Leveraging Branching CNNs to Identify Playing Cards

Logan Jay Williams
Electrical Engineering Department
San Jose State University
San Jose, CA
williamsloganj27@gmail.com

©2023 IEEE

***Abstract*— Image recognition is the first and most critical step of many automation directives. Using the playing card dataset linked in the references, the goal of this project is to identify the rank and suit of a given card with maximal accuracy. This recognition could enable automated mediating in private card games, empower other card game adjudicators, detect cheating, and more. The design process intends to use multiple concepts and tools from EE267 to engineer a model that is optimal for this application. The model was designed for this purpose in order to minimize error and also maximize the chance that at least one of rank/suit was correct and resulted in a branching model. The results are mixed given hardware limitation but show the feasibility of the objective.**

***Keywords*— CNN, branching model, computer vision, backpropagation**

I. Project Objective

This work aims to develop and tune a computer vision model that will recognize specific playing cards from an image. In this case, we presume that the image of the card has already been parsed from a given input image and so the images we use as inputs are already in the format of 224x224 RGB images that only contain the face of the card. The target is to maximize recognition accuracy within the limits of the available hardware.

Using the dataset (referenced at the end, discussed in the next section), the goal is to take similar photos and be able to categorize them on the fly. This sort of recognition would be used in conjunction with a semantic segmentation system to parse video frames or other entire input photos for a variety of applications. Private users could use recognition to provide third party objective refereeing of their private card games. Vision-impaired users could use this as an aid for card games. A camera above a card table could track the cards on the mat and show them on private screens near or far. Casinos or other card game adjudicators could use this to simplify their own work.

Although relatively simple on the scale of possible machine learning applications, this could be a powerful example of simple computer vision affecting everyday activities. However, this particular application also exemplifies a high risk. For high stakes card games with money on the line, a computer vision model with 99% accuracy cannot be used as the sole source of refereeing. Human card dealers, especially at the high-end level, will have a near zero failure rate, so it would take incredible work and a seemingly impossible error rate to replace that kind of work. Thus, this work only purports to suggest a possible aid for general card game adjudicating instead of a replacement.

II. Dataset Detail

The dataset from Kaggle gives pre-processed RGB (3 channel) images that are all resized to be 224x224 pixels. From a visual observation, it is clear that at least some, if not all, of the card have had their proportions skewed in order to maintain a square input size.

Figure 1

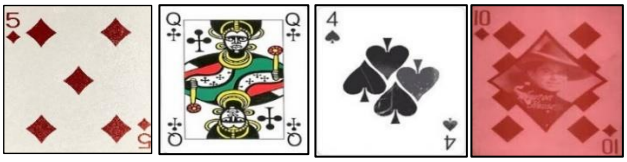

Figure 1. Every card has been scaling to be a non-traditional square shape.

There are approximately 150 examples of each card. However, this results in approx. 2000 examples per suit and approx. 600 examples per rank. Also, there is a large number of 'joker' cards included. In some cases, a designer may want to remove the jokers from the dataset for clarity of inputs. However, for the proposed applications, it's also important that the model recognize at least some 'non-playing' cards (Figure 2).

Figure 2

Figure 2. The jokers look significant different from other playing cards and from each other.

Similarly, there are a few examples of each card that may be considered as outliers because they look so visually different than the others (Figure 1). However, we keep these examples in for robustness. Not only should the model be able to detect cards of varying designs, but perhaps including these in the training set will aid in helping the model to recognize the signature of what makes a card have a given suit/rank. To be clear, by showing crowded/outlier cards adjacent to clean prints of the "same" card, we train the model on the distinguishing factors of a card rather than the arbitrary ones.

Here is an example. For this card, it is a heart. With only clean prints used, the model may learn to recognize all red as being hearts and diamonds while all black being clubs and spades. However, by including this 'dirty' example, we counter this type of overfit. (Figure 3)

Figure 3

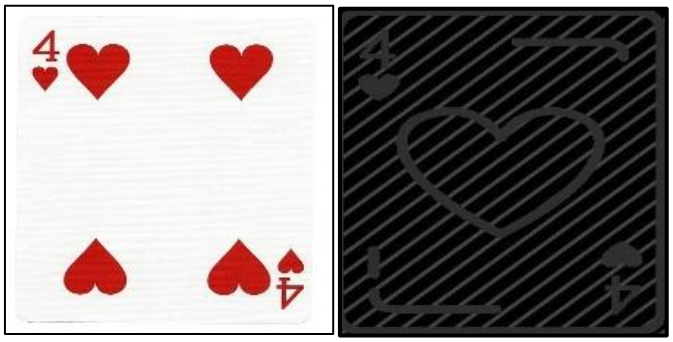

Figure 3. The four of hearts looks very different depending on the sample. The model should be able to categorize all types.

