PLAYING CARD CLASSIFICATION USING DEEP LEARNING

Drew Burritt, Jose Cazarin, Bhavyai Gupta, Michael Man Yin Lee, Thomas Scott

University of Calgary

Abstract

Playing card recognition is one practical example of image classification. By utilizing deep learning architecture, playing cards can be classified according to their suit and value, enabling the likes of casinos and game tournaments to either survey games for cheating or provide game analytics, such as real-time chance of winning. This project investigated the potential of deep learning for playing card classification. A card dataset was assembled using group member obtained card images, independent datasets, and augmented card images. This dataset was subsequently tested on five deep learning classification models (VGG16, ResNet 50, EfficientNetB0, EfficientNetB4, and a custom model) using both fully trained models with no preexisting weights and transfer learning from ImageNet weights. Results obtained indicated the validity of deep learning towards playing card classification, with fully trained EfficientNetB4 in particular showing strong ability to classify playing cards. Overall this project indicates how basic deep learning pipelines can be applied to an image classification problem with industry relevance.

Keywords: Machine Learning, Deep Learning, Playing Cards, Object Classification

1. Introduction

Playing card detection can be used in a variety of applications, especially in licensed game establishments where it can assist with detecting cheating and acquisition of general statistical data [1]. In particular deep learning pipelines have enabled casinos and professional card game tournaments to recognize the suit and values of a card. With sufficient accuracy, these organizations can utilize deep learning models for general surveillance, such as detecting signs of improper play through effective object recognition [2]. Moreover any deep learning model with high accuracy in detecting cards can also be an effective input for other models, such as predictive analytics for professional poker whereby the model can recognize the cards played at the table, which then feeds into another model to predict the real-time chance of winning from each player.

The objective of this project was to explore the potential of deep learning towards playing cards recognition. A card dataset was acquired for model training through each group member taking several card pictures and combining them with three Kaggle card datasets [3, 4, 5]; the combined dataset was further augmented using brightness, rotation and shear changes to yield different card pictures for each individual card in the dataset. This dataset was then trained on five different deep learning classifiers (VGG16, ResNet50, EfficientNetB0, EfficientNetB4, and a custom model) via both conventional training with no preexisting weights and (except the custom model) transfer learning to analyze how model choice and model format influences playing card classification [6, 7, 8]. The best performing models obtained were then tested with new card pictures independently acquired outside the dataset to better gauge their performance and robustness and observe the overall effectiveness of deep learning towards this classification problem.

2. Related Work

While a relatively niche area of image classification, playing card classification has been studied and analyzed both in literature and via personal projects. For example, contemporary machine learning has been applied to this problem, with state vector machines and logistic regression both used to classify cards by suit and value [9, 10]. In comparison, deep learning offers greater variation of models and pipelines used. Convolutional neural networks for instance are frequently employed, including YOLO (You Only Look Once) object detection, R-CNN algorithms, and custom convolutional layer architectures [11, 12, 13]. Likewise, transfer learning has seen use, with ResNet and EfficientNet models utilized in both classification and real-time detection scenarios [10, 14]. In particular, real-time detection allows for use of these models in poker tournaments for real-time prediction of winning odds, an example of which can be found in this project's associated Git repository [15].

3. Materials and Methods

3.1 Dataset Creation

Dataset creation began with each group member taking five pictures of each card found in a standard 52 card deck (for a total of 25 images per card). Each member deck was sourced individually with no deck shared among group members. Of these pictures, at least one per card had the card partially covered to reduce image information and potentially mitigate model overfitting. Card images were classified according to their suit and value with 52 classes resulting from the dataset.

To this dataset, three additional datasets were added to further increase the number of card images per class [3, 4, 5]. Moreover, to avoid overfitting from small dataset size, this combined dataset was then run through a Python script generating nine new images for each card image [16]; these new images were made by varying the shear, rotation, and brightness parameters found in the TensorFlow ImageDataGenerator class [17, 18]. All steps combined yielded a final dataset of 38,960 images evenly distributed between 52 classes.

The test dataset was made by randomly sampling 20% of images of each class, to achieve class balance, resulting in 7,794 images in the test set and 31,166 images for training and validation. It should be noted this results in a test dataset with augmented images which is not standard practice, however given the number of non-augmented images was small, it was decided that using augmentation and limiting potential overfitting was better from a model evaluation standpoint.

Two additional test datasets were also created using new images for further model analysis: a “hard” dataset comprised of partially covered cards, cards positioned far from the camera and cards with busy backgrounds; and an “easy” dataset consisting only of rotated and unobstructed cards positioned close to the camera.

