# Playing Card Classification using Convolutional Neural Networks

Jerome Goh

**STP598 - Machine Learning and Deep Learning**
Professor: Shiwei Lan

December 8, 2023

# 1 Introduction

While their exact origin cannot be pinpointed for certain, playing cards have been around for a long time. There is evidence of their existence in China as early as the 10th century, as well as in the Middle East by the 13th century, and later in Southern Europe by the 14th century. Their design and use evolved throughout the years and decks comprised of different designs, numbers of cards, and suit-number systems. The "standard" or "traditional" deck that is the most recognizable and commonly played today emerged in the 19th century and is also known as the French-suited playing card deck. It consists of 52 cards, with each card having 1 of four suits (clubs, hearts, diamonds or spades) and 1 of 13 ranks (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King). This deck may often contain Joker cards as well. However, even though this deck has been standardized, there are many different designs of the standard deck such that cards of the same rank and suit may look extremely different.

While there may not be much practical need for constructing an algorithm to classify all 53 classes of cards (including the Joker card) of a standard deck given a digital image input, this still presents an interesting exercise to the challenging task of digital image recognition/classification. Image classification is a growing and important area of research in machine learning. Creating an algorithm that can classify digital image inputs (in this case, images of playing cards) into one of 53 classes therefore presents itself as a fruitful way of exploring a solution to computationally demanding multi-class problems.

One of the most popular neural network architectures that have emerged in the field of image classification is Convolutions Neural Networks (CNNs). A standard CNN includes an input layer, convolutional layers, pooling layers, and a fully-connected layer.

**Convolutional Layer.** A convolutional network takes an input image, scans the image using a filter of a (smaller) set size and stride, and then outputs a feature map of the image that is smaller than the original input. Suppose the input image is $I_{n\times n}$, an $n \times n$ matrix, and the filter is $F_{m\times m}$, a $m \times m$ matrix such that $m < n$. Then, starting with a specified $I_{1_{m\times m}}$ region of the $n \times n$ matrix, the dot product $I_{1_{m\times m}} \cdot F_{m\times m} = I'_{1_{m\times m}}$ is computed, and then the sum of each element of $I'_{1_{m\times m}}$ is computed $s_{1,1} = \sum_{i=1}^{m} i'_{1ij}$. This is repeated, given a set stride, until the feature map is formed, which is a matrix containing all the sum values $s_t$ of the dot products computed in this process $S = \begin{pmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,t} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,t} \\ \vdots & \vdots & \ddots & \vdots \\ s_{t,1} & s_{t,2} & \cdots & s_{t,t} \end{pmatrix}$, where $t = \frac{n-m}{z} + 1$, and $z$ is the size of the stride.

After this, the outputs are usually passed through an activation function. One of the most popular non-linear activation functions, and the one we use in the project, is the rectified linear unit (ReLU). This activation function assigns any negative value to a value of 0 and leaves any positive value unmodified: $g(x) = \max(0, x)$.

**Pooling Layer.** A pooling layer in a CNN also has a filter of its own of a set size. In a similar fashion, the filter scans the feature map matrix using a set stride. However, in this layer, a max function or averaging function is used as an operation. For example, in

a max pooling layer, a filter of $m \times m$ outputs, that is, retains, only the maximum value in a particular $m \times m$ window of the feature map.

**Fully-Connected Layer.** In this layer, each one of the neurons in this layer is, as the name suggests, connected to all the neurons in the previous layer. This layer is useful for, in a sense, compiling the extracted features of the convolutional layers, and learning to discern among different classes based on these features.

In our project, we aim to design and optimize a CNN that would be able to classify all 53 types of cards found in a standard deck of cards.

# 2 Methodology

### Dataset

Our dataset was obtained from Kaggle. It consists of 7624 training images, 265 test images, and 265 validation images. Each image is a $(224 \times 224 \times 3)$ image of ".jpg" format representing one of the 53 cards obtained from a variety of different decks with differing designs.

### Software and Libraries Used

The analysis is conducted primarily through the use of TensorFlow, a machine learning and deep learning library. PyTorch, another deep learning library, is also briefly used. For APIs in TensorFlow, we used keras and kerasTuner, which allowed for intuitive implementation of neural networks and automated optimization hyperparameter searches. We also used Optuna for Bayesian optimization.

Due to the computationally demanding nature of image classification on high-resolution images, we used ASU's SoL supercomputer to run our scripts in a Jupyter Notebook environment.

