Simple Image Classification Using Transfer Learning, Hyperparameter Optimization, and Data Augmentation on the Google QuickDraw Dataset to Play a Game of Pictionary

Jacob Warshauer[1], Anna Tsatsos[1], Jiin So[1], Chris Hennighausen[1]
[1]*Boston University, Boston, MA 02130, USA*
(Dated: December 16, 2020)

I. INTRODUCTION

A turn in the game of Pictionary is usually played like this: a player moves around the game board before picking up a card with various ideas for what to draw. The player then draws one of those ideas while other players attempt to guess the drawing in progress before time is up. Pictionary plays into how people interpret different drawings and visuals, serving as a example of image classification. Image classification is a useful tool for visual communication, design, and artistic interpretation. For small doodles, playing Pictionary with a machine would be able to show the different ways models interpret doodles, logos, signs, and more.

Using a variety of CNN models and features, we aim to do just that. With common doodle drawings and guesses as our categories, such as what would be seen on Pictionary cards, each group member trained their own CNN model on a selection of doodles from the Google Creative Lab QuickDraw dataset. The dataset is a collection of 50 million user-submitted drawings sorted into 345 categories. These samples are offered as 256x256 images or simplified 28x28 arrays in npy files in which each pixel is in grayscale, allowing us to treat this dataset similarly to how the MNIST dataset has been used. The CNN models were trained on the following ten categories: airplane, Mona Lisa, dragon, giraffe, axe, banana, Eiffel Tower, snail, windmill, and snowman. These categories have similar and different features that put our CNNs to the test.

After training, the CNNs were tested on a portion of these data which had been set aside in order to create a fair comparison of the models. Additionally, we created a program that allows for users to input their own drawings or images into the CNN models for classification. The group's architectures, results, and commentaries are presented in this paper.

II. ARCHITECTURES USED

Each group member created their own CNN architecture to train on the dataset. Additionally, we explored other aspects such as processing the training data and hyperparameter optimization in order to improve our neural networks. The basic architectures (each with names to indicate which group member did what) are listed below.

- **Chris is a Model** A high speed, efficient architecture, created by Chris Hennighausen.
- **JiinNet** A convolutional neural network built with optimization of hyperparameters, created by Jiin So.
- **Anna Banana** A strict, yet streamlined convolutional neural network focused on optimizers and minimal parameters, created by Anna Tsatsos.
- **Warshenstein** A simple convolutional neural network focused on preprocessing data, created by Jacob Warshauer.

The other architectures used are listed below:

- **ResNet50V2**, A residual neural network that takes advantage of skip connections to jump over layers.
- **MobileNetV2**, A residual neural network made for mobile image recognition, smaller more effective version of its predecessor.

III. FULL COMPARISON OF ARCHITECTURES

A. Full Comparison of Architectures Introduction

In order to have a fair comparison between each of our neural networks, all architectures were trained on the same dataset. To accomplish this, the data into two parts: 80 percent of the dataset would be used for individually training, validating, and testing our model, while the other 20 percent would be untouched and reserved for the full comparison between the group's models.

The full comparison compares the architectures each group member came up with to the published ResNet50V2 and MobileNetV2 architectures. In order to compare our architectures, transfer learning was utilized to re-train these architectures on the QuickDraw dataset. ResNet50V2 is particularly massive, so it was heavily restricted by the computational time. Either computational time or number of epochs would need to be restricted to keep a somewhat even playing field; it was decided the latter option would be more simple.

Each model trained on the same training, validation, and test sets over 10 epochs with a batchsize of 1024. All of these trained models were then evaluated on the

untouched *only test* data set. The primary reason for restricting the full comparison was because group members who would want the best results would commit far more time to the training of their own model as opposed to the comparison architectures, thus resulting in a less valid comparison. Restricting the length of the training for each model allowed us to subvert this bias.

