

The proposed architecture is based on a 2015 paper from Ganin et Al. (5). At a high-level, it consists firstly of a feature extractor, which is a neural network that outputs a hidden representation of the input image. The output code is then fed to both a classifier which outputs the label of the piece on the input square, and a domain discriminator that attempts to predict whether the input is from the source or target domain. The key idea here is that the feature extractor and domain discriminator are trained following an adversarial approach. The full architecture is presented in Figure 11. Throughout subsequent sections, we will progressively explain each portion.

2. Feature extractor

The feature extractor was based on the VGG16 convolutional neural network (CNN) architecture popularized by Simonyan & Zisserman's seminal 2014 paper (23). Further, the VGG16 network that was used was pre-trained on the ImageNet dataset (3). The model architecture is summarized by Figure 12.

During training, all but the last 3 convolutional layers were frozen, meaning that only the latter were fine-tuned.

3. Classifier

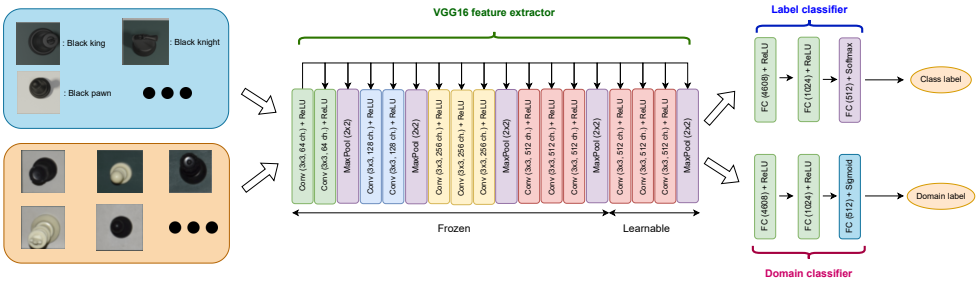


Figure 11: Complete DANN architecture

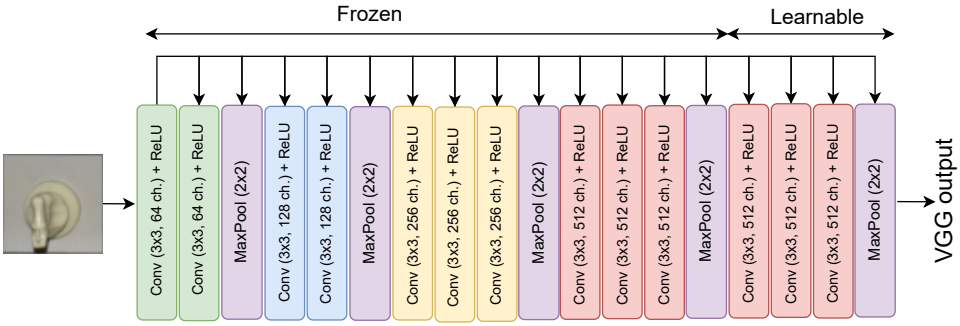


Figure 12: Feature extractor architecture

The classifier portion of the DANN consists of three fully connected linear layers using ReLU activation functions throughout. The final output is then passed to a softmax function which leads to probability predictions, as shown in Figure 13).

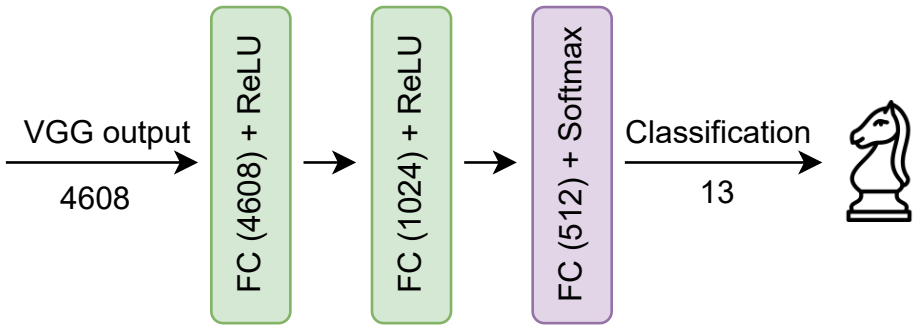


Figure 13: Classifier architecture

4. Domain discriminator

The domain discriminator receives all outputs from the feature extractor and passes them through three fully connected linear layers with progressively decreasing widths, using ReLU activation functions. The outputs of the final layer are then passed through a sigmoid activation function for a binary prediction (see Figure 14).

2.5.1.2 Loss functions

1. Classifier loss

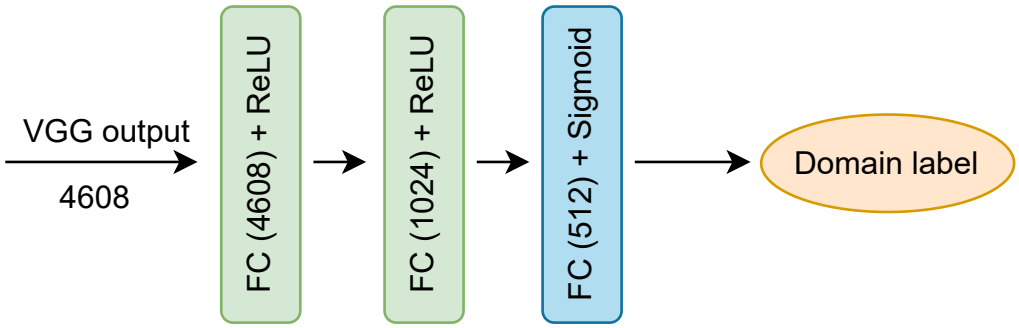


Figure 14: Domain discriminator architecture

The most straightforward and widely used approach to assessing multi-class classification performance is categorical cross-entropy loss (24):

$$\text{CE} = -\log(p_t)$$

where p_t is the predicted probability for the true class. The problem with this function for chess piece classification is that it assigns equal weight to all classes in the total loss calculation. However, some classes are clearly much easier to predict than others, which could give the impression that the classifier is doing well when it is in fact performing poorly. In such cases, a generalized version of cross-entropy called Focal Loss (25) can be used:

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

As p_t becomes larger, meaning that the model is finding it easier to classify the correct class, $(1 - p_t)$ decreases and less weight is assigned to that class. As such, Focal loss assigns less weight to classes that are easier to classify and more weight to classes that are more difficult to classify. Further, as the hyperparameter γ increases, the loss function down-weights classes that are easier to classify incrementally more. This is best illustrated with Figure 15 which is taken from (25):

It can be observed that, for regular cross-entropy, which is just a special case of Focal loss where $\gamma = 0$, an example predicted correctly with probability 0.6 is assigned a loss value of approximately 1, while for $\gamma = 5$ the loss value is practically 0.

2. Discriminator loss

The discriminator's error was calculated using the Binary Cross Entropy (BCE) loss:

$$\text{BCE}(p, y) = -y \log(p) - (1 - y) \log(1 - p)$$

where p is the predicted probability for the positive class and y is the true label.