The dataset as given is pre-sorted into training, validation, and testing sets already. However, in our initial processing we actually combine them all in order to allow designer control over the split these groups. We also utilized a random seed in order to ensure we get the same split when testing multiple models.

III. Preprocessing

The computer vision program was built to support three kinds of augmentation to the dataset: horizontal flip, random zooming, and zoom+flip. Rotation was specifically not included under the assumption stated earlier, where some semantic model would correctly present the aligned images to this model.

However, due to hardware limitations with this author's system, the system did not have enough VRAM to support the full dataset in addition to 3 augments per image. And since original inputs are preferable to augmented versions of reused inputs, we opt to use the entire dataset without augments. The hardware limitations will be discussed in a later section.

IV. Basic CNN model

For this model, we prioritize a deep convolutional approach. For a traditional convolutional neural network, there are multiple convolutional blocks followed by fully connected layers. The initial convolutional layers are very low in parameters compared to a fully connected model and thus allow for much better processing of large inputs. Including multiple convolutional blocks allows for the deeper layers to recognize more complex patterns and shapes within the input photo.

Figure 4

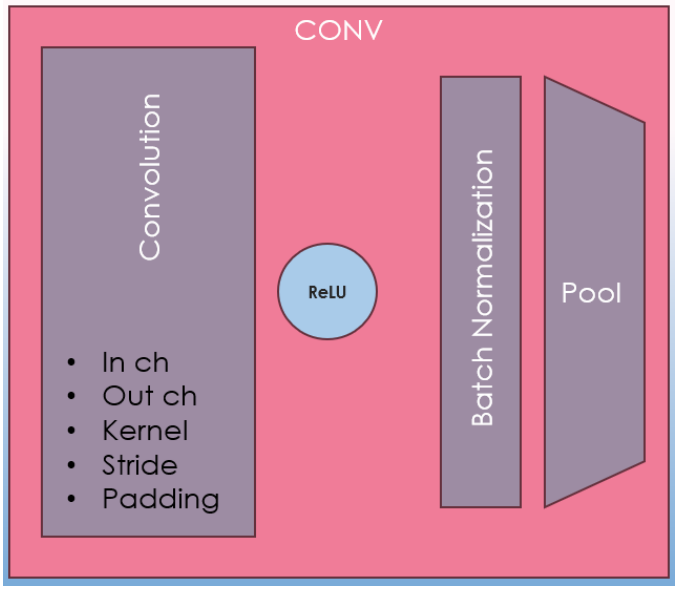

Figure 4. A typical convolutional block consists of a convolution layer, activation function, normalization, and pooling. The convolution block includes several parameters.

After the convolutional blocks is a flattening section and fully connected linear sections. The fully connected layers are optimal for finding the best fit for the given data, so are very important for a model such as this with a relatively high number of output classes.

Figure 5

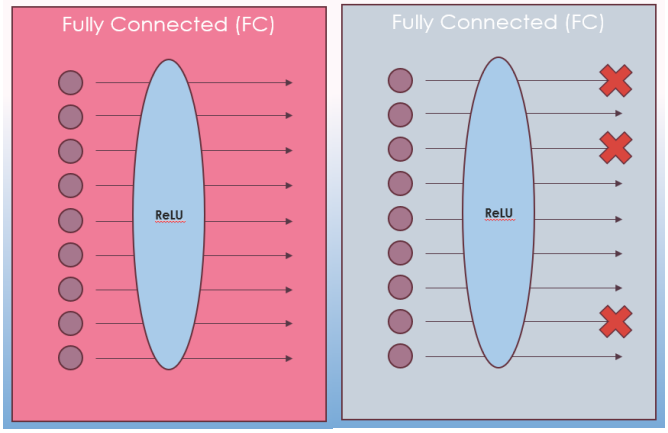

Figure 5. A. The fully connected layer has each neuron pass through the same activation function. B. When dropout is triggered, a random subset of outputs is blocked out.

Additionally, dropout is an important tool to prevent overfitting. We use larger layers to get the highest number of possible features, but don't want specific layers to overfit on certain classes or features. So, for the largest layers we include dropout to help close the gap between training and testing accuracy. Dropout is the random ignoring of some percentage of the outputs from the previous layer.

The following is an *example* of a convolutional network previously created by this author to see what we are framing the model off of for this project.

Figure 6

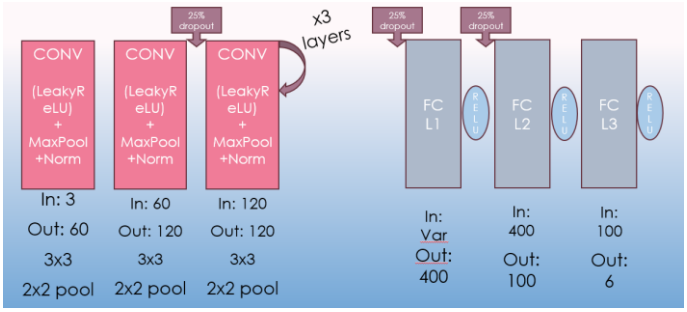

Figure 6. A sample CNN including specifications for each layer and details about the layers' parameters.