B. Full Comparison Results and Accuracies

All architectures, when trained, achieved accuracies above 90 percent on the test set. The rate at which models trained did have more variance, with Warshenstein and MobileNetv2 overfitting in the early epochs [Fig 1]. MobileNetV2 was unable to overcome this. Somewhat confusingly, many models had better validation set accuracies than training set accuracies during the first epoch, something Warsh quadruple checked since it did not align with what was expected.

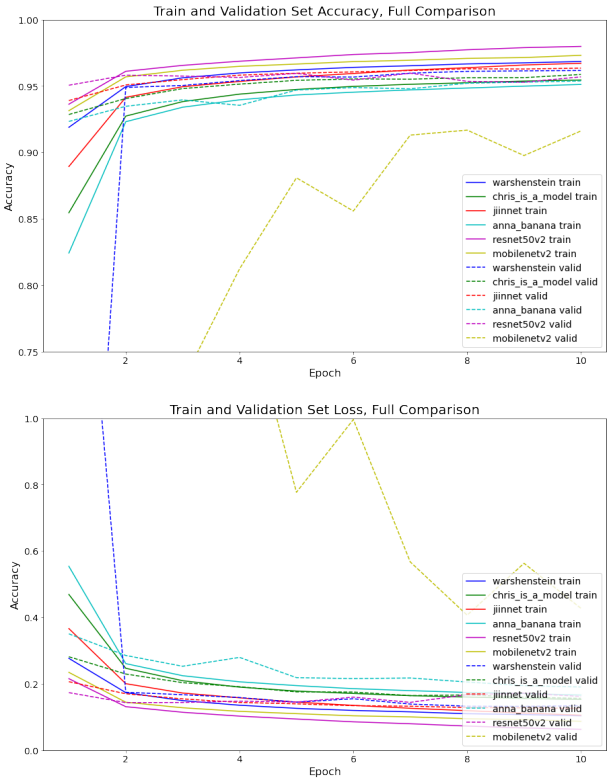

FIG. 1: Training and validation Set accuracy vs epoch.

The best performing architecture was Warshenstein, and the worst performing architecture was MobileNetV2 [Fig 2].

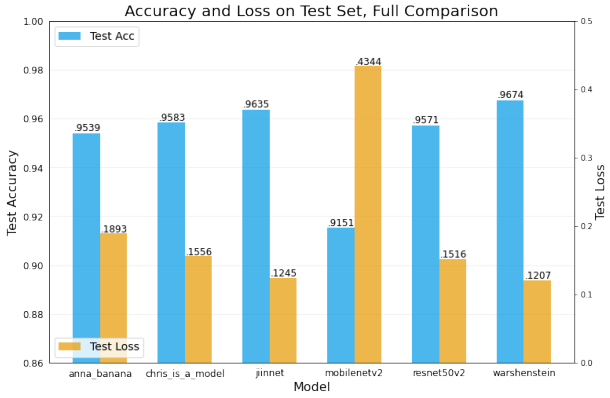

FIG. 2: Test Set Accuracy and loss for all models in the full comparison

C. Full Comparison Commentary

There is a huge diversity of architectures in this section. ResNet50V2 and MobileNetV2 both use skip connections and extremely high numbers of parameters. Both of these are designed for use on much larger inputs than the 28x28 images they are fed here, so this comparison should not be seen as fair. The CNNs created by the group were all designed and optimized for this particular data set.

Despite the disadvantage of not being meant for this application, ResNet50V2 performed quite well, even beating out one of the group's models; however, its training time was excessively long, despite the restrictions put on epoch and batchsize. This should stand as a testament to how flexible this architecture is, performing extremely well without much modification to this transfer learning task.

MobileNetV2 did not perform as well. It is a significantly smaller architecture than ResNet50V2, but still is more than four times larger than any CNN architecture in the group. MobileNetV2 overfit excessively, and was not able to train long enough to correct this [Fig 1].