2.5.1.3 Training overview

1. Forward pass

During a forward pass, minibatches consisting of a 50/50 split of source and target domain samples are passed to the model in succession. After passing all examples through the feature extractor, the resulting hidden representations are passed through both the classifier and the domain discriminator for the source domain, while the codes for the target domain data are only passed to the domain discriminator, since no labels are available for that portion of the minibatch.

2. Backward pass

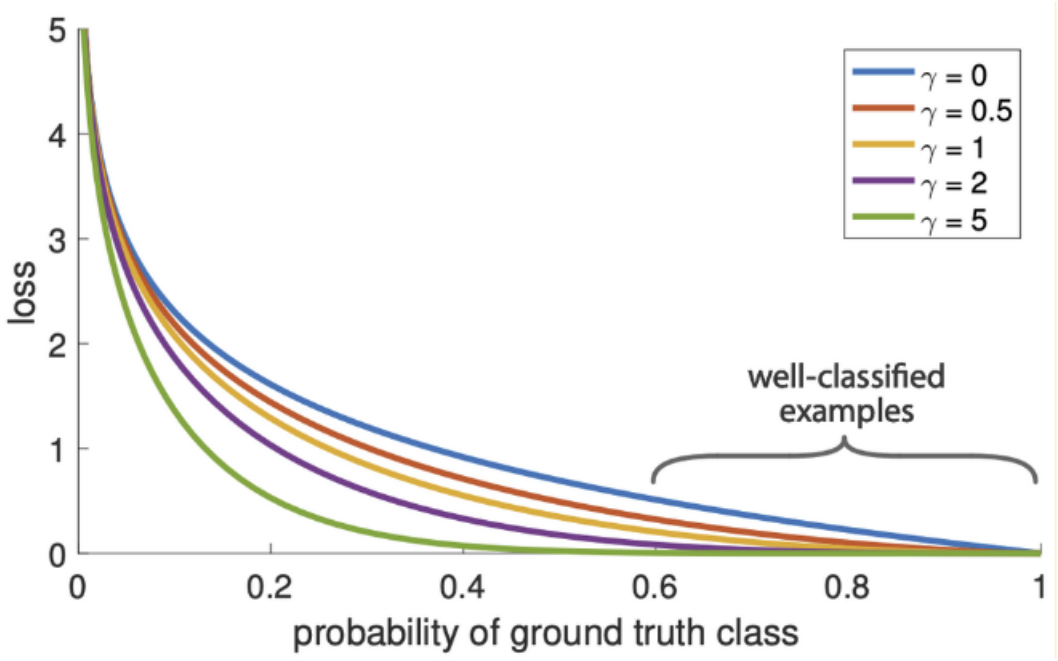


Figure 15: Gamma hyperparameter and its effect on Focal loss

As for any neural network trained through backpropagation, the weights of the classifier and the domain discriminator were updated to optimize the Focal and BCE losses, respectively. The domain adaptation, however, actually occurs through the adversarial training of the feature extractor and domain discriminator. Indeed, assuming L_y is the classification loss, and L_d is the discrimination loss, the feature extractor, instead of optimizing the sum of the two quantities, will optimize their difference:

$$L_f = L_y - \lambda L_d$$

where L_f refers to the loss optimized by the feature extractor, and λ is a hyperparameter that controls the strength of the domain adaptation process. This is summarized in Figure 16. In simple terms,

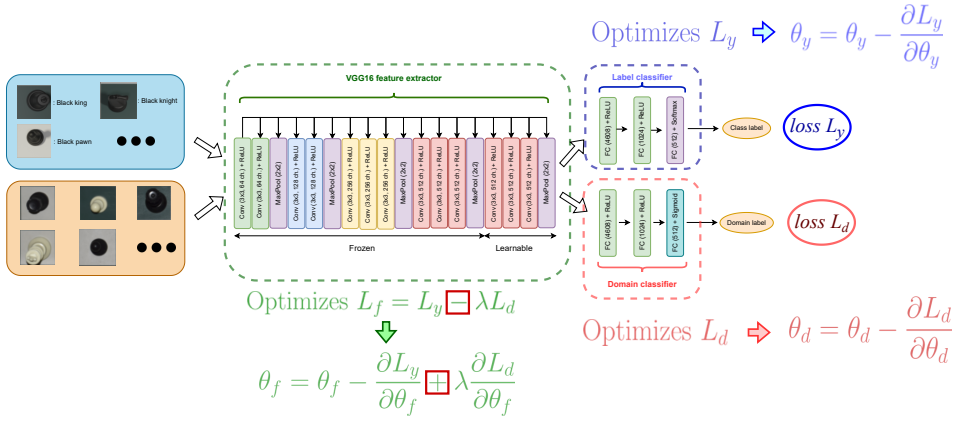


Figure 16: Complete DANN architecture with loss functions

this means that the feature extractor will try to minimize the classification loss while maximizing the discrimination loss. Because the discriminator will attempt to optimize its own weights in order to minimize that same loss, it ensures that training a DANN turns into a min-max optimization problem where the feature extractor is attempting to confuse the domain discriminator, while the latter is trying to differentiate the two domains. Through evolving together, the aim is to reach an equilibrium where

the feature extractor is able to confuse the domain discriminator by outputting similar representations for both domains, making the good classification accuracy on source domain images port well to the target domain.

Finally, implementation of the aforementioned optimization problem was achieved through a gradient reversal layer at the start of the domain discriminator, as recommended by (5). During the forward pass, the gradient reversal layer acts as an identity function. However, during backpropagation, it multiplies the gradient passed through it by $-\lambda$, basically reversing the gradient and applying the domain adaptation factor. This is summarized in Figure 17.

It is to be noted that a regular Stochastic Gradient Descent (SGD) optimizer with a momentum factor of 0.9 (26) was chosen, instead of a more sophisticated approach such as the Adaptative Moment Estimation (ADAM) (27). This is because training an adversarial network is a very fragile process, and thus it is best to avoid interfering with the learning rate in order to facilitate isolating the potential reasons for the training not going in the expected direction.