V. Creating Branching CNNs

A. Modelling with multiple labels

As discussed previously, this model is meant to maximize the chance of at least partial success. In order to get partial successes, we break the output of the model into two different class types. We predict by the suit (clubs, hearts, etc.) then by the rank (ace, 3, king, etc.).

There are a few ways to classify all 53 card types. Most obviously, we can create a model with 52 output classes. Then, via some post processing, we can sort each of the rank and suit predictions were correct. However, this creates a large and error-prone final output layer. Also, this doesn't allow feedback via backpropagation on whether at least one label (suit/rank) was correct.

To maximize towards that, we can create two different models: one for rank and one for suit. This theoretically will maximize the accuracy of both models since each will be dedicated to only identifying a single label type. However, this is expensive for storage of the model, expensive for training as every train must be performed twice and is especially expensive on memory use (doubly so if these models will be run concurrently). A further extreme is creating a third model that simply predicts 'joker' vs 'non-joker', then passes likely 'non-joker' results to the aforementioned model pair.

B. Branching models

Instead, we propose a combination of the above methods, combining a shared convolutional section and then branching into separate models for identifying their two respective labels. In theory, this minimizes wasted memory usage, allows for double-dipping into the early layers for automatic feature extraction, and enables quicker training and inference. One side would predict suit (including joker) and the other side would predict rank (also including joker). Here is an example of such a model layout.

Figure 7

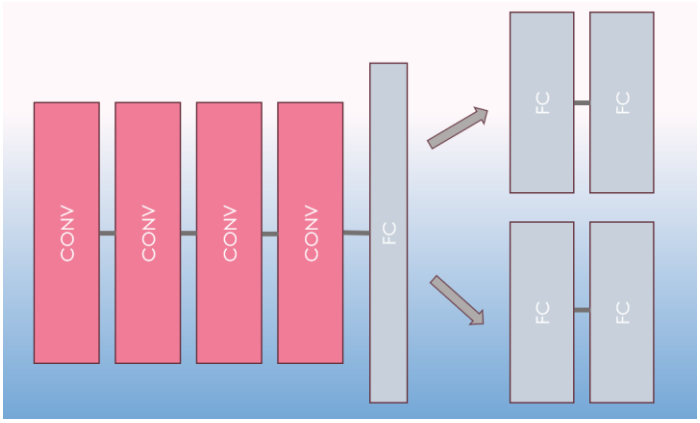

Figure 7. Branching CNN design varies by splitting at some point in the path.

This design will train the larger convolutional+FC section on the general features of the inputs, allowing the final layers to become specialized in different labels. In this case, the final models appear identical, but could easily be entirely different architectures. However, for the sake of training and this relatively quick turnout for this project, it's a positive that these output models are similar.

Inference with this model type is simple. The output of the first 'processing' chunk is fed into the output models to get two total label inferences, one per output model. However, training and backpropagation with these models requires a bit more thought. The error from each output model must be combined before used as error for the processing model.

$$loss = \lambda_1 loss_1 + \lambda_2 loss_2 \quad (1)$$

The selection of these λ s are hyperparameters in themselves. However, with very similar output models, we can expect a similar λ for both. In this case, initial testing showed favoring of the output model with less output cardinality. So the first design was to weight total loss based on the number of output labels.

$$\lambda_i = \frac{p_i}{\sum_k p_k} \text{ for } p \text{ output classes} \quad (2)$$

However, this result overemphasized the model with more labels. Therefore, we use a squared weighting so as not to overemphasize the importance of class numbers while still considering it.

$$\lambda_i = \frac{\sqrt{p_i}}{\sum_k \sqrt{p_k}} \; for\; p\; output\; classes \quad (3)$$

Table 1

Model Weighting	λ 1 (5 classes)	λ 2 (14 classes)	Result
50/50	0.5	0.5	Favor 1
Label count	26%	74%	Favor 2
Squared weighting	37%	63%	1 ≈ 2

Table 1. Depending on the algorithm selected for λ, label set 1 or 2 may be favored.

This type of update of the combined loss functions is not affected by factors such as batch size, optimizing algorithm, or the size of the processing model. Additionally, the entire loss can be scaled linearly as well if there are concerns about vanishing or exploding gradients.

VI. Programming Implementation

The implementation was performed in Python using the PyTorch library for sequential neural networks. See the appendix for the source code repository.

Defining the model parameters is a matter of configuring a sequence of layers and adjusting the parameters for each layer. Activation functions are defined after a given layer and dropout layers are inserted between other layers. (Figure 8)

Figure 8

```
torch.nn.Conv2d(3,60,3,padding_mode='same'),
torch.nn.LeakyReLU(),
torch.nn.LazyBatchNorm1d(),
torch.nn.MaxPool2d(2,2),
torch.nn.Dropout(0.15),
```

Figure 8. Example construction of a single convolutional layer from the model.