As one would expect, the top three models were all from the group; all were designed specifically for this Google QuickDraw dataset. Warshenstein had the best performance, followed by JiinNet, Chris is a Model, our two comparison models, and Anna Banana [Fig 2]. These models were trained and compared on the notebook *ML_QD_Full_Comparison.ipynb*.

IV. UNRESTRICTED GROUP COMPARISON

A. Unrestricted Group Comparison Introduction

In order to add some healthy competition and to see what our own architectures were capable of, a separate comparison/competition was made between the architec-

tures made in the group.

The primary reason for separating these from the comparison in *Full Comparison of Architectures* was limited computational power. If we were to make it fair for the more well known architectures, it would require that we give significant training time to the models which are substantially larger that the architectures we made ourselves. The massive size of architectures such as ResNet is entirely overkill for a doodle classification task such as our project.

For our unrestricted group comparison, group members were allowed to do anything they wanted in training except use the *only test* portion of the data. Computational time and model size were entirely up to the group member.

The reasoning behind each group member's CNN design is discussed in the sections corresponding to each model, and the notebook used to load and compare the models is titled *ML_QD_Unrestricted_Comparison.ipynb*.

B. Group Unrestricted Comparison Results

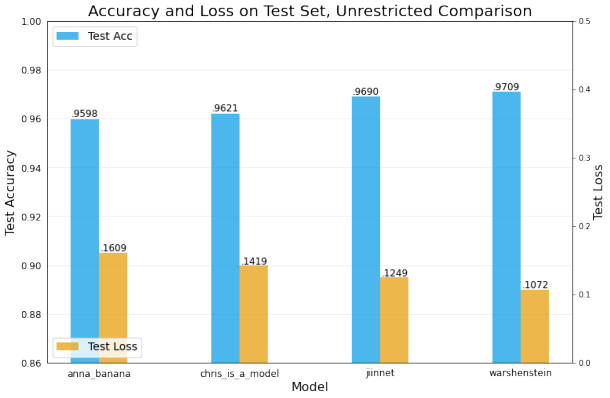

FIG. 3: Test Set Accuracy and loss for all models in the full comparison

C. Group Unrestricted Comparison Commentary

Each group member took a different route with their neural networks: Chris wanted to make a parameter efficient architecture, Jiin experimented with large dense layers, Anna explored other optimizers, and Warsh focused on data augmentation and preprocessing training data. When allowed more time to train, the performance of all models increased, but the order of accuracy stayed the same. All accuracies are in the same ball park. Though Warshenstein had the best accuracy, it took nearly 20 times as long to train as the Chris is a Model architecture. Anna Banana was the only model which used a different optimizer that Adam. JiinNet had the greatest number of parameters at 1.9 million, narrowly beating out Warshenstein at 1.6 million. Despite this, JiinNet trained in around half the time of Warshenstein due to the large portion of its parameters coming from its first dense layer. Chris is a model and Anna Banana both had significantly smaller numbers of parameters at 120 thousand and 80 thousand respectively. As established in the first comparison, number of parameters did not seem to be a major player in the performance of the models. More in depth discussion of the goals and process in creating each model is discussed in the *Individual Architectures Discussion.*

V. INDIVIDUAL ARCHITECTURES DISCUSSION

A. Chris is a Model

The Chris is a model architecture was conceived in an attempt to maximize performance while keeping a small number of parameters. With a large model, the training times are too large to feasibly experiment with how independently changing features of the architecture affects the performance. Therefore, the architecture was limited to 4 convolutional layers and two pooling layers.

The features that were primarily studied were the kernel sizes and activations of the two inner layers, the number of filters of all layers, and the dropout rates. Kernel sizes of the inner layers that were too large caused error messages, and as they were minimized the performance of the architecture improved. It was also found that setting the activations of the inner layers as elu, instead of relu like the outside layers, improved the model. The number of filters was supposed to be as small as possible in order to minimize the parameters, and it was found that at this small range that having each layer larger than the previous helped performance. Lastly, tweaking with the dropout rates revealed maximization between 0.25 and 0.50.