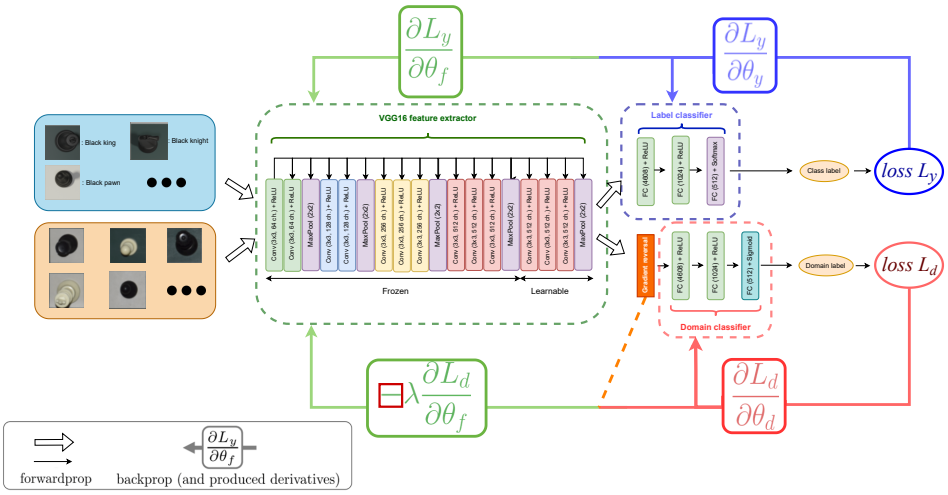


Figure 17: Backpropagation phase for the DANN model

3. Metrics tracked

During training, the classifier and discriminator losses were tracked through epochs. In addition, the classification performance with respect to source domain data was monitored using standard classification accuracies and weighted F1 scores. Because of the oversampling applied to the data, classes were balanced, and thus standard accuracy shown below is a viable option:

$$Accuracy = \frac{\#\text{correctly classified inputs}}{\#\text{inputs}} = \frac{TP + TN}{TP + TN + FP + FN}$$

where TP, TN, FP, and FN are the respective numbers for true positives, true negatives, false positives, and false negatives classified.

Further, the F1 score is the harmonic mean of precision and recall, where precision is the proportion of true positive predictions among all predicted positive instances, and recall is the proportion of true positive predictions among all actual positive instances (28):

$$F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

The weighted F1 score metric additionally computes a weighted average of the F1 scores across all classes, where the weights defined below as w_i represent the number of instances of class i, and $F1_i$ is the score of class i.

$$F_1^w = \frac{\sum_{i=1}^{n} w_i \cdot F1_i}{\sum_{i=1}^{n} w_i}$$

2.5.1.4 Validation

1. Metrics tracked

During validation, the labels that were kept for the target domain's validation set were utilized and the model's Focal losses, Binary Cross Entropy losses, standard classification accuracies, and weighted F1 scores were tracked for both source and target domain data. This is in contrast to training, where these metrics were only tracked for the source domain.

2. Hyperparameter tuning strategy

The two hyperparameters which were tuned during training were the learning rate and the λ domain adaptation factor.

The learning rate values attempted were 0.01 and 0.0075.

As for the domain adaptation factor λ, the values attempted were 0.1, 0.2, as well as a dynamic range of values whose calculation was derived as follows:

$$\lambda = \frac{2}{1 + e^{-\gamma p}} - 1$$

Where p represents the training progress linearly changing from 0 to 1, and γ is a scaling factor fixed to a value of 10, as suggested by (5). This dynamic range of λ values is denoted as *variablepaper* in the hyperparameter tuning tabular results in Section 3.1.1.

For the Focal loss function, the hyperparameter γ was fixed to a value of 2 in order to avoid penalizing the classes that are easy to predict too heavily, while still being able to assign the classes that are harder to predict a larger portion of the loss function.

2.5.2 Alternative baseline 1: Base model

2.5.2.1 Architecture

In order to benchmark the performance of the DANN, the same VGG16 architecture described in Section 2.5.1.1 was utilized for the feature extractor. This was followed by three fully connected layers, interleaved with dropout layers to avoid overfitting, leading up to the classification. This VGG16 was also pre-trained on ImageNet, with the difference being that all of its layers were frozen, and only the fully connected portion was fine-tuned. This is summarized in Figure 18.

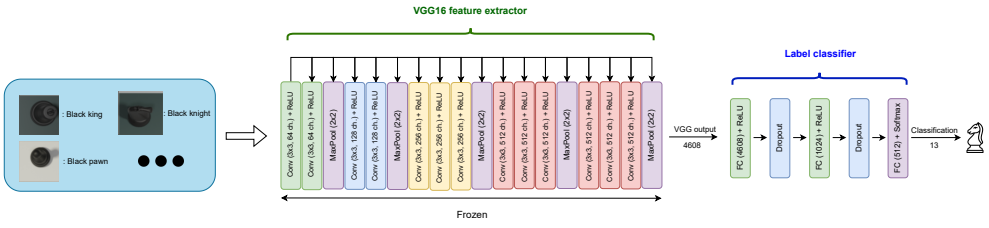


Figure 18: Full architecture of the Base model

2.5.2.2 Loss function

The Base model utilized the same Focal loss used to assess the classification error of the DANN.

2.5.2.3 Training overview

1. Forward and backward passes

The training process was conducted using only the source domain data, without utilizing any domain adaptation whatsoever. The backpropagation update formulas were applied using an ADAM (27) optimizer.

2. Metrics tracked

The metrics tracked during training of the Base model were Focal loss, standard accuracy and weighted F1 scores for the source domain data.

2.5.2.4 Validation

1. Metrics tracked

During validation, the same metrics used during training were tracked, but this time for both the source and target domains, since labels for the validation subset of the target data were now available.

2. Hyperparameter tuning strategy

While fine-tuning the Base model, the majority of the hyperparameters were fixed. Only the influence of training set batch size, Focal loss γ values and dropout rates were assessed. Batch sizes of 100 and 200 were attempted in order to determine the optimal rate of stochasticity. Similarly, Focal loss γ was set to both 0.2 and 0.5 sequentially so as to determine the optimal strength of the down-weighting of easy examples. Finally, dropout rates of 0.2 and 0.5 were attempted. The learning rate was fixed to be 0.001 because slow training was not an issue, especially given that the Base models were trained for a single epoch.

2.5.3 Alternative baseline 2: CORAL model

2.5.3.1 Architecture

The CORAL model has the same architecture as the Base model, except for 3 key differences:

- The dropout layers of the classifier portion are discarded.
- The last 3 convolutional layers of the VGG16 are made learnable.
- Another loss function, the correlation alignment loss (CORAL), is computed at the first and last linear layers of the classifier

The reason for the first change is to make the classifier portion of the CORAL model be the same as that of the DANN model, rendering the comparison equitable. The reason for the second & third changes will become apparent in the training section. The architectural changes applied are shown in Figure 19. It is to be noted that the intuition behind λ will become apparent in the training section.