3.2 Multilabel problem

One option tested during this project to mitigate overfitting was the use of multi-labeling, whereby the 52 classes were split into two label outputs: value (e.g. 3, King) and suit (e.g. Clubs, Hearts). To achieve this the final dataset was processed using a custom PlayingCardsGenerator class derived from the TensorFlow ImageDataGenerator class, the code of which can be found in this project’s Git repository [15].

3.3 Transfer learning

Four models were utilized for testing the validity of transfer learning in this project: VGG16, ResNet50, EfficientNetB0, and EfficientNetB4. All four models employed the same overall procedure whereby their fully-connected top layers were removed, new top layers inserted, and the model is fitted to the data using the TensorFlow pretrained ImageNet weights [19]. Specifics for each model are as follows:

VGG16: the default VGG16 top layer structure was used consisting of flatten - dense (208 classes) - dense (104 classes) - dropout (20%) - dense (52 classes). Neuron numbers for the first and second dense layers were arbitrarily chosen in order to limit the number of trainable parameters. This model was then tested in two setups varying the number of trainable layers. First setup allowed the last two input layers to be trained (7,605,516 million total trainable parameters) and the second the last three layers (9,965,324 million trainable parameters). Both setups were run using the Adam optimizer with a 0.0001 learning rate and default values for remaining parameters. Input images used the default shape (224, 224, 3). A batch size of 32 was used and both models were trained for up to 50 epochs with early stopping (with patience parameter of 10).

ResNet50: the default ResNet50 top layer structure was used consisting of 2D average pooling - flatten - dense (52 classes). This model was then tested in two setups varying the number of trainable layers. First setup allowed the last five layers to be trained (2,758,708 million total trainable parameters) and the second the last 11 layers (6,169,652 million total trainable parameters). Both setups were run using the Adam optimizer with a 0.001 learning rate and default values for remaining parameters. Input images used the default shape (224, 224, 3). A batch size of 8 was used and both models were trained for up to 100 epochs with early stopping (with patience parameter of 10).

EfficientNet: the default EfficientNetB0 model was tested with three different tops. The first version had flatten and dense layers on top of convolutional layer (~3 million total trainable parameters). The second version had a top consisting of GlobalAveragePooling2D, BatchNormalization, Dropout, Flatten, and Dense layers (~69 thousand total trainable parameters). The third version consisted of GlobalAveragePooling2D, Dropout, and Dense layers (~66 thousands total trainable parameters). Version 3 yielded the high accuracy and was then used with the default EfficientNetB4 model for further testing. Each setup was configured to use Adam optimizer with 0.001 learning rate during initial training and 10^{-8} learning rate during fine tuning (with all input layers unfrozen for fine tuning training). A batch size of 16 was used and each model was trained for 100 epochs during initial training and 50 epochs during fine tuning.

3.4 Training pre-built models from scratch

Four pre-built models were trained from scratch with no initial weights: VGG16, ResNet50, EfficientNetB0, EfficientNetB4. The same procedure was followed for all four models. As these models are already complete networks, the only alteration was setting the final layer output number to 52 (one for each card class). In particular EfficientNetB4 was chosen after obtaining good performance with the B0 version, owing in part to increased number of parameters and suitable balance between model size and accuracy [20].

3.5 Custom model

Using basic deep learning techniques, a model was built for testing multilabel classification. Input image size for this model was set to 256x256. The convolutional section contained 12 convolutional layers, 5 max pooling layers, and 6 dropout layers with dropout set to 50%. The classification section featured one dense layer (17 neurons) connected to two output dense layers, one with 4 neurons (number of suits) and the other with 14 neurons (number of values). Total trainable parameters for this model was 1,459,147. The model was run using the Adam optimizer with default parameters and categorical cross entropy loss, and arranged to train for 1000 epochs with an early stopping callback.

A model version with a single output dense layer with 52 neurons was also tested, but produced poor results compared to the two output version, only yielding 25% accuracy on the test dataset. Likewise "on the fly" augmentation, which would apply new augmentation on top of the already augmented images of the dataset, was tested by specifying augmentation parameters, but did not yield any accuracy increase after the first 10 epochs and so was not used.