Bringing all of this together produced the following architecture: an input layer with 20 filters, relu activation, and a 5x5 kernel, a (2,2) maxPool layer, a second convolutional layer with 30 filters, elu activation, and a 1x1 kernel, a third convolutional layer with 40 filters, elu activation, and a 1x1 kernel, and the final convolutional layer wit 50 filters, relu activation, and a 5x5 kernel, a 0.35 dropout layer, a (4,4) maxPool layer, a 320 dense layer with relu activation, a 0.35 dropout layer, a 10 category classification dense layer, and lastly a cross entropy compiler with ADAM as the optimizer. In total, this architecture has 119,970 parameters and achieved an accuracy of .962. Although it performed worse than Warshenstein and JiiNet, it is much less complex than them. Its total runtime was an hour and a half, which was shorter than every other model. Chris is a model specializes in efficiency

B. JiiNet

JiiNet's architecture consists of two convolutional layer of 64 neurons (kernel 3x3, activation function relu), each followed by max pooling and dropout, ending with flattening, 800 dense layer, and a ten category classification dense layer. In total there were 1.8 million parameters. The driving strategy behind creating JiiNet was to optimize hyperparameters via Keras's Grid Search. Specifically, the tuned hyperparameters were the batch size, number of epochs, optimization algorithm, neuron activation function, and dropout rate. Of the tested batch sizes {10, 20, 40, 60, 80, 100} and epochs {10, 50, 100} the largest batch size (N=100) with epochs = 50 had the highest accuracy. For the final architecture, we ended up increasing the batch size to 500 and decreasing the epochs to 30 to decrease computational cost, which increased the accuracy slightly. The rest of the hyperparameters were tuned using two epochs for computational efficiency. In comparing the different optimization algorithms, Adam performed the best (accuracy = 0.890974), with the close second and third being RMSProp (accuracy = 0.885645) and Nadam (accuracy = 0.880547). For neuron activation functions, we only tested that of the last dense layer for the ten categories. Although softplus had the best performance (accuracy = 0.890163). we ended up using softmax since we wanted the label vectors to be normalized to one. The convolutional layers worked best with the highest number of neurons {1, 5, 10, 15, 20, 25, 30, 32, 64} at an accuracy of 0.896536. The final training was done by using these hyperparameters together, and reached an accuracy of 0.9690. Restricted by computational cost, the optimization of hyperparameters were by no means exhausted–the grid searches did not test for different combinations of hyperparmeters but rather focused on one parameter at a time. With that in mind, there is much more work to be done in terms of optimization.

C. Anna Banana

The primary study for Anna Banana was in the usage of optimizers to obtain the best accuracy. In testing out different optimizers, Anna Banana found Nadam to reveal the highest accuracies, beating Adam by at most one percent. This could be due to the difference between Nadam and Adam; while Adam examines the previous momentum term in order to perform its gradient descent, Nadam examines the updated momentum term and uses it alongside the updated gradient descent to optimize the weights of each CNN layer.

Like Chris is a Model, Anna Banana was not as accurate as models trained on larger, more powerful computers. However, its runtime was also way shorter than that of the larger, highly-parameterized counterparts. Anna Banana also has the minimum number of parameters to train, opting for stricter regularizers and more filters in each layer rather than more layers to filter the dataset. Combine this with a batch size of 128 and two epochs, and Anna Banana stood at its highest accuracy of 92.3 percent during preliminary training at five percent of the dataset.

Compared to the other models in this set, Anna Banana was not as accurate, being beaten by at most 2 percent. This could partly be due - surprisingly or unsurprisingly - to the optimizer used. While Nadam performed better in the model during the training and validation phase, when it came down to testing using the full dataset, Nadam could have ultimately been a worse performer than Adam, the optimizer used in the other three models presented here. The updated momentum terms in the gradient descent could have hindered the accuracies in a training where the entire dataset is used and there is not a need to be as stringent with the parameters.