2.5.3.2 Loss functions

The loss functions utilized by the CORAL model during training included Focal loss for the label classification, as well as CORAL loss, which aims to minimize the domain shift between two distributions by aligning the second-order statistics of their features (4). It computes the covariance matrix of the source and target features and then minimizes the Frobenius norm between the difference of the two covariance matrices. This way, the correlation between the features is preserved and domain shift is reduced. It can be obtained using:

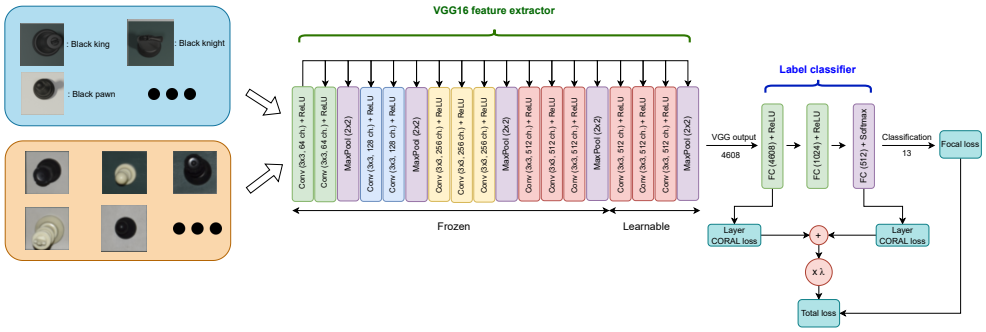


Figure 19: Complete CORAL model architecture

$$L_{\text{CORAL}} = \frac{1}{4d^2}\|C_s - C_t\|_F^2$$

Where d refers to the dimensionality of the feature space, $4d^2$ is a scalar constant that scales the loss function to make the magnitude more interpretable, C_S and C_T are the covariance matrices of the source and target data, and $|\cdot|_F^2$ denotes the squared Frobenius norm. The covariance matrices of the source and target data are specifically given by:

$$C_S = \frac{1}{n_S - 1}(X_S - \bar{X}_S)(X_S - \bar{X}_S)^T$$

$$C_T = \frac{1}{n_T - 1}(X_T - \bar{X}_T)(X_T - \bar{X}_T)^T$$

Where n_S and n_T denote the number of samples in the source and target domain, X_S and X_T denote the data matrices for the source and target domain, and the terms $\bar{X}_S$ and $\bar{X}_T$ represent the column-wise mean of X_S and X_T, respectively.

2.5.3.3 Training overview

1. Forward and backward passes

During a forward pass, minibatches consisting of a 50/50 split of both domains were passed to the model. The source domain examples are the only portion that reach the classification stage, while both domains are equally used to compute the CORAL losses at the first and last linear layers of the classifier. The total loss of the model can then be computed as:

$$TL = FL + \lambda(L_{\text{CORAL}})$$

where FL represents the Focal loss, and where increasing the value of λ assigns more weight to the CORAL loss value, as was the case for the domain adaptation factor of the DANN model. The challenge is to find a value of λ that is large enough so that sufficient domain adaptation occurs in order to predict the target domain accurately, but not too large so that the classifier is neglected by the optimizer, leading to poor accuracies in both domains.

The hope is that, through minimizing the CORAL loss, the hidden representations of the two domains will become similar enough so that predictions that are accurate for the source domain will work comparably well for the target domain.

After the losses are computed, backpropagation is performed using an ADAM optimizer, just as for the Base model.

2. Metrics tracked

During training, various metrics were tracked, including Focal loss, standard accuracy, and weighted F1 scores, all for the source domain data. Further, the model's training performance was also assessed based on its total CORAL loss (Sum of the two CORAL losses computed), as well as the total loss, as outlined in the preceding subsection.

2.5.3.4 Validation

1. Metrics tracked

During validation, Focal losses, standard classification accuracies, and weighted F1 scores were monitored for both source and target domain data, making use of the labelled validation subset of the target data. The model's validation performance was also assessed based on its aggregated layer CORAL loss as well as its total loss, with the formula for the latter using the Focal loss of the source domain data only (not the sum of the Focal losses of both domains) in order to stay on the same scale as the training total loss.

2. Hyperparameter tuning strategy

For the Focal loss function, the hyperparameter γ was fixed to a value of 2. The two hyperparameters which were tuned during training were the learning rate, with attempted values of 0.001 and 0.01, as well as the domain adaptation factor λ.

The classifier was given a head start before gradually increasing the degree of domain adaptation by progressively incrementing λ from 0 to λ_{max} throughout epochs, using the formula:

$$\lambda = p \times \lambda_{\text{max}}$$

Where p indicates the linear progress of the training going from 0 to 1, and the λ_{max} values of 0.01, 0.1, 1, 10 and 100 were attempted.

As indicated in the next section, many combinations of only these two hyperparameters were tested because their interplay led to drastically different performance rates (see Section 3.3.1).

3 Results

3.1 DANN

3.1.1 Hyperparameter tuning results

The hyperparameter tuning for the DANN model was done with the following fixed parameters: γ (Focal loss) $= 2$; $Batch\ Size = 100$; $Number\ of\ epochs = 30$. The best validation accuracy value on the target dataset was 84.1%, obtained for λ (DA factor) $= 0.2$ and $Learning\ rate = 0.01$, as shown in Figure 20.

As a reminder, the *variablepaper* value for λ refers to the suggestion by the DANN paper (5) that makes λ increase from 0 to 1 throughout the training, as explained in section 2.5.1.4.

3.1.2 Validation curves

The validation metrics mentioned in Section 2.5.1.4 corresponding to the hyperparameters that yielded the best validation accuracy (as shown in 20) are depicted in Figure 21.

The discriminator loss plots of both domains in Figure 21a show oscillations that, except for the first few iterations, gradually increase, converging to around 0.5 These oscillations are a reflection of the dynamic competition between the feature extractor and the domain classifier. Indeed, while the discriminator aimed to accurately classify the domain of the input data, the feature extractor was consistently improving to generate hidden feature representations that are domain-invariant, thus making it difficult for the discriminator to accurately classify the domain. The challenge resided in

Lambda (Domain adaptation)	Learning Rate	Final validation accuracy
0.1	0.075	74.0%
0.1	0.01	75.9%
variablepaper	0.075	80.5%
variablepaper	0.01	82.8%
0.2	0.0075	80.0%
0.2	**0.01**	**84.1%**

Figure 20: Hyperparameter tuning results for the DANN model

striking a balance between the discriminator learning too fast for the feature extractor when λ was too big, or the feature extractor out-learning the discriminator when λ was too small. It is to be noted that, while the feature extractor should aim to fool the discriminator, the discriminator needs to be performing well to begin with. Otherwise, there is no point in confusing a discriminator that consists of only random weights.

In an ideal scenario, the discriminator would become totally random despite its attempts to correct its predictions. The discriminator of the DANN model in this project converged at a BCE loss of around 0.7. Assuming that p_t is the probability predicted for the true class, this means that:

$$\text{BCE} = 0.7$$

$$-\log(p_t) = 0.7$$

$$p_t = 2^{-0.7} \simeq 0.6$$

which is very close to the desired 0.5. This shows that the feature extractor nearly reached its full domain-invariance potential. On the other hand, the classifier losses for both domains depicted on Figure 21b showed a decrease over iterations. This indicates that, in addition to the feature extractor generating domain-invariant features, the classifier also effectively learned to classify properly, thereby proving that λ was not too big. As a result, the validation accuracies as well as the weighted F1-scores for both domains shown in Figures 21c and 21d increased as iterations progressed.