4. Results

4.1 VGG16

Two variations of the layers were run for the VGG16. The first model utilizes two outputs as discussed in the multilabel section. This model had a total of 134,330,499 trainable parameters and presented poor performance with no accuracy increase for the first 10 epochs, resulting in it being abandoned. The second model has one output and 52 classes. This model showed average results, achieving 49% validation accuracy. Figures 1 and 2 showcase the accuracy and loss during training.

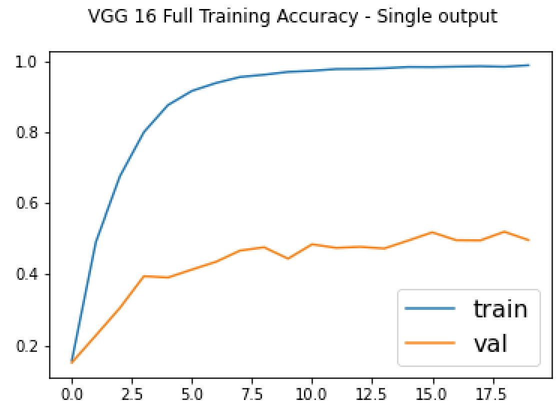

Figure 1: VGG16 accuracy during training, for training and validation datasets.

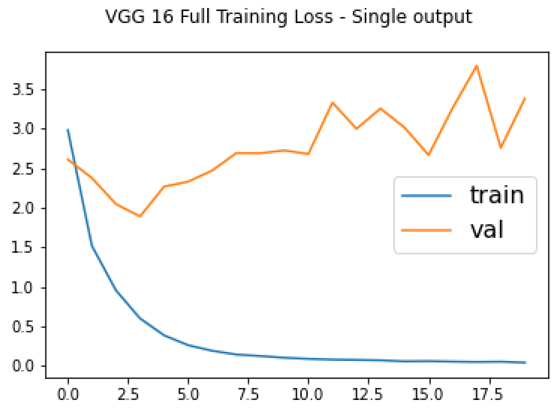

Figure 2: VGG16 loss during training, for training and validation datasets.

4.2 Resnet50

The ResNet50 model has a total of 23,641,140 trainable parameters, 53,120 non-trainable parameters, and 52 classes to coincide with the number of cards in the deck. The graphs for accuracy and loss are shown in Figures 3 and 4.

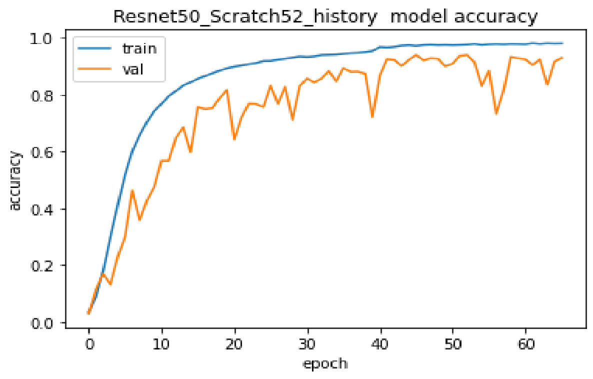

Figure 3: Resnet50 accuracy during training, for training and validation datasets.

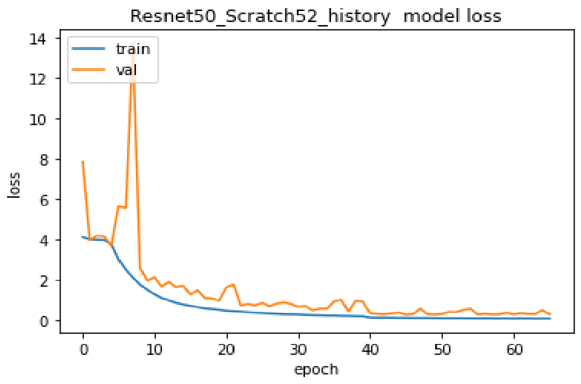

Figure 4: Resnet50 loss during training, for training and validation datasets.

4.3 EfficientNetB0

The EfficientNetB0 model was imported from Tensorflow and used the default image size of 224x224. The only parameters changed were the dropout rate and number of classes, being set to 0.3 and 52 respectively. This gave a model with 4,074,160 trainable parameters. Early stopping with a patience of 20 epochs was used to monitor validation loss with a batch size of 16 and initial learning rate of 0.001 (halved every 30 epochs), resulting in the model being trained for 85 epochs. A multi-label approach was also tried but quickly abandoned due to poorer performance. Figures 5 and 6 indicate model accuracy and loss during training.