### Experimental Approach

Our project can be broken down into two parts: 1. Designing our own CNN architecture and 2. Optimizing a pre-trained model for our dataset using transfer learning.

**Designing our own CNN.** The first part of the project involved starting with a simple CNN architecture with commonly used parameters, then adding more layers, regularization and/or hyperparameter optimization as needed. We used parallel PyTorch and TensorFlow to explore the two major deep learning libraries and compare their functionalities.
*Starting models.* Our first model is a simple 2-layer convolutional neural network with the following architectures in both TensorFlow and PyTorch.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 16, 224, 224]             448
              ReLU-2         [-1, 16, 224, 224]               0
         MaxPool2d-3         [-1, 16, 112, 112]               0
            Conv2d-4         [-1, 32, 112, 112]           4,640
              ReLU-5         [-1, 32, 112, 112]               0
         MaxPool2d-6           [-1, 32, 56, 56]               0
            Linear-7                 [-1, 128]      12,845,184
              ReLU-8                 [-1, 128]               0
            Linear-9                  [-1, 53]           6,837
================================================================
Total params: 12,857,109
Trainable params: 12,857,109
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 20.67
Params size (MB): 49.05
Estimated Total Size (MB): 70.29
----------------------------------------------------------------
```

(a) PyTorch

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling_1 (Rescaling)     (None, 224, 224, 3)       0

 conv2d (Conv2D)             (None, 224, 224, 16)      448

 max_pooling2d (MaxPooling2  (None, 112, 112, 16)      0
 D)

 conv2d_1 (Conv2D)           (None, 112, 112, 32)      4640

 max_pooling2d_1 (MaxPoolin  (None, 56, 56, 32)        0
 g2D)

 flatten (Flatten)           (None, 100352)            0

 dense (Dense)               (None, 128)               12845184

 dense_1 (Dense)             (None, 53)                6837

=================================================================
Total params: 12857109 (49.05 MB)
Trainable params: 12857109 (49.05 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

(b) TensorFlow

Figure 1: Simple 2-layer CNN, Model Architecture

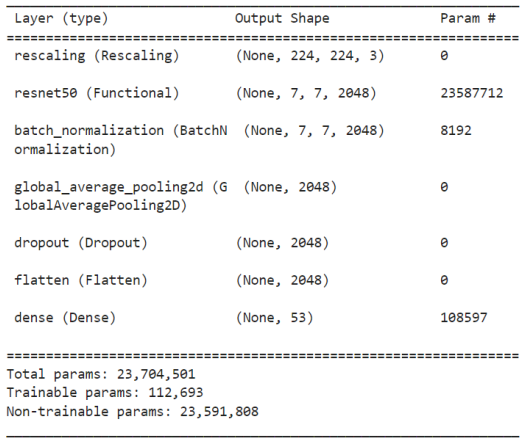
In our PyTorch model, the first convolutional layer ('conv1') takes input images with 3 channels and applies 16 filters of size $3 \times 3$, using a stride of 1 and zero-padding of 1. The second convolutional layer ('conv2') follows, utilizing 16 input channels (from the previous layer) and generating 32 output channels with similar kernel size, stride, and padding, from which the rectified linear unit ('ReLU') activation function is applied after