3.2 Base model

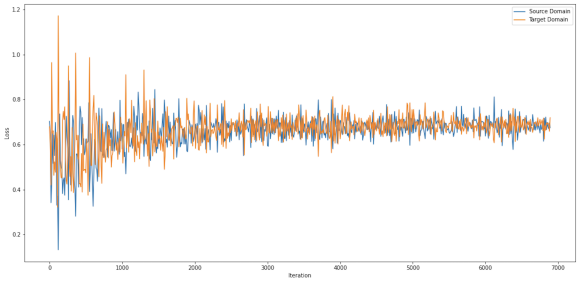
3.2.1 Hyperparameter tuning results

The results of the fine-tuning of the Base model are shown in Figure 22. This hyperparameter tuning process was executed using the following fixed parameters: $Learning\ rate = 0.001$ and $Number\ of\ epochs = 1$. The best validation accuracy value on the target dataset was 56.81%, found with $\gamma = 5.0$, $Batch\ Size = 100$ and $Dropout\ rate = 0.5$.

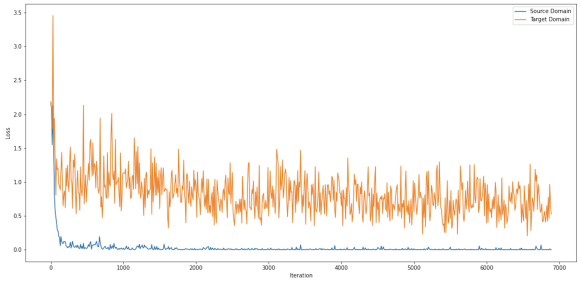
3.2.2 Validation curves

Similarly to the DANN model, the metrics from section 2.5.2.4 were plotted for the combination of hyperparameters that performed best on the validation set. They can be found in the form of a table in Figure 22. The corresponding plots are depicted in Figure 23.

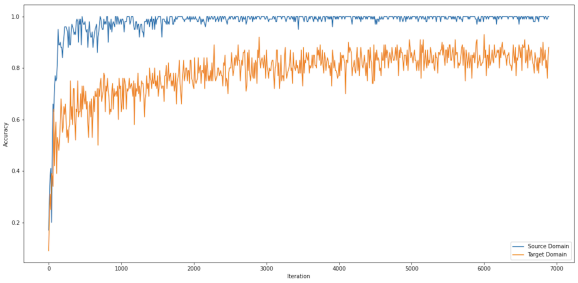
During the training process, the validation loss on the target domain did not show any significant decrease and instead oscillated, while the validation error on the source domain decreased to nearly 0. This indicates that the model struggled to adapt to the target domain, and proves the need for domain adaptation to solve the problem at hand.



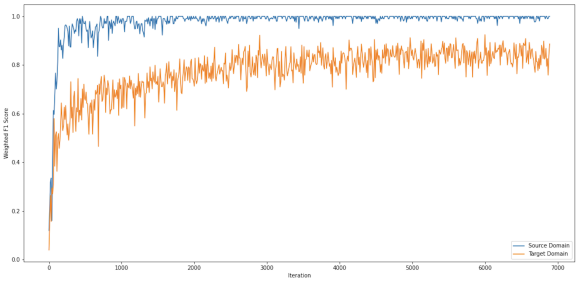
(a) Validation Discriminator Losses



(b) Validation Classifier Losses



(c) Validation Accuracies



(d) Validation weighted F1-score

Figure 21: Final DANN Model validation curves

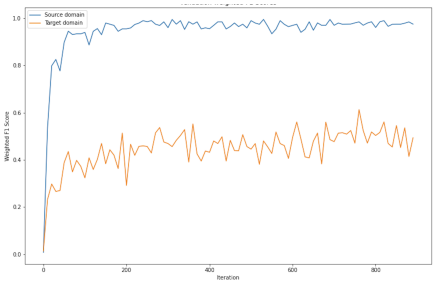
3.3 CORAL model

3.3.1 Hyperparameter tuning results

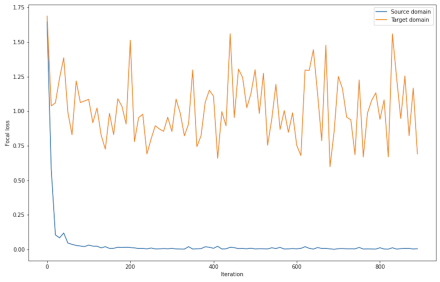
The results of the fine-tuning of the CORAL model are shown in Figure 24. This hyperparameter tuning was done with the following fixed parameters: $Batch\ size = 200$; $Number\ of\ epochs = 20$; γ (Focal Loss) $= 2$. The best validation accuracy value on the target dataset was 85.25%, found with $Learning\ rate = 0.001$ and λ_{max} (coral) $= 10.0$.

Gamma	Batch size	Dropout rate	Final validation accuracy
2.0	200.0	0.2	52.82%
2.0	100.0	0.2	56.67%
2.0	200.0	0.5	49.26%
2.0	100.0	0.5	51.76%
5.0	200.0	0.2	52.81%
5.0	100.0	0.2	46.53%
5.0	200.0	0.5	51.2%
5.0	**100.0**	**0.5**	**56.81%**

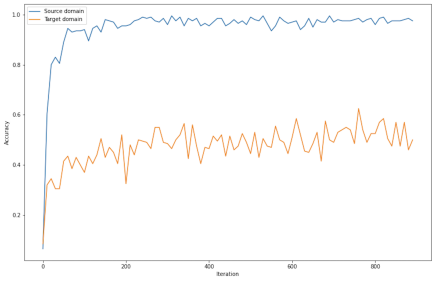
Figure 22: Hyperparameter tuning results for the Base model



(a) Validation weighted F1 score



(b) Validation Losses



(c) Validation Accuracies

Figure 23: Final Base Model validation curves.

As a reminder, λ_{max} is the maximum value of the domain adaptation factor reached through a linear ascent starting at 0, as explained in Section 2.5.3.4.

The drastic difference in validation accuracies between the top and bottom 4 rows of the table can clearly be attributed to the learning rate, showing that 0.01 was overshooting. Furthermore, looking solely at the top 4 rows, we can see that the λ_{max} values of 0.01 and 100 were too small and too large respectively, while 10 was the domain adaptation factor value that struck the best balance between minimizing the domain divergence and learning to classify the pieces to a sufficient degree.

Learning rate	Lambda max coral	Final validation accuracy
0.001	0.01	75.04%
0.001	0.1	83.15%
0.001	1.0	81.73%
0.001	**10.0**	**85.25%**
0.001	100.0	71.69%
0.01	0.01	9.35%
0.01	0.1	59.97%
0.01	1.0	39.52%
0.01	10.0	27.27%
0.01	100.0	7.76%

Figure 24: Hyperparameter tuning results for the CORAL model.

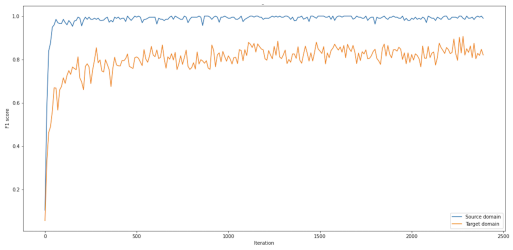
3.3.2 Validation curves

Similarly as the previous models, during the training of the CORAL model, the validation metrics in 2.5.3.4 were monitored thoughout iterations. Figure 25 shows these metrics for the hyperparameter combination with the best validation accuracy that can be found in Figure 24.