For implementing a simple backpropagation algorithm, the sequence of commands is well defined (Figure 9). The program included here includes a library previously developed by Logan Williams to expedite bringup of a computer vision model with arbitrary input data and model.

Figure 9

```
for data in training_data:
  y_pred = model(x)
  loss = loss_fnc(y_pred, y_actual)
  optimizer.zero_grad() # clears gradient
  loss.backward() # BPA
  optimizer.step()
```

Figure 9. Psuedocode for a typical training loop. The loss function is defined from a PyTorch library. The optimizer is also a gradient descent implementation provided by PyTorch.

Some improvements and quality-of-life additions in the custom computer vision library includes the ability to batch the training, functions to split, save, load, and manage the raw data, functions to get validation and test accuracy, save model checkpoints, and create augments of the data.

In order to implement the branching model, we define three different models and train them in sequence. There are three distinct steps to training (inference, gradient, and weight updates) and we have to address them all when working with these sequenced models.

Inference does not involve any special consideration. We simply evaluate each model and feed the output into the relevant second-stage model. (Figure 10)

Figure 10

```
output1 = model_processing(x)
y_suit = model_class1(output1)
y_rank = model_class2(output1)
```

Figure 10. Inference of the branching model.

For gradient calculation, we use the loss function on both outputs, then use our weighted sum from (*1*). Then, we use a separate optimizing algorithm instance for each model. Because PyTorch associates each loss function output with the parameters used to get that output, we don't need to perform any special step for gradient calculation besides the weighted sum. We simply call '.backward()' on the summed loss function (Figure 11).

Finally, for weight updates, we call for the optimizers of each model to perform their '.step()' weight updates (Figure 11). See the full program for additional details.

Figure 11

```
weighted_loss.backward()
self.optimizer_process.step()
self.optimizer_class_1.step()
self.optimizer_class_2.step()
```

Figure 10. Gradient calculation and weight update of each model.

VII. Design and Calibration

Two designs performed full training episodes after initial tweaking of both.

A. *Model 1*

This first design utilizes replicate style convolution padding and LeakyReLU for an efficient but effective activation function. It uses 4 convolutional layers of increasing width in order to provide in-depth feature precision. The model then splits into two nearly identical class models with a small dropout into the two FC layers. The initial dropout from the classifying models is intended to prevent too much double-dipping from the last layer of the processing model. The goal was to prevent the output models 'fighting' to leverage the same nodes for different features. The final class models were kept small to maximize use of the shared convolutional layers as much as possible. This model was used for two different training episodes, discussed in the results section.

Figure 12

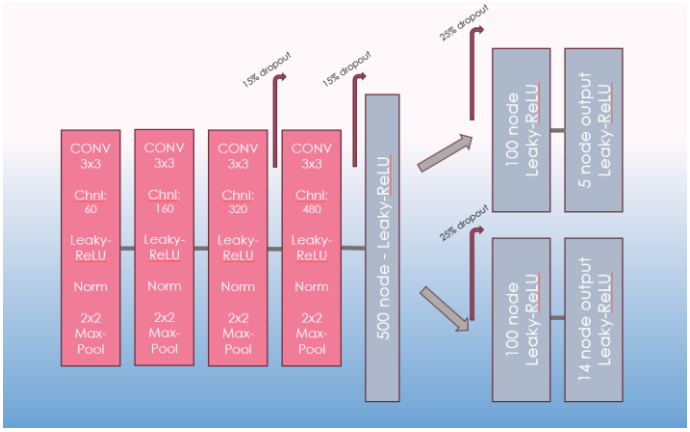

Figure 12. The first model uses a FC layer and several convolutional layers within the processing model and only FC layers in the output class models.

B. *Model 2*

The second model is based off the first but with a few distinctions. First, the 500 FC layer from the processing model was moved to each of the class models. The goal here was to further prevent each model from conflicting at the smaller FC layer. Additionally, the processing model replaced its FC layer with another wide convolutional layer. The goal was to continue to provide maximum feature mappings to each of the child models. Concerns about minimum channel size meant we also removed the pool from the final convolutional layer. This total model is clearly larger and slower, and approaches the two-model approach discussed earlier.