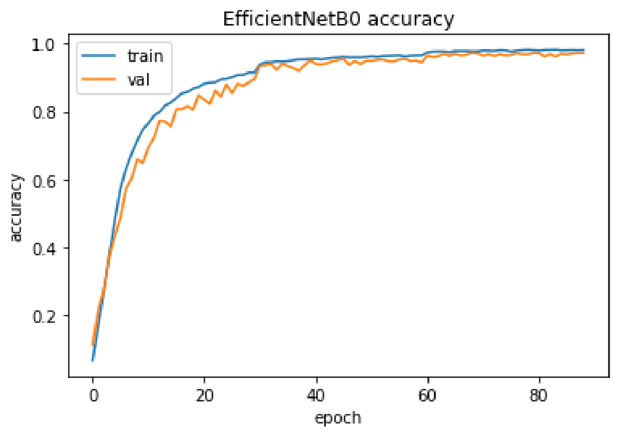

Figure 5: EfficientNetB0 model accuracy during training, for training and validation datasets.

4.4 EfficientNetB4

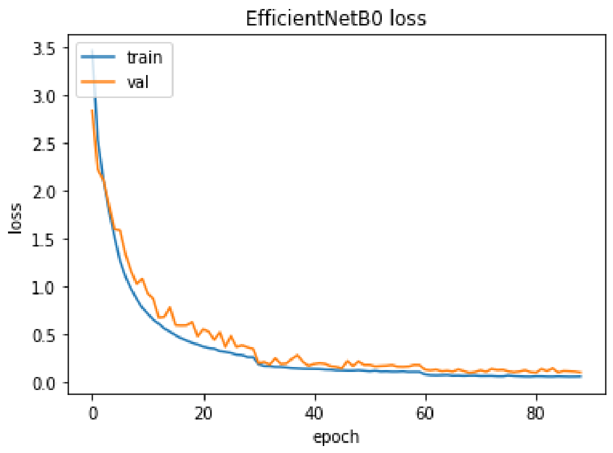

Figure 6: EfficientNetB0 model loss during training, for training and validation datasets.

EfficientNetB4 was trained in the same manner as EfficientNetB0 with the only difference being the larger image size of 380x380. This gives a model with 17,641,852 trainable parameters. The best model was obtained after 95 epochs. Graphs for accuracy and loss are very similar to Figures 5 and 6 for EfficientNetB0.

4.5 Custom model

The best results for the custom model were obtained using a learning rate of 5x10^{-5} , batch size of 64, and different weights applied to the two loss functions of the output layers. The weights took into account the number of different outputs for each layer, with a weight of 4/17 applied to the suits output (4 classes) and a weight of 13/17 applied to the values output (13 classes). Due to the early stop callback, training was stopped at epoch 111. The graphs for the loss and accuracy values of both outputs, for training and validation sets, are shown in Figures 7 and 8.

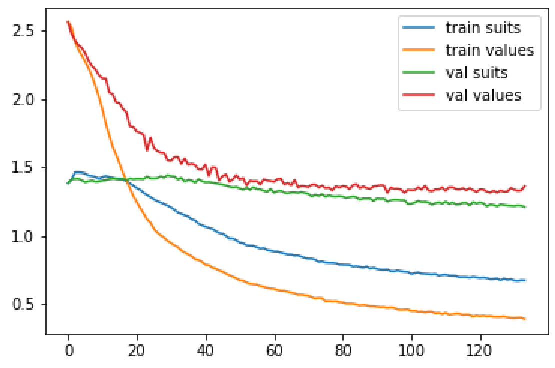

Figure 7: Custom model loss, for both outputs, during training, for training and validation datasets.

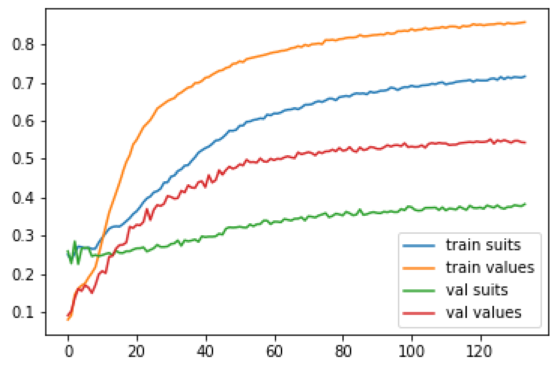

Figure 8: Custom model accuracy, for both outputs, during training, for training and validation datasets.

Table 1: Recorded max accuracy and min loss for VGG16, ResNet50, EfficientNet, and custom models when trained from scratch with no preexisting weights.