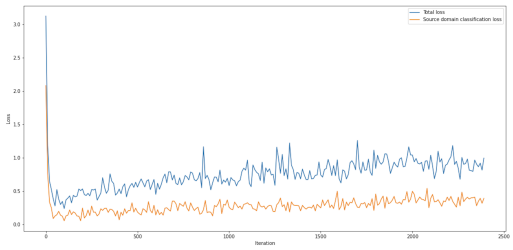
As the regularization parameter λ increased throughout iterations from 0 to λ_{max}, we observe that the CORAL loss showed a significant decrease in value (see Figure 25e). This can be attributed to the fact that as λ increases, the weight of the CORAL loss in the total loss becomes more prominent, leading to a more effective alignment of the domain features. Because a larger λ places more emphasis on aligning the feature distributions across domains, which may result in a higher overall loss value, the total loss increased with the number of iterations as seen in 25b. However, this increase in loss is not an indicator of overfitting of any kind, and rather a natural result of increasing the value of λ throughout iterations, which modifies the scale of the total loss. Further, the classification loss for the source domain increased throughout iterations, while the target domain classification loss remained roughly constant (see Figure 25d). This can be attributed to the fact that, as λ becomes larger, the classification loss is assigned a smaller portion of the total loss, causing the backpropagation process to focus less on optimizing classification accuracy and more on optimizing the CORAL loss. However, it is important to note that, even though the source domain classification loss increased, this is totally normal for an incremental domain adaptation approach, since as epochs progress, the aim becomes less and less to classify the source data, and the focus shifts to the target data. This is justified by the fact that the target domain accuracies and F1 scores consistently improved (See Figure 25a and 25c). This indicates that the domain alignment achieved through the CORAL loss was very beneficial for improving the model's performance on the target data, despite the increases in the source domain losses.

3.4 Cross-model metric comparison

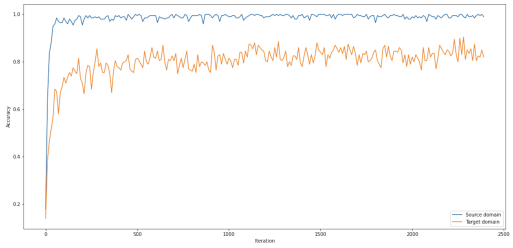
For this section, hyperparameter combinations that yielded the best validation accuracies were utilized for each of the models. Then, multiple metrics were accumulated using the labelled target and source domain testing sets. However, it is to be noted that all reported metrics were achieved on the **oversampled and augmented** testing sets such that all classes are balanced. However, in a realistic chess classification setting, the proportion of empty squares would be much larger than any other class, which means that the accuracy that the proposed model would achieve in a real-life situation would be much larger. In any case, only the balanced dataset accuracies are reported because they provide more insight into the model's performance.



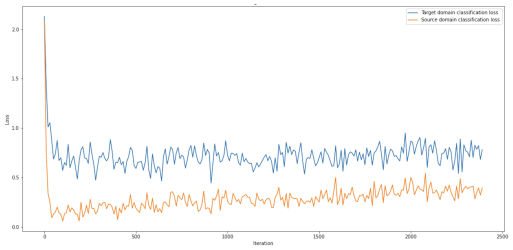
(a) Validation weighted F1 score



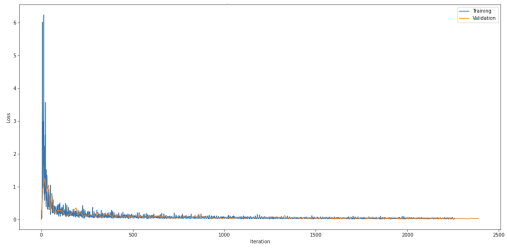
(b) Validation total and source domain classification losses



(c) Validation Accuracies



(d) Validation source and target domain classification losses



(e) Training and validation CORAL losses

Figure 25: Final CORAL Model validation curves.

3.4.1 Testing accuracy

The results obtained after testing the three implemented models showed varying levels of accuracy across the two domains. For measuring the testing accuracy, the standard accuracy score was used, because the oversampled datasets are used to compute the metrics, as mentioned.

On the source domain data, the DANN model achieved the highest testing accuracy of 99.94%, followed by the CORAL model with 99.74%, and the Base model with 97.30%. Thus, all three models performed exceptionally well, with the DANN model being the most accurate. The first row of table 3 shows the source data final testing accuracies. This metric is interesting, even though it does not really represent the objective of this paper.

More importantly, when it comes to the target domain data, the DANN model achieved the highest accuracy of 83.80%, followed closely by the CORAL model with 83.68%, and the Base model with 61.97%. The superior performance of the DANN and CORAL models compared to the Base model highlights the effectiveness of domain adaptation as a viable solution for addressing domain shift and improving model accuracy in real-life data scenarios. The second row of Table 3 displays the testing accuracies on the target domain data.

Dataset	DANN	CORAL	Base Model
Generated (Source)	**99.94%**	99.74%	97.30%
Real-Life (Target)	**83.80%**	83.68%	61.97%

Table 3: Testing accuracies results

3.4.2 Weighted F1 Score

The Weighted F1-score presented in section 2.5.1.3 was also used as an evaluation metric for measuring model performances. On the source domain data, the DANN model achieved the highest F1-score of 0.999, followed by the CORAL model with 0.997, and the Base model with 0.973. These results indicate excellent performance of all three models on the generated data, with the DANN model exhibiting the best performance again. The first row of Table 4 shows the F1-scores for the source data. However, on the real-life data (target data), the DANN model also outperformed the other models with the highest F1-score of 0.847, followed closely by the CORAL model with 0.834, and the Base model with 0.592. The second row of Table 4 shows the F1-scores for the target data.

Dataset	DANN	CORAL	Base Model
Generated (Source)	**0.999**	0.997	0.973
Real-Life (Target)	**0.847**	0.834	0.592

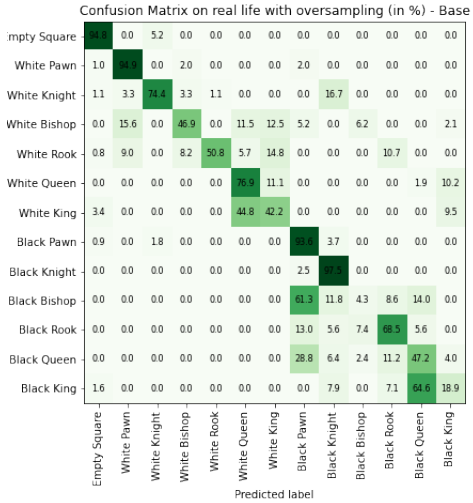
Table 4: Weighted F1-score results

3.4.3 Confusion matrices

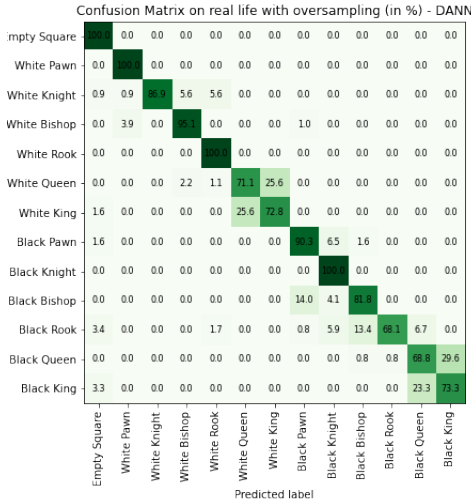
Three confusion matrices were generated based on target domain data, one for each model. It is worth noting that these matrices are normalized by row, so going through each, row by row, gives an idea of what the model classified the row piece as. A perfect model should have a confusion matrix with a value of 100% in every diagonal entry, meaning that each class is predicted as itself.