Figure 13

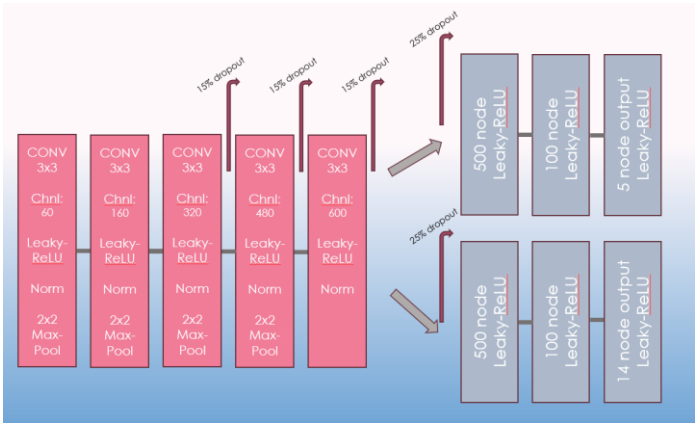

Figure 13. The second model moves the FC processing into the class output models and adds additional convolution.

VIII. Hardware Limitations

In initial testing, there were no issues training either model. However, this was because initial testing was performed with smaller samples of the input images. Eventually, upon testing with all ~8000 input images, the training would run out of memory availability and crash. Therefore, an initial training with 4000 datapoints was performed on the first model and was expected to be used as the final results. However, a second training was performed with all datapoints, except downscaled to only 100x100 input size instead of the full 224x224 resolution. This was done in order to save memory while initially loading each datapoint. While initially expected to have much worse results, since the new input data would have less than a quarter of the resolution, the results were similar. In fact, likely due to the doubled amount of data available, the resulting test accuracy was actually higher than the full-resolution training. This lower resolution solution was used to test the second model as well.

IX. Results

Overall, we have results for three total training episodes between two models. The first is the full resolution training on model 1, then the lower resolution training on model 1, and finally the low res training on the second, deeper model.

A. *Parameters*

The hyperparameters were calibrated initially for the first model, but for consistency of the model comparison, remained the same for all further trainings.

- Optimizing function
  - Adam
- Loss Function
  - MSE
- Epochs
  - 50
  - cut short if validation acc reduced
- Learning Rate
  - 1e-5
- Batch size
  - 10
- Test set
  - 20%
- Validation set
  - 20% of remaining after test set allocated

B. *Result Summary*

The following table is the collected results from each model. The combined accuracy is defined as the percent of points where both suit and rank were accurate, while the "1+" accuracy is the percent where at least one of suit or rank were correct. (See full results in the Appendix, p9.)

Table 2

Model	Suit	Rank	Combined	1+
Model 1 (full res)	74.14%	72.03%	59.55%	86.62%
Model 1 (low res)	83.61%	69.77%	64.81%	88.57%
Model 2 (low res)	80.75%	71.28%	63.76%	88.27%

Table 2. The results of the three models, with top performing model for each category highlighted.

C. Analysis

As shown in the results, the low-res model 1 has the best overall results in 3 of 4 categories. While the 1+ results are similar for each, Model1 shows higher overall in every category except rank accuracy. However, this may be related. If model 1 is biased towards training on suit, then it will have a higher 1+ score since there are only 5 possible suits to get incorrect. If it gets a high enough suit accuracy, it can almost guarantee a higher 1+ accuracy.

Now, all models show a higher suit accuracy than rank accuracy. This may be expected, since of course there are 14 options for rank but only 5 options for suit. It is designer choice whether they prefer to get the highest 1+ accuracy or the highest overall accuracy. In this case, the designer may wish to weight the λs differently to get a closer accuracy for both classifiers. This is also possibly because the suit classifier reached maximum training accuracy and so the overall gradient for the processing model vanished.

Notably, model 2 was much worse at recognizing the jokers. This could be because the larger FC stage in the class models were overfitting on the other classes. The joker is unbalanced relative to these other classes for both sub-models, since each other card gets categorized into subsets for each class label.

X. Future Work

Using this structure, this work can certainly be improved upon with the tweaking of hyperparameters and overall model structure. Using learnings from these results, here are a few opportune areas to focus on first.

A. Model Structure

Seeing the change from model 1 to model 2 using identical other settings, we hypothesize that learning more toward the direction of model 1 may yield stronger results. One note, the dropout of 25% after the final convolutional layer of model 2 compounds with the input dropouts of 15% for the class models, resulting in a 36.25% total dropout, which may be too high for this model. Also, pushing more of the FC layer into the class model may have been the wrong direction. A future model may try using an even smaller class model or none at all (except output neurons). We cannot make a conclusion on whether the convolutional layer added to model 2 caused any major change to the performance of the rest of the model.

B. Hyperparameters

One issue that became more apparent as the higher intensity trainings began is that the split in accuracy between rank and suit class labels became broader. While this is expected for the label with more classes, we can make effort to reduce this effect by tweaking the hyperparameters λ1 and λ2. Currently, they are algorithmically set based on the relative number of parameters in the model and they are set over the course of the training. However, if the parameter was tweaked over the course of training similar to the SGD algorithms, it may result in a much better result. The validation accuracy for each class model could be used to affect its weight. Here is a possible algorithm for this theory.

$$a_i = val.accuracy,$$

$$\lambda_i = \frac{\sqrt{1-a_i}}{\sum_{k=0}^{n}\sqrt{(1-a_k)}}, \text{for } n \text{ class models} \quad (4)$$

This model results in the λs still summing to 1, which stop these hyperparameters from affecting the Adam (or other) gradient descent learning rates.