	Dataset accuracy (%) / loss				
Model	***Training***	***Validation***	***Test***	***Easy test dataset***	***Hard test dataset***
VGG16	99 / 0.04	50 / 3.38	65 / 2.17	63 / 2.26	7 / 10.59
Resnet50	98 / 0.06	93 / 0.29	95 / 0.27	96 / 0.13	31 / 8.22
EfficientNetB0	97 / 0.06	97 / 0.10	98 / 0.07	100 / 5.95×10^{-4}	56 / 2.41
EfficientNetB4	99 / 0.01	99 / 0.03	99 / 0.03	100 / 6.65×10^{-5}	77 / 1.49
Model created from scratch (two outputs)	78 / 0.53 (suits) 86 / 0.38 (values)	47 / 1.22 (suits) 58 / 1.14 (values)	55 / 1.02 (suits) 56 / 1.74 (values)	31 / 1.61 (suits) 40 / 2.28 (values)	44 / 1.59 (suits) 20 / 3.11 (values)

4.6 Transfer learning

Transfer learning overall yielded worse results overall compared to models trained from scratch. VGG16 for example obtained very high training accuracies as shown in Table 2, however yielded weaker validation and test accuracies with only minor increase as trainable layer number increased. Likewise ResNet50 provided strong predictions during fitting with high accuracies increasing as the number of trainable layers increased, yet also saw lagging validation and test accuracies. ResNet50, however, did see increased validation and test accuracy as the trainable layer number increased. Considering ResNet50 saw the greatest accuracy gain with fewer trainable parameters, further testing with a fully trainable model with no preexisting weights was especially encouraged.

Likewise the results obtained for EfficientNetB0 and B4 were varied, though as per Table 2 general accuracies were middling at best. EfficientNetB0 Version 3 yielded the highest accuracy of each tested B0 model at 69% (51% validation) which led to its top layer architecture being used for the tested EfficientNetB4 model. This proved successful as shown in Table 2 with an accuracy increase to 72% (55% validation), though still relatively poor in comparison to ResNet50 and VGG16.

Table 2: Recorded max accuracy and min loss for ResNet50, VGG16, and EfficientNet transfer learning models. Trainable layers indicate how many input layers were allowed to train during fitting.

		Dataset accuracy (%) / loss		
Model	**Trainable Layers**	***Training***	***Validation***	***Test***
ResNet50	5	78 / 1.65	55 / 3.73	59 / 2.86
ResNet50	11	87 / 0.49	72 / 1.44	75 / 1.25
VGG16	2	97 / 0.12	71 / 0.96	76 / 0.84
VGG16	3	97 / 0.09	79 / 0.73	82 / 0.68
EfficientNet B0 (Version 1)	All	33 / 11.9	22 / 14.9	29 / 13.7
EfficientNet B0 (Version 2)	All	65 / 1.1	50 / 1.47	55 / 1.47
EfficientNet B0 (Version 3)	All	69 / 0.97	51 / 1.42	56 / 1.40
EfficientNet B4	All	72 / 0.84	55 / 1.3	64 / 1.14

5. Discussion

Overall the results obtained from this project show the effectiveness of deep learning towards classifying playing card images. Transfer learning for example highlighted the strength of this approach, with certain models - i.e. ResNet50 - obtaining high accuracies when initially trained with ImageNet weights, but still worse than training the models from scratch. Likewise the middling accuracy obtained for EfficientNet transfer learning was contrary to the results obtained using these models during full model training.

The results obtained from transfer learning are likely in part due to used ImageNet weights. The current ImageNet database contains no playing card images, inviting challenges for any transfer learning. Given playing cards are differentiated by drawings found on card edges and centres, the more generalized identification of varied shapes and colours for ImageNet-trained models can potentially struggle to adapt to this more specific classification requirement.

Although poor overall, the high training accuracies and improving validation accuracies obtained when using transfer learning with ResNet50 gave impetus towards fully training this model on the utilized dataset. As previously indicated in Table 2, the very strong accuracies obtained both in validation and testing corroborated using a fully trained ResNet50 model for this project. Of particular note, however, is the poor accuracy obtained when testing the trained ResNet50 model on the "hard" dataset (95% test versus 31% "hard" test) as indicated in Table 1 above. The potential reason for this can be seen in Figure 9. As both gradcams indicate, ResNet50 focuses on a particular corner of the image. If the corner is clearly visible, it can accurately classify the image as indicated by the top gradcam. If the corner chosen is not readily visible per the bottom gradcam, ResNet50 struggles to properly classify the image. These results suggest that image quality has a noticeable effect on ResNet50, with the model working best when inputted images are largely unobstructed or manipulated.