each convolutional layer. Max-pooling is performed using $2 \times 2$ kernels with a stride of 2. The first fully connected layer ('fc1') has $32 \times 56 \times 56$ input features, which corresponds to the flattened output of the last convolutional layer. Then the second fully connected layer ('fc2') connects to the previous layer with 128 neurons and produces the final output, with the number of neurons equal to the specified number of labels.

In our TensorFlow model, The first layer in the model is Rescaling, which normalizes input pixel values to the range $[0, 1]$, which improves training times. The first convolutional layer ('Conv2D') has 16 filters of size $3 \times 3$, using 'same' padding with 'ReLU' activation. followed by the max-pooling layer and then a second similar convolutional layer. The 'Flatten' layer flattens the output from convolutional layers to a one-dimensional vector. The first fully connected 'Dense' layer of 128 neurons uses ReLU activation, followed by the final layer which produces the model's output equal to the number of class labels.

*Optimization:* We manually experimented with various architectures. Once we got the model with the best validation accuracy, we tested adding L2 regularization to convolutional layers, Batch Normalization, and Dropout layers to address overfitting. We finally ran a Bayesian optimization algorithm, with the Optuna API, to optimize the hyperparameters given our choice of the best model, which models the surrogate function to approximate the true performance of the model with different hyperparameter settings. This model is then to be compared with results obtained from our transfer learning models.

**Optimizing a pre-trained model for our dataset using transfer learning.**
*ResNet50.* Transfer Learning refers to utilizing and implementing the weights of an existing pre-trained model, usually trained on huge datasets, into an existing model architecture as a layer. ResNet50 is one such pre-trained model with 50 layers in terms of depth. It is trained on the ImageNet database which comprises more than a million

images across 1000 labels. We chose this architecture because the input size $(224 \times 3 \times 3)$ of the images trained on the model is compatible with our data. Given the popular success of pre-trained models, we believe that this would be a more suitable solution to improving our fit.



| Layer (type) | Output Shape | Param # |
|---|---|---|
| rescaling (Rescaling) | (None, 224, 224, 3) | 0 |
| resnet50 (Functional) | (None, 7, 7, 2048) | 23587712 |
| batch_normalization (BatchNormalization) | (None, 7, 7, 2048) | 8192 |
| global_average_pooling2d (GlobalAveragePooling2D) | (None, 2048) | 0 |
| dropout (Dropout) | (None, 2048) | 0 |
| flatten (Flatten) | (None, 2048) | 0 |
| dense (Dense) | (None, 53) | 108597 |

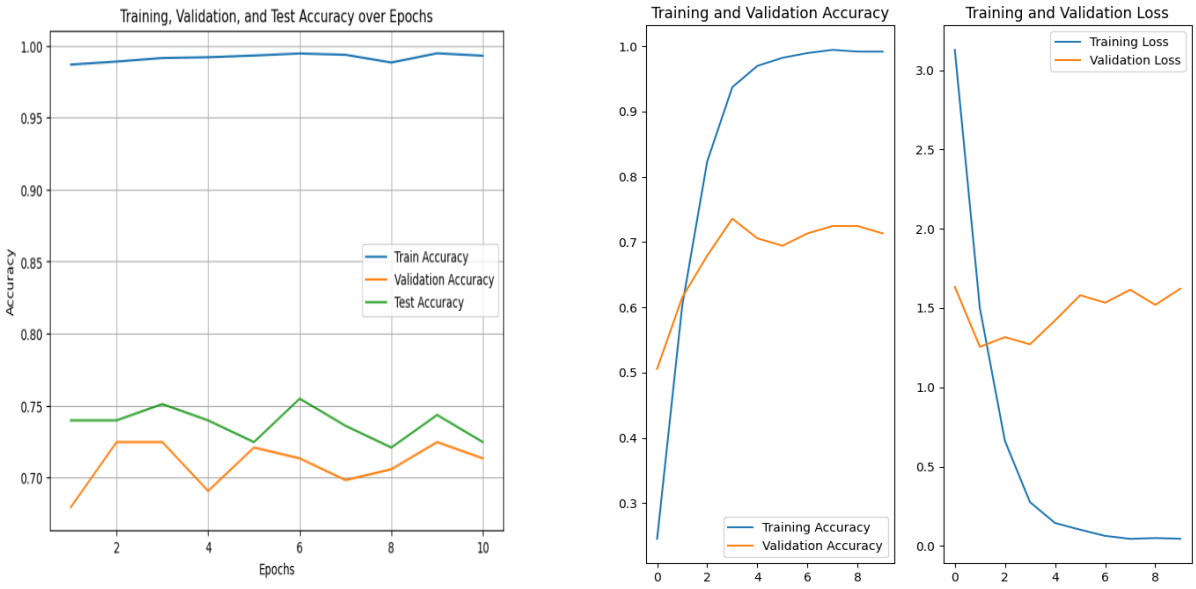