C. Inputs

After lowering the input size for the sake of memory handling, we propose two hypotheses to counteract this issue. Of course, using more capable hardware will also resolve this issue to a certain point.

1) Code Optimization

By changing the code so the original inputs aren't loaded into memory and instead open each file as its needed for the epoch and close immediately after. This code will need to be pipelined and may not be possible in a high-level language like Python. In C or C++, the memory could be handled much better and may allow for this optimization.

2) Middle ground

Examining different input sizes as a hyperparameter will help improve the results at a design level. However, this is functionally the same as measuring the limitation of a given machine.

D. Joker Handling

One major outlier and point of struggle for the models was the joker card. While every other card has both a suit and a rank, the joker is simply the joker. The joint model handles this by assigning both a false rank and suit to this card. However, it is a disproportionately rare card taking up the equivalent of 27% of the total model bandwidth, or 14x the amount of relative output space as the other cards. Again, we propose two possible solutions.

1) Null Class in Suit Model

Instead of handling joker in both the suit and rank class models, we may instead only include this label in the rank model, which is already sorting on a more granular level. This would reduce the joker's model output space to that of any other card. However, we have to address how it is handled in the suit model. We propose to feed it back as a null result ([0,0,0,0]) and set a minimum threshold for activating each output neuron on the suit model. This would allow 'null' as a result for the suit, which may handle not only the joker, but also any unintelligible output seen by that model.

2) Middle ground

Instead of handling the joker as a normal card, instead, insert another class model early in the model. This would remove any potential jokers from the pool and would only pass probably non-jokers onto the class models. The infrastructure would need to be designed such that this new model is both effective and does not significantly bloat the size and speed of the total model.

XI. Conclusion

As an examination of the dataset and the possibility of using a branching model, this project was a success. We were able to get non-trivial results in training 3 joint models simultaneously on a single task. With results that look promising for better hardware and further model design, we can be certain there is merit in this method of multi-class classification. There is also room in the design of hyperparameters that pertain to these branching models, perhaps algorithmically. We also examined how the introduction of a non-balanced, visually outlying joker card can affect the overall design and results of a computer vision model such as this.

For the applications proposed, these quantitative results are certainly not sufficient for use in any setting. Also certainly, major casinos and other invested parties have already invested into technology like this. However, showing that this design is feasible is still an interesting proof of concept. With optimizations, real-time analysis of card faces and their situations might prove to be a useful tool.

References

[1] https://www.kaggle.com/datasets/gpiosenka/cards-image-datasetclassification

XII. Appendix

See the following repository and readme file for instructions on how to run this simulation.

https://github.com/logjay/ee267_computer_vision

Each section below includes two graphs and two tables pertaining to the accuracy of each model. The graphs show single-variable accuracy for a single class label, while the tables show a breakdown of each class and their joint rates. The joint rates are broken down even further by "1+" which determines whether at least one class was accurate and "BOTH" which notes where both classes were correct.

A. *Model 1 (Full Resolution)*

Figure A. 1

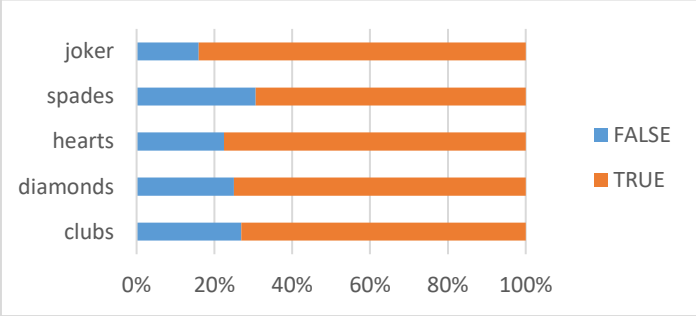

Figure A. 2

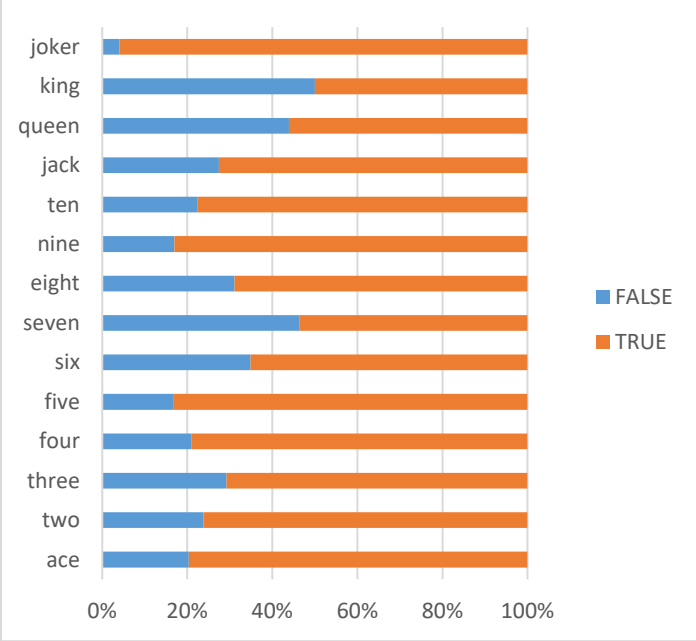

Table A. 1.