The Base model's confusion matrix shown in Figure 26a is not very diagonal, indicating that it misclassifies certain pieces. For example, it exhibits notable confusion between chess pieces that have similar top-view appearances. Indeed, it confuses a significant percentage of bishops with pawns, with 61.3% of black bishops predicted as black pawns and 15.6% of white bishops predicted as white pawns. Additionally, it struggles with classifying kings, as 44.8% of white kings are predicted as white queens and 64.5% of black kings are predicted as black queens.

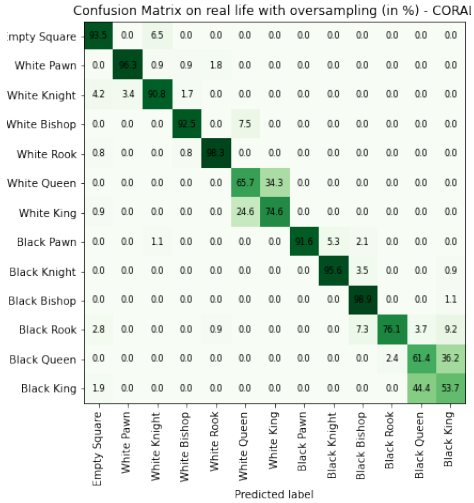
On the other hand, both DANN and CORAL models show much more diagonal confusion matrices, highlighting once again that domain adaptation was the solution to this unsupervised learning problem. The DANN confusion matrix is depicted in Figure 26b, while the CORAL confusion matrix is presented in Figure 26c.



(a) Base Model



(b) DANN



(c) CORAL

Figure 26: Confusion matrices on real life data with oversampling

Interestingly, both models showed some degree of confusion between kings and queens, with the CORAL baseline showing slightly more confusion than the DANN model. This could explain why DANN achieved higher accuracy results compared to CORAL, as it may be making relatively fewer mistakes in classifying kings and queens accurately. In any case, it is clear that these models would perform exceptionally well on a more realistic chess data distribution, since all chess games have 1 king and usually at most 1 queen (except for rare cases where piece promotions are achieved).

3.4.4 Average AUPRC

The average AUPRC measures the overall quality of the model's predictions, taking into account both precision and recall across all possible classification thresholds. A high average AUPRC indicates that the model has a good balance of precision and recall across all possible classification thresholds, and is able to accurately distinguish between positive and negative cases. Conversely, a low average AUPRC indicates poor performance, and suggests that the model is making many incorrect predictions or missing many true positive cases (29).

On the source data, the DANN model achieved the highest average AUPRC score of 0.999, followed by the CORAL model with 0.997, and finally the Base model with 0.975. The first row of Table 5 displays the average AUPRC results on the source data. When it comes to the real-life data, the DANN model again outperformed the other models with an average AUPRC score of 0.856, followed closely by the CORAL model with 0.847, and the Base model with 0.641. The second row of Table 5 displays the Average AUPRC results on the target data.

Dataset	DANN	CORAL	Base Model
Generated (Source)	**0.999**	0.997	0.975
Real-Life (Target)	**0.856**	0.847	0.641

Table 5: Average AUPRC results

4 Concusions and Discussion

4.1 Summary

In conclusion, this paper presents an end-to-end pipeline that utilizes domain adaptation to automate the process of predicting the labels of chess pieces in unlabelled top-view images of chess boards. The pipeline comprises a pre-processing phase, a Deep Learning model for prediction, and a post-processing phase for generating FEN strings and reconstructing the image. Synthetic 3D images of chessboards were used as the source domain data, while unlabelled top-view photographs of chess positions served as the target domain data. Three approaches for optimizing the model were considered, including a pre-trained VGG16 model, an upgraded version with CORAL loss for domain adaptation, and a Domain Adversarial Neural Network (DANN) with adversarial training.

The testing metrics were evaluated for the various models using balanced target domain data, although it should be noted that the oversampling of minority classes does not show the full potential of the models in a real-life scenario. The results showed that the DANN model was the most accurate among the models, with the CORAL model being a close contender. In contrast, the Base model performed poorly in comparison to both. These findings highlight the potential of domain adaptation, for automating the time-consuming process of manual record-keeping in chess without requiring any data labelling.

4.2 Limitations

4.2.1 Limitation to top-view images

The end-to-end pipeline, as discussed in Section 2.2.1, is specifically designed to classify top-view images of the target dataset. However, this approach has certain limitations. Firstly, the pipeline partially tolerates variations in angles. While it can successfully rectify images that are not entirely top-view through image warping, it may struggle with accurately rectifying side-view images. Additionally, the model may encounter challenges in distinguishing certain chess pieces, such as bishops and pawns, which have similar appearances when viewed from the top. To address this issue, future research could involve modifying the warping and square detection algorithms in order to support training on side-view images, which would make the model completely robust to camera angles, as mentioned in Section 2.2.1.5.

4.2.2 Dependence on board texture and piece set

During training, validation, and testing, the models were trained using the same target domain chess set and its corresponding source domain imitation. This means that they inherently lack the ability to perform consistently on different board textures and piece styles. To improve the models for recognition across a range of chess sets in future work, it would be beneficial to attempt an unsupervised domain adaptation task where the source and target domain datasets mix a wide range of board textures and piece styles.

4.3 Advantages

4.3.1 Excellent results through the use of domain adaptation

The results compared in Section 3 clearly show the superiority of a domain adaptation approach when compared to a baseline which utilized no domain adaptation. Further, the confusion matrices from said section showed that most pieces were classified with near perfect accuracy, except for the kings and queens which make up the minority of a real-life distribution. This showcases that it is worth considering an unsupervised domain adaptation approach over labelling the target data and training the model directly on it. Through such an approach, the authors of this project were able to meet all of the constraints mentioned in the introduction, from having the generated data be similar enough to the target domain data, to surpassing the performance of the Base model, thus allowing us to consider the project as successful.

4.3.2 Abundance of training examples without manual labeling

This project highlights that generating data and performing domain adaptation can yield comparable results to labelling and training on the target domain, without the effort required for the labelling. Further, the ease of generating data meant that the total size of the training dataset was much larger than what would have been achieved through manual labelling. On the data generation front, future work could be done to have the source domain match the target domain even more closely, which would likely imply even better results.