Total params: 23,704,501
Trainable params: 112,693
Non-trainable params: 23,591,808

Figure 2: ResNet50 Model Implementation

*HyperResNet.* After fitting our data on ResNet50, we wanted to see if we could improve the model through optimization. We used KerasTuner's HyperResNet model, which is a hypermodel built on ResNet's hyperparameters, allowing for automated optimization of hyperparameters and architecture selection best suited for the given dataset. We experimented with two hyperparameter search algorithms in KerasTuner in conjunction with HyperResNet, namely GridSearch and RandomSearch. RandomSearch exhaustively evaluates a predefined set of hyperparameter combinations, while Random Search randomly samples hyperparameter values from predefined ranges.

# 3 Results

### 2-Layer CNN




(a) PyTorch (b) TensorFlow

Figure 3: Simple 2-layer CNN, Results

*L2 Regularization.* We noticed that the model was overfitting on the training set, so we decided to add L2 regularization to address that.

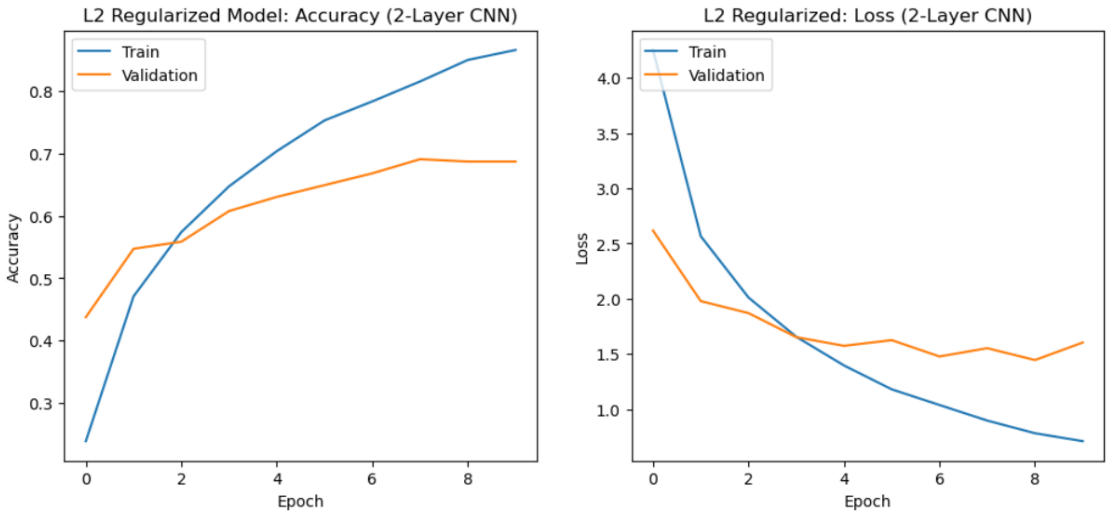



Figure 4: 2-layer CNN with L2 penalization on each Conv2D layer, with $\alpha = 0.05$

*Hyperparameter Tuning: Bayesian Optimization.* We added another layer to our 2-layer model and optimized some of the hyperparameters using Bayesian optimization. These include filters, units in the fully-connected layer, and dropout values. We ran 50 trials with 5 epochs per trial. Our best trial yielded the following parameters for

the model: $('filters\_1' : 32,' filters\_2' : 64,' units' : 256,' dropout' : 0.3172935466205262)$. We then fit the model with those parameters and ran 10 epochs.

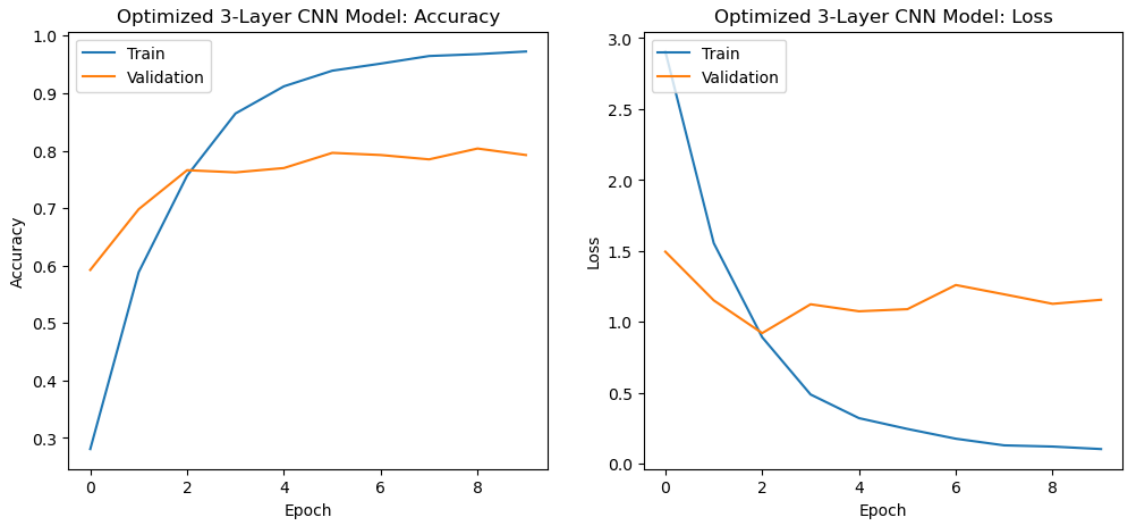



Figure 5: 3-layer CNN with Optimized Hyperparameters using Bayesian Optimization

Both models achieved similar results in 10 epochs, with validation accuracies in the range of $(0.70, 0.73)$. We note that the validation accuracy begins to plateau in this range, as training accuracy improves. This is an early sign of *overfitting*. However, PyTorch took significantly longer, even on the supercomputer, for this simple deep learning task. The syntax for TensorFlow was also much more intuitive. As such, we decided to pursue the rest of the project with TensorFlow instead.

**ResNet50**

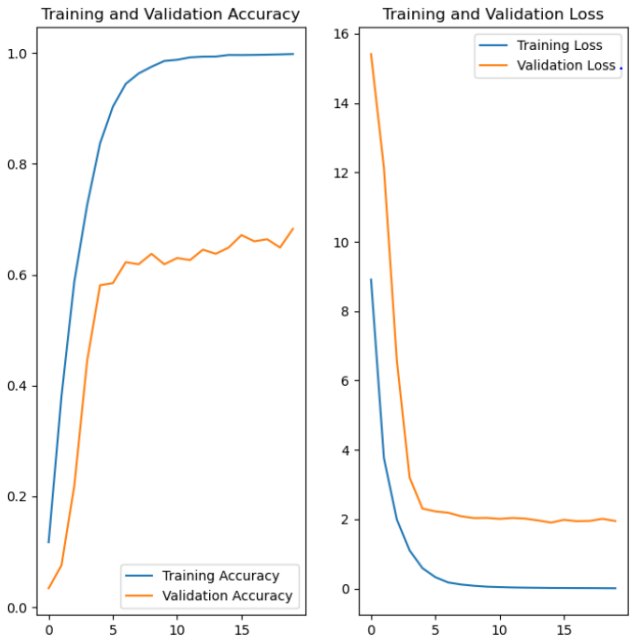



Figure 6: ResNet50 Model Implementation

With the ResNet50 model compiled, we trained it for 20 epochs which gave us the above results. Disappointingly, we note that there were no improvements when compared to our original results, even with fine-tuning by unfreeze-ing our weights. It is counter-intuitive that one of the most successful pre-trained models still overfit or data, which is indicative of other issues.

*Hyperparameter Optimization and HyperModel*

Using GridSearch with the HyperResNet hypermodel, we were able to identify the best hyperparameters:

```
The hyperparameter search (using HyperResNet) is complete. {'version': 'v2', 'conv3_depth': 4, 'conv4_depth': 6, 'poo
ling': 'avg', 'optimizer': 'adam', 'learning_rate': 0.001}
```

Figure 7: Optimized hyperparameters, GridSearch w/ HyperResNet

Using these optimal hyperparameters, we then built a model automatically with the API and trained it on the data for 50 epochs, to identify the optimal number of epochs at a count of 32. Then, we re-instantiated the hypermodel to train based on the optimal hyperparameters and optimal epochs:

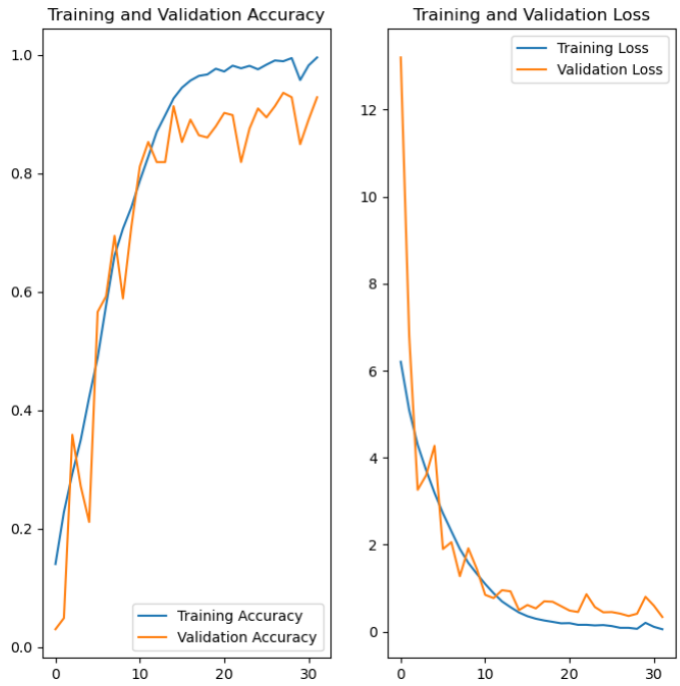



Figure 8: Optimized hyperparameters, GridSearch w/ HyperResNet