	ace	eight	five	four	jack	joker	king	nine	queen	seven	six	ten	three	two	Grand Total
1+ FALSE	**15**	**22**	**18**	**20**	**24**	**4**	**27**	**11**	**23**	**30**	**20**	**20**	**15**	**20**	**269**
BOTH FALSE	4	9	5	5	8	1	10	5	10	14	5	7	4	2	89
BOTH TRUE	11	13	13	15	16	3	17	6	13	16	15	13	11	18	180
1+ TRUE	**39**	**39**	**36**	**37**	**27**	**21**	**13**	**42**	**18**	**24**	**23**	**29**	**26**	**22**	**396**
BOTH TRUE	39	39	36	37	27	21	13	42	18	24	23	29	26	22	396
Grand Total	**54**	**61**	**54**	**57**	**51**	**25**	**40**	**53**	**41**	**54**	**43**	**49**	**41**	**42**	**665**

Table A. 2.

	clubs	diamc	hearts	joker	spade	Grand Total
1+ FALSE	**64**	**66**	**68**	**4**	**67**	**269**
BOTH FALSE	23	20	20	1	25	89
BOTH TRUE	41	46	48	3	42	180
1+ TRUE	**96**	**94**	**92**	**21**	**93**	**396**
BOTH TRUE	96	94	92	21	93	396
Grand Total	**160**	**160**	**160**	**25**	**160**	**665**

B. *Model 1 (Low Resolution)*

Figure A. 3

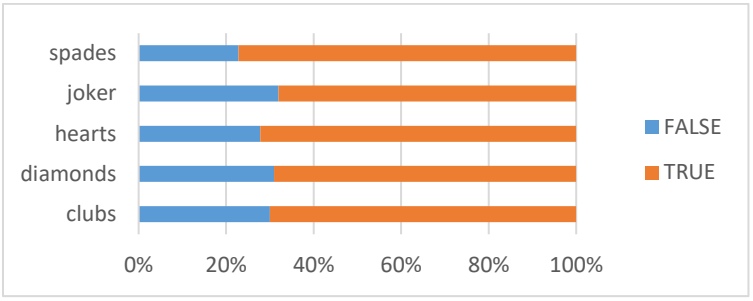

Figure A. 4

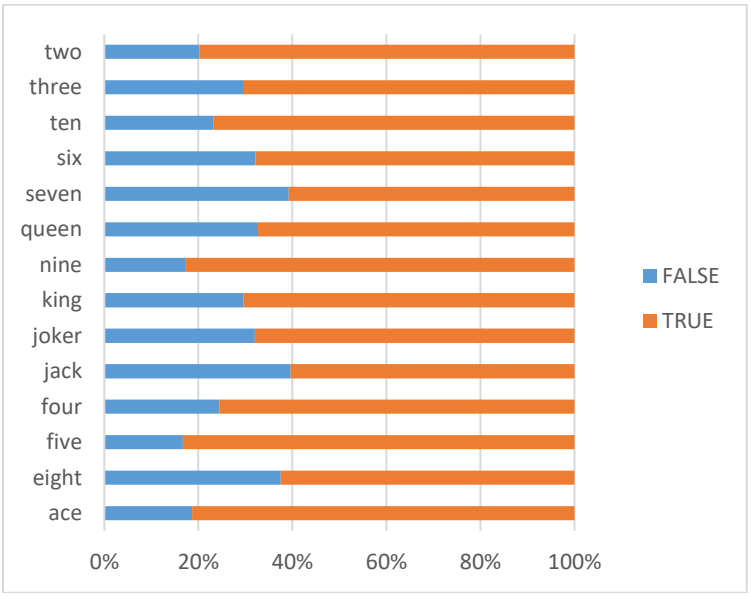

Table A. 3.