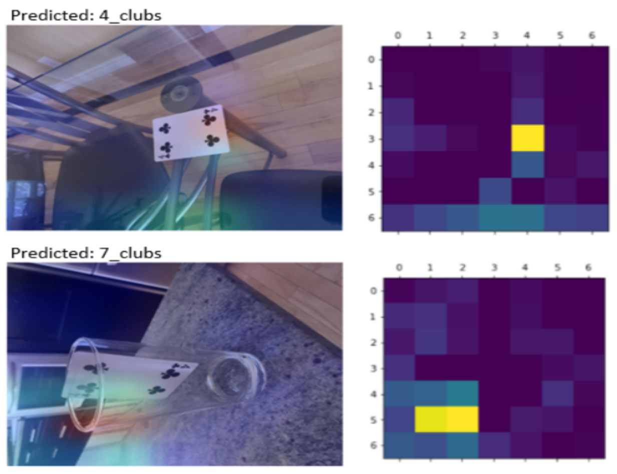

Figure 9: Pass (Top) and Fail Case (Bottom) of Resnet Model for the hard dataset

In comparison, EfficientNetB0 trained with no initial weights performed extremely well on both the test and validation sets, with this model also achieving 56% on the "hard" test set. As this test set was intentionally designed with difficult pictures (e.g. partially obscured, complicated patterns, featuring small portions of the frame) this accuracy is very good. Considering EfficientNetB0 is the simplest of the EfficientNet models, this performance led to EfficientNetB4 also being tested. B4 was selected owing to B5-B7 offering slightly better performance at the cost of significantly larger models; B4 was therefore the optimal balance between performance and computational cost.

Using EfficientNetB4 yielded improvement across training, validation, and test datasets, with 99% accuracy observed on each. In particular, EfficientNetB4 saw significant improvement over B0 on the "hard" dataset, with 76% accuracy achieved. Marginal improvement potentially could be obtained by using the EfficientNetB5-B7 models, however their significantly higher parameter counts may start overfitting on the available dataset given its size (~39000 images) compared to the original ImageNet dataset (1.2 million images) [19,20].

The custom model created used an architecture similar to VGG16 and obtained similar results to the fully trained VGG16 model tested. Both models had the worst performance of all the tested models. As the oldest architectures tested, these models do not contain features present in modern architectures (e.g. skipped connections, batch normalizations, MBConv convolution), potentially limiting the accuracy obtained. Moreover, EfficientNetB0 was obtained using AutoML techniques, something not used in the other architectures considered.

5.1 EfficientNet Gradcam Comparisons

Using the Gradcam technique, activation patterns obtained from EfficientNetB4 and EfficientNet B0 were compared to potentially identify reasons for their high accuracies.

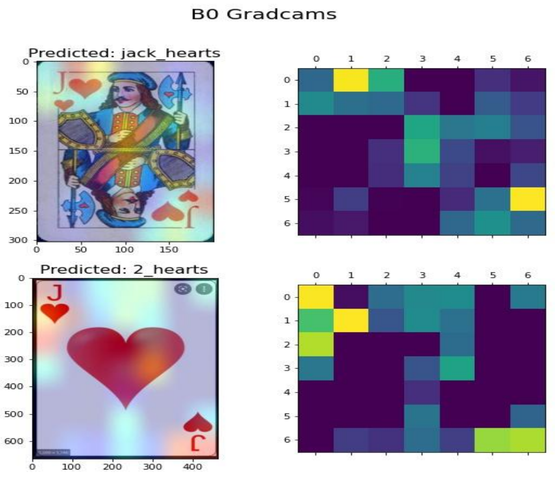

Figure 10: Comparison of B0 gradcams for a standard and non-standard Jack of Hearts card.

As evidenced in Figure 11 it is immediately apparent in the B4 model that few activations occurred in the image's centre compared to the B0 model shown in Figure 10 with more activations in the same area. This is particularly important given that the Jack of Hearts cards used in Figures 10 and 11 were not present in any of the datasets tested [21, 22].