4.3.3 Invariance to rotation, translation, and lighting

As discussed in Section 2.1.2.2, due to the authors having full control over the source domain data, different lighting conditions, top-view camera angles, translations, and rotations were applied to the generated examples. This shows another advantage of data generation, which is the ability to make the trained models invariant to different conditions as required.

4.4 Future work

4.4.1 Other domain adaptation approaches

A reconstruction-based approach to domain adaptation, such as a Deep Reconstruction-Classification Network (DRCN) (30) could be worth exploring. Such an approach would entail reconstructing the target domain data as the means to achieve domain adaptation.

Another divergence-based approach that could be attempted is the Wasserstein Distance Guided Representation Learning Model (WDGRL) (31), which is a middle ground between the CORAL and DANN models presented in this paper. Essentially, it replaces the domain discriminator portion of the DANN with a domain critic that uses learnable weights to learn the Wasserstein distance between domains. Results presented in the cited paper show a clear improvement over DANN, suggesting that this method is worth investigating.

4.4.2 Further hyperparameter exploration

Due to limited time and resources, hyperparameter tuning was not extensively explored in this project. It is possible that the models could have achieved even better performance, had a wider range of hyperparameter combinations been validated. For example, the DANN model was only able to confuse the discriminator to an accuracy level of 0.6 (See Section 3.1.2), something that could have been improved by specifically exploring slightly larger λ values.

4.4.3 Regularization for the domain adaptation models

Regularization techniques such as weight decay, dropout, and early stopping could be further explored for the two domain adaptation models.

4.4.4 Live video annotation

Another potential extension of this research could be processing entire videos to identify various chess positions within them, subsequently utilizing the presented model to analyze each position

sequentially. This approach would even further advance the goal of achieving streamlined and readily available automatic chess labeling.

4.4.5 Utilizing the rules of chess as constraints

One final thing that could improve the performance of the utilized model is to incorporate the rules of chess as constraints (32). For example, the model could be forced to predict exactly a single king per position during testing, or prohibited from predicting a pawn on the first and eighth ranks, as that is not possible in a real chess position. This could be implemented by adding a soft constraint as a penalty term to the loss function, or even a hard constraint (33) incorporated within the network's architecture. In any case, as mentioned in Section 2.1.2.2, the generated data, when viewed position by position, fully adheres to every rule of chess. Therefore, anyone willing to attempt such an approach would be able to re-use this project's generated full position dataset before it was cropped into individual squares.

References

[1] Wikipedia contributors, "Forsyth–edwards notation — Wikipedia, the free encyclopedia," 2023, [Online; accessed 11-April-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Forsyth%E2%80%93Edwards_Notation&oldid=1130931937

[2] A. de Sá Delgado Neto and R. Mendes Campello, "Chess position identification using pieces classification based on synthetic images generation and deep neural network fine-tuning," in *2019 21st Symposium on Virtual and Augmented Reality (SVR)*, 2019, pp. 152–160.

[3] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[4] B. Sun, J. Feng, and K. Saenko, "Correlation alignment for unsupervised domain adaptation," 2016. [Online]. Available: https://arxiv.org/abs/1612.01939

[5] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, and V. Lempitsky, "Domain-adversarial training of neural networks," 2015. [Online]. Available: https://arxiv.org/abs/1505.07818

[6] S. Ben-David, J. Blitzer, K. Crammer, A. Kulesza, F. Pereira, and J. W. Vaughan, "A theory of learning from different domains," *Machine learning*, vol. 79, pp. 151–175, 2010.

[7] Blender Foundation, "Blender," 1998–*present*. [Online]. Available: https://www.blender.org/

[8] S. Ryan, "Chess dataset," https://github.com/samryan18/chess-dataset, 2021.

[9] TidusMMM, "3d chessboard model," https://www.turbosquid.com/3d-models/3d-chess-chessboard-board-model-1612338, 2020.

[10] L. B., "Chessset free 3d model," https://www.cgtrader.com/free-3d-models/sports/toy/chessset, 2016.

[11] "", "Chess bishop 3d model," https://open3dmodel.com/3d-models/chess-bishop_493566.html, "".

[12] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[13] Adobe Inc., "Adobe photoshop." [Online]. Available: https://www.adobe.com/products/photoshop.html

[14] S. Ryan, M. Venkateswaran, M. Deng, and K. Convey, "Chess-ray-vision/chess ray vision.pdf at master · samryan18 ... - github." [Online]. Available: https://github.com/samryan18/chess-ray-vision/blob/master/Chess%20Ray%20Vision.pdf

[15] Elucidation, "Chessboard detection," https://github.com/Elucidation/ChessboardDetect, 2018.

[16] O. Vincent and O. Folorunso, "A descriptive algorithm for sobel image edge detection," 01 2009.

[17] J. Canny, "A computational approach to edge detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. PAMI-8, pp. 679 – 698, 12 1986.

[18] F. Rivest, P. Soille, and S. Beucher, "Morphological gradients," *J. Electronic Imaging*, vol. 2, pp. 326–336, 01 1993.

[19] M. Visvalingam and D. Whyatt, "The douglas-peucker algorithm for line simplification: Re-evaluation through visualization," *Computer Graphics Forum*, vol. 9, pp. 213 – 225, 10 2007.

[20] E. Dubrofsky, "Homography estimation," *Diplomová práce. Vancouver: Univerzita Britské Kolumbie*, vol. 5, 2009.

[21] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[22] "Fen to image." [Online]. Available: https://chessvision.ai/docs/tools/fen2image/

[23] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[24] Z. Zhang and M. R. Sabuncu, "Generalized cross entropy loss for training deep neural networks with noisy labels," *CoRR*, vol. abs/1805.07836, 2018. [Online]. Available: http://arxiv.org/abs/1805.07836

[25] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," *arXiv preprint arXiv:1708.02002*, 2017.

[26] Y. Liu, Y. Gao, and W. Yin, "An improved analysis of stochastic gradient descent with momentum," *arXiv: Optimization and Control*, 2020.

[27] M. Alom, "Adam optimization algorithm," 06 2021.

[28] D. M. Powers, "Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation," *arXiv preprint arXiv:2010.16061*, 2020.

[29] P. Flach and M. Kull, "Precision-recall-gain curves: Pr analysis done right," in *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28. Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2015/file/33e8075e9970de0cfea955afd4644bb2-Paper.pdf

[30] M. Ghifary, W. B. Kleijn, M. Zhang, D. Balduzzi, and W. Li, "Deep reconstruction-classification networks for unsupervised domain adaptation," 2016. [Online]. Available: https://arxiv.org/abs/1607.03516

[31] J. Shen, Y. Qu, W. Zhang, and Y. Yu, "Wasserstein distance guided representation learning for domain adaptation," 2018.

[32] R. Stewart and S. Ermon, "Label-free supervision of neural networks with physics and domain knowledge," 2016. [Online]. Available: https://arxiv.org/abs/1609.05566

[33] P. Márquez-Neila, M. Salzmann, and P. Fua, "Imposing hard constraints on deep networks: Promises and limitations," 2017.