	ace	eight	five	four	jack	joker	king	nine	queen	seven	six	ten	three	two	Grand Total
1+ FALSE	**29**	**37**	**20**	**33**	**53**	**10**	**35**	**19**	**37**	**43**	**41**	**32**	**34**	**28**	**451**
BOTH FALSE	5	18	6	7	21	6	10	9	13	7	16	11	13	7	149
BOTH TRUE	24	19	14	26	32	4	25	10	24	36	25	21	21	21	302
1+ TRUE	**62**	**51**	**82**	**61**	**53**	**15**	**46**	**74**	**64**	**59**	**74**	**80**	**67**	**66**	**854**
BOTH TRUE	62	51	82	61	53	15	46	74	64	59	74	80	67	66	854
Grand Total	**91**	**88**	**102**	**94**	**106**	**25**	**81**	**93**	**101**	**102**	**115**	**112**	**101**	**94**	**1305**

Table A. 4.

	clubs	diamonds	hearts	joker	spades	Grand Total
1+ FALSE	**112**	**115**	**113**	**10**	**101**	**451**
BOTH FALSE	27	47	35	6	34	149
BOTH TRUE	85	68	78	4	67	302
1+ TRUE	**208**	**205**	**207**	**15**	**219**	**854**
BOTH TRUE	208	205	207	15	219	854
Grand Total	**320**	**320**	**320**	**25**	**320**	**1305**

Table A. 6.

	clubs	diamonds	hearts	joker	spades	Grand Total
1+ FALSE	**116**	**116**	**117**	**13**	**99**	**461**
BOTH FALSE	38	39	32	10	28	147
BOTH TRUE	78	77	85	3	71	314
1+ TRUE	**204**	**204**	**203**	**12**	**221**	**844**
BOTH TRUE	204	204	203	12	221	844
Grand Total	**320**	**320**	**320**	**25**	**320**	**1305**

C. *Model 2 (Low Resolution)*

Figure A. 5

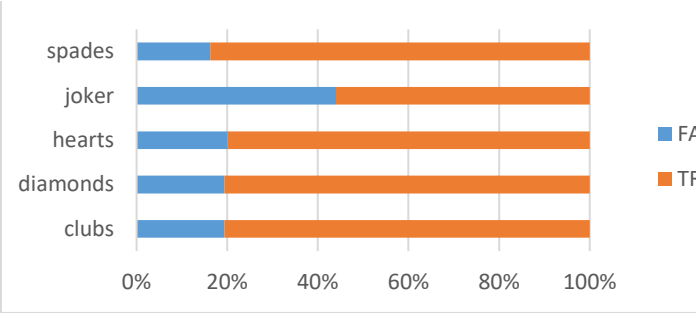

Figure A. 6

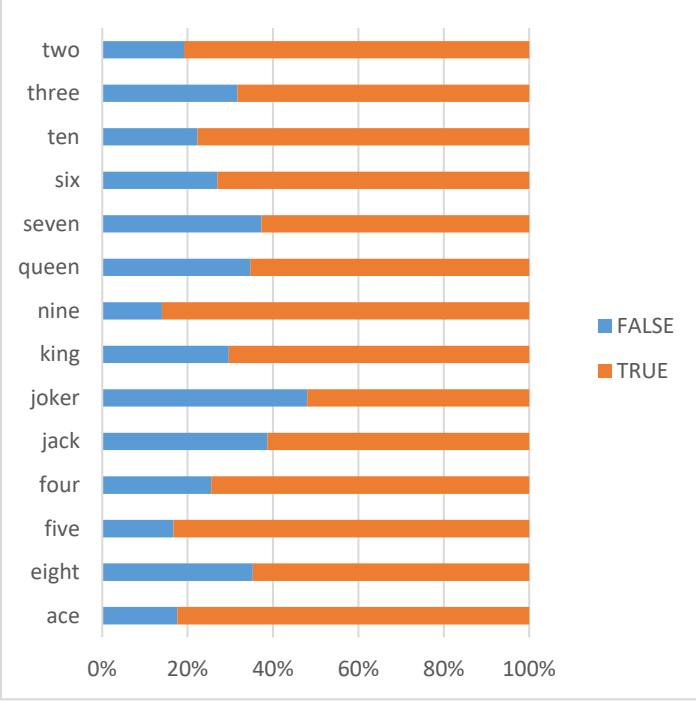

Table A. 5.

	ace	eight	five	four	jack	joker	king	nine	queen	seven	six	ten	three	two	Grand Total
1+ FALSE	**27**	**36**	**21**	**30**	**54**	**13**	**39**	**18**	**47**	**42**	**41**	**33**	**35**	**25**	**461**
BOTH FALSE	5	15	5	14	15	10	8	6	11	10	13	12	15	8	147
BOTH TRUE	22	21	16	16	39	3	31	12	36	32	28	21	20	17	314
1+ TRUE	**64**	**52**	**81**	**64**	**52**	**12**	**42**	**75**	**54**	**60**	**74**	**79**	**66**	**69**	**844**
BOTH TRUE	64	52	81	64	52	12	42	75	54	60	74	79	66	69	844
Grand Total	**91**	**88**	**102**	**94**	**106**	**25**	**81**	**93**	**101**	**102**	**115**	**112**	**101**	**94**	**1305**