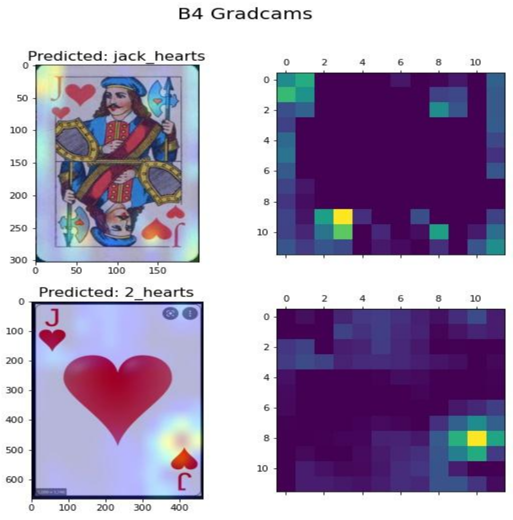

Figure 11: Comparison of B4 gradcams for a standard and non-standard Jack of Hearts card.

It is anticipated that information on the four corners of cards will be used for classification given they contain the majority of card information (e.g. suit, value). This principle extends to face cards as the centre-positioned drawings are often significantly varied and are therefore not desired to be used as information by the model. In this particular case, EfficientNetB4 is the better model as per the Jack of Hearts gradcams in Figure 11 since it relies more on corner information than centre information.

Non-standard cards like the specified Jack of Hearts, however, can fool even EfficientNetB4, as observed in the Figure 11 gradcams. The likely reason for this is the non-standard heart at the centre of the card; while EfficientNetB4 relies more on corner information for classification, it still uses centre information per Figure 11's heatmap, and given a typical Two of Hearts has two hearts at its centre, EfficientNetB4 likely interpreted the non-standard Jack of Heart's centre as having two hearts. This shows that even more discerning models such as EfficientNetB4 can be fooled should the tested card sufficiently diverge from standard playing card layout, and that additional model adjustments are required for any model classifying non-standard cards.

6. Future work

Although this project was successful in classifying playing cards, any significant work on implementing deep learning in casinos would need real-time applications. For example, extraction of cards from a live video feed into a deep learning model can be implemented, enabling players' chance of winning odds to be calculated in real-time. This project lacked the time to create such a system, however the project repository does contain an example application for calculating the chance of winning for a given poker hand [15, 23]. Another example application provided is the classification of playing cards from a live video stream using a Jetson Nano board [24].

Likewise, the models used could be trained again using MixUp Augmentation [25]. This technique enables the creation of new samples in which labels can be blended, which in this case, can be represented by card centres being exchanged, potentially forcing the model to classify based on the edges or corners of the card, considering that even non-standard cards tend to keep the suit and value drawings in that area. Such additional image augmentation could serve to increase model robustness to adversarial examples and therefore enable better classification of non-standard cards.

7. Conclusion

Playing card classification is a small yet important segment of image classification. By successfully identifying playing cards on the basis of suit and value using deep learning models, organizations such as casinos and card game tournaments can detect potential cheating and augment systems such as chance of winning predictors to provide feedback to audiences. This project analyzed the effectiveness of some deep learning models in classifying playing cards. The dataset created for this project was tested on VGG16, ResNet 50, EfficientNetB0, EfficientNetB4, and a custom deep learning model, with both fully trained models with no preexisting weights and transfer learning utilized. The results obtained show that deep learning is applicable to classifying playing cards, and that EfficientNetB4 in particular is an optimal model for classifying cards in a general setting.

References

[1] - Bishop, E. *Machine learning set to transform casino industry by 2030*. Market Business News. Retrieved April 2, 2022, from https://marketbusinessnews.com/machine-learning-transform-casino-industry-2030/251593/

[2] - Dhote, R. *Machine learning and AI in casinos and gaming*. Proche. Retrieved April 2, 2022, from https://www.theproche.com/2021/12/16/machine-learning-and-ai-in-casinos-and-gaming/

[3] - Haldar, G. *Playing cards images - object detection dataset*. Kaggle. Retrieved April 2, 2022, from https://www.kaggle.com/gunhcolab/object-detection-dataset-standard-52card-deck

[4] - Lordloh. *Playing Cards Detection*. GitHub. Retrieved April 2, 2022, from https://github.com/lordloh/playing-cards

[5] - McGuigan, J. *Playing cards*. Kaggle. Retrieved April 2, 2022, from https://www.kaggle.com/datasets/jamesmcguigan/playingcards

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

[7] - He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[8] - Tan, Mingxing, and Quoc Le. "Efficientnet: Rethinking model scaling for convolutional neural networks." International conference on machine learning. PMLR, 2019.