```
]: eval_result = hypermodel.evaluate(test_ds)
   print("[test loss, test accuracy]:", eval_result)

239/239 [==============================] - 4s 17ms/step - loss: 0.0527 - accuracy: 0.9871
[test loss, test accuracy]: [0.05265914276242256, 0.9871458411216736]
```

Figure 9: Optimized hyperparameters, GridSearch w/ HyperResNet

We achieved significantly better results with approximately 0.9283 validation accuracy and 0.2879 validation loss, as well as test accuracy of 0.98715 and test loss of 0.052659

We similarly tested RandomSearch with the HyperResNet model, to see if further improvements can be achieved. We identified a different set of hyperparameters:

```
The hyperparameter search (using HyperResNet) is complete. {'version': 'v1', 'conv3_depth': 4, 'conv4_depth': 36, 'po
oling': 'avg', 'optimizer': 'sgd', 'learning_rate': 0.1}
```

Figure 10: Optimized hyperparameters, RandomSearch w/ HyperResNet

Over 50 epochs, we trained the model to identify the optimal epoch at a count of 45. Again, we re-instantiated the hypermodel based on the optimal hyperparameters and epochs:

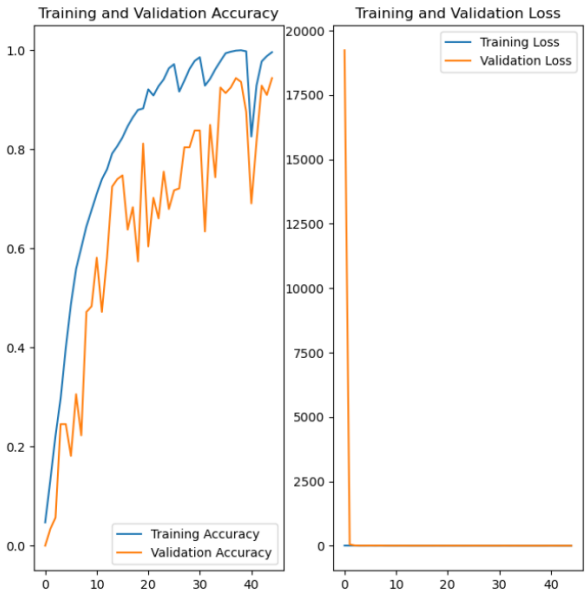



Figure 11: Optimized hyperparameters, RandomSearch w/ HyperResNet

```
eval_result = hypermodel_rs.evaluate(test_ds)
print("[test loss, test accuracy]:", eval_result)
```

```
239/239 [==============================] - 13s 50ms/step - loss: 0.0275 - accuracy: 0.9919
[test loss, test accuracy]: [0.027490193024277687, 0.9918677806854248]
```

Figure 12: Optimized hyperparameters, GridSearch w/ HyperResNet

With RandomSearch, we had slightly better results with 0.2326 validation loss, 0.9434 validation accuracy, as well as 0.02749 test loss and 0.9918 test accuracy.