[9] - Ekorudiawan. *Playing-card-classifier*. GitHub. Retrieved April 2, 2022, from https://github.com/ekorudiawan/Playing-Card-Classifier

[10] - Tetelepta, S. *Detecting set cards using transfer learning*. Medium. Retrieved April 2, 2022, from https://towardsdatascience.com/detecting-set-cards-using-transfer-learning-b297dcf3a564

[11] - Redmon, Joseph, et al. "You only look once: Unified, real-time object detection." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[12] - Geaxgx. *playing-card-detection*. GitHub. Retrieved April 2, 2022, from https://github.com/geaxgx/playing-card-detection

[13] - Kaleem, N. *Identifying types of playing cards using an object detection classifier*. Medium. Retrieved April 2, 2022, from https://medium.datadriveninvestor.com/identifying-types-of-playing-cards-using-an-object-detection-classifier-fdd1bae02251

[14] - Chen, Qianmin, et al. Poker Watcher: Playing Card Detection Based on EfficientDet and Sandglass Block." 2020 11th International Conference on Awareness Science and Technology (iCAST). IEEE, 2020.

[15] - ENEL645 Group #1. *Term project for the ENEL645 Course* GitHub. Retrieved April 2, 2022, from https://github.com/ENEL645Group

[16] - Perez, Luis, and Jason Wang. "The effectiveness of data augmentation in image classification using deep learning." arXiv preprint arXiv:1712.04621 (2017).

[17] - Rohlfing-Das, A. *Image Classification for Playing Cards* . Medium. Retrieved April 2, 2022, from https://medium.com/swlh/image-classification-for-playing-cards-26d660f3149e

[18] - *Tensorflow*. TensorFlow. Retrieved April 3, 2022, from https://www.tensorflow.org/

[19] - ImageNet. Retrieved April 2, 2022, from https://www.image-net.org/

[20] - Mingxing, T. Quoc, L. *EfficientNet: Improving accuracy and efficiency through AutoML and model scaling*. Google AI Blog. Retrieved April 2, 2022, from https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html

[21] - *Jack of hearts birth card. free destiny cards reading.* Seven Reflections. Retrieved April 3, 2022, from http://www.sevenreflections.com/cards-of-destiny/cards/jack-of-hearts/

[22] - *File:Jack of hearts.svg*. Wikimedia Commons. Retrieved April 3, 2022, from https://commons.wikimedia.org/wiki/File:Jack_of_hearts.svg

[23] - Souzatharsis. *Texas hold'em odds calculator with support to ranges*. GitHub. Retrieved April 2, 2022, from https://github.com/souzatharsis/holdem_calc

[24] - *Nvidia Jetson Nano for edge AI applications and Education*. NVIDIA. Retrieved April 2, 2022, from https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/

[25] - Zhang, Hongyi, et al. "mixup: Beyond empirical risk minimization." arXiv preprint arXiv:1710.09412 (2017).

Team Member	Contribution	Score
Drew Burritt	Took 5 pictures of every card for the original dataset. Took 241 completely new pictures with difficult backgrounds/partially obscured cards to create a hard test set. Created notebooks/talc scripts for training and testing of EfficientNetB0 and EfficientNetB4 models. General report preparation and editing.	3
Jose Cazarin	Took 5 pictures of every card for the original dataset, took 53 more clear pictures (one of each card) and added rotation to them, creating the easy dataset. Wrote the code necessary to train models with two outputs (PlayingCardsGenerator.py) and wrote the code used to apply data augmentation (data_augmentation.py) to the original dataset. Researched more datasets on the internet and merged them together with our dataset. Wrote the notebooks for creating a model from scratch, training it and testing it. Created the sample applications available.	3
Michael Lee	Took 5 pictures of every card for the original dataset. Researched the Kaggle dataset as part of the original dataset. Wrote talc scripts for training VGG16 models for both single and two outputs. Wrote a talc script for training the Resnet50 model. Created notebooks to test and graph the outputs of the VGG16 and Resnet50 models.	3
Thomas Scott	Took 5 pictures of every card for the original dataset. Wrote Python script for VGG16 and ResNet50 transfer learning models and created notebooks to train models. Researched common top layers for used transfer learning models and ran models on a local system. General report preparation, writing, and editing.	3
Bhavyai Gupta	Took 5 pictures of every card for the original dataset. Developed java code to randomly partition dataset into train and test while considering pictures from every source to ensure inclusion of data from every source. Developed python scripts for EfficientNetB0 (with different top layers) and EfficientNetB4 transfer learning models to run on TALC.	3