```
: print(val_loss)
```

```
[19238.421875, 49.34721374511719, 9.779139518737793, 3.5670363903045654, 4.153811454772949, 6.338341236114502, 4.0487
31327056885, 6.7021331787109375, 2.8964850902557373, 2.3589112758636475, 1.956958293914795, 3.1449942588806152, 2.488
9564514160156, 1.3746660947799683, 1.3719632625579834, 1.2791045904159546, 1.6654750108718872, 1.5575944185256958, 2.
7601399421691895, 0.945551335811615, 2.5176939964294434, 1.3853909969329834, 2.1838972568511963, 0.9869624376296997,
1.717570424079895, 2.254672050476074, 1.4640387296676636, 0.9703810214996338, 1.0205063819885254, 0.6584292650222778,
0.5578811764717102, 2.8216428756713867, 0.7023236155509949, 1.6246311664581299, 0.3892669975757599, 0.319659590721130
37, 0.38308537006378174, 0.25823670625686646, 0.30876436829566956, 0.7758459448814392, 2.0019819736480713, 0.86667662
85896301, 0.4176035523414612, 0.4196210205554962, 0.23262417316436768]
```

Figure 13: Validation loss, RandomSearch Hypermodel

However, we note that in the first epoch, we had a bizarre 19238 validation loss, which was reduced to 49.34721 in the second epoch. Nevertheless, the model achieved far better results towards the end.

Table 1 (below) summarizes our findings of the training, validation, and test accuracy of some of the models we have tested.

Table 1: Summary of the Accuracy of Various Models We Have Tested

| | 2-layer | 2-layer (L2) | 3-Layer (Optimized) | ResNET50 | HyperResNET (GridSearch) | HyperResNET (RandomSearch) |
|---|---|---|---|---|---|---|
| Training | 0.9941 | 0.8654 | 0.9726 | 0.9984 | 0.9955 | 0.9957 |
| Validation | 0.6792 | 0.6868 | 0.7925 | 0.6830 | 0.9283 | 0.9434 |
| Test | 0.7057 | 0.6642 | 0.7774 | 0.6788 | 0.9871 | 0.9919 |

# 4 Discussion

Our experimentation of designing a CNN that can discern between 53 different classes among less than 8000 images showed that using transfer learning and pre-trained models is a more efficient way of obtaining an accurate model than starting from scratch. This is especially true of KerasTuner's HyperModel that not only provides ResNET architecture, but also automates optimization of hyperparameters and architecture as well. Unsurprsingly, using this algorithm provided the best results. Our best model seems to be the HyperResNET model that used the RandomSearch algorithm to search for and optimize hyperparameter selection, while GridSearch was only marginally behind, given that the search space of RandomSearch is larger.

In the process of manually constructing our neural architecture, we tried out L1, and L2 regularization as well as Dropout and Batch Normalization in hopes of addressing overfitting. However, these methods did not prove overall to be successful since they either did not reduce overfitting if their parameters were too small, or they ended up severely impacting/reducing and plateauing training and validation accuracy. This may in part be due to the combination of many classes and not enough data, relative to the number of classes we are trying to identify. This is especially apparent since, on average, we have only about 150 images per class. Furthermore, our 2 and 3-layer models may not have been complex enough to be able to properly incorporate regularization and may end up easily (but are strangely complex enough to produce overfitting). Data augmentation incorporated with L2 regularization seemed to plateau training accuracy to below 0.90. To resolve these issues that we found with our manually constructed CNNs, we would need to use optimization techniques that would be able to find a value in the narrow interval for any combination of regularization methods. However, the fact that the unoptimized ResNET50 also similarly overfitted the training data to our simpler models speaks to the necessity of careful optimization.
Besides ResNet50, other pre-trained models deserve testing on this dataset, such as VGG, MobileNet and EfficientNet. Since there is little documentation on HyperModels in general, this is an area of study that is worth investigating, as HyperModels have been the most effective framework for our dataset.

# 5 Conclusion

Our manually constructed and optimized CNN proved to be more time inefficient and did not perform nearly as well as optimized ResNET hypermodels using automated optimization through KerasTuner, such as HyperResNet. This speaks to the power of both

transfer learning and the efficiency of optimization algorithms included in Keras packages.

# 6 References

Galt, D., & Greenwald, D. (1999). Playing Card and Game Collection, PR 115, Department of Prints, Photographs, and Architectural Collections, The New-York Historical Society. Playing card and game collection: NYU Special Collections Finding Aids. https://findingaids.library.nyu.edu/nyhs/pr115_playing_cards_games

Chollet, F. et al. Keras. (2015). GitHub repository, https://github.com/keras-team/keras

Amidi, A., & Amidi, S. (2020). Convolutional Neural Networks cheatsheet star. CS 230 - Convolutional Neural Networks Cheatsheet. https://stanford.edu/šhervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

Wang, Z., Turko, R., Shaikh, O., Park, H., Das, N., Hohman, F., Kahng, M., & Chau, D. H. (Polo). (2020). CNN Explainer: Learning Convolutional Neural Networks with Interactive Visualization. https://doi.org/2004.15004

O'Shea, K., & Nash, R. (2015). An Introduction to Convolutional Neural Networks. https://doi.org/10.48550/arXiv.1511.08458