D. Warshenstein

The main idea which Warshenstein was meant to explore with this architecture was data augmentation. It would be interesting to explore how some ways to messing with the data set may affect the actual performance. Warshenstein also played with batch normalization as opposed to L1 and L2 regularization. The CNN was structured as follows: Input convolutional layer with 3x3 kernel, outputs 28x28x64; a max pool 2x2 layer with output 14x14x64; a batch normalization layer; a 2nd convolutional layer with 3x3 kernel with output 14x14x128; a max pool 2x2 layers with output 7x7x128; a batch normalization layer; a 3rd convolutional layer with 3x3 kernel with output 7x7x256; a final 2x2 max pool; a final batch normalization; flatten; a 512 dense layer; and a ten category classification dense layer. Overall, it is not particularly remarkable or interesting in its architecture; the interesting portion is in data augmentation.

The data augmentation used random image zoom, rotation, and shift of the images. A grid search was conducted for these values at 0%, 4%, 8%, 12%, and 16% [Fig 4].

The data augmentation also played with random image flipping, both vertically and horizontally. It was hypothesized that this would not increase performance because people generally have a bias for orientation of these types of simple doodles. These results can be seen by comparing top and bottom rows of the grid search [Fig 4].

Typically, the random image flipping did not increase the performance of the model. It was found that around 12% with no random image flipping, the model performed best. This was explored in the notebook titled *ML_QD_warshenstein_data_augment_gridsearch.ipynb*.

Data augmentation typically increased the required number of epochs in training [Fig 5]. On the small test set used for the grid search, accuracy improved from 0.9336 to 0.9514 with 12% images augment. On the full test data set, the accuracy improved from 0.9565 to .9709 with the

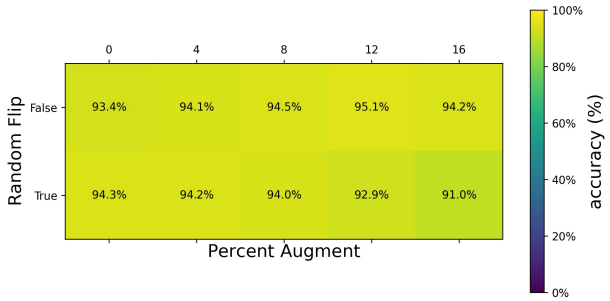

FIG. 4: Plot of the grid search conducted on training data processing. Random image flipping on or off on y axis and different levels of rotation, shift, and zoom on x axis.

12% data augmentation in training.

Warshenstein performed the best in both the full comparison [Fig 2] and the unrestricted comparison [Fig 3], despite its relatively simple architecture. Without messing with the data, this architecture performed significantly worse, indicating the value of preprocessing training data.

VI. PICTIONARY

The pictionary program, listed in the github as *ML_QD_PictionaryProgram.ipynb*, is a program put together in order to allow one to play around with the group's trained models. One is able to select a trained model that will classify either an image selected for upload or a drawing completed in the program. It uses the *tkinter* package to make a GUI, and the instructions for use are on the displayed intro image.

Though many of the correlations and important features that the CNNs find during training are pretty unintelligible, but using the pictionary program one is able to find some interesting connections. It also allows one to compare how different the CNNs are in their classification. Despite all scoring in a similar range on the test set, their interpretation of some doodles can vary widely.

Perhaps the most interesting difference found between the models is that Warshenstein will take a round head with a face as the Mona Lisa, where consistently the other models will take it as a snowman [Fig 6, Fig 7]. Though the origin of this trait is likely impossible to pinpoint due to the complexity of the models, it certainly is interesting.

It is our hope that other interesting characteristics of the CNNs can be explored in this program, and we also hope that it offers a good amount of entertainment value to the user. Instructions for its use are in the notebook, and do not hesitate to reach out to Warsh for help because he fully acknowledges he is not the world's greatest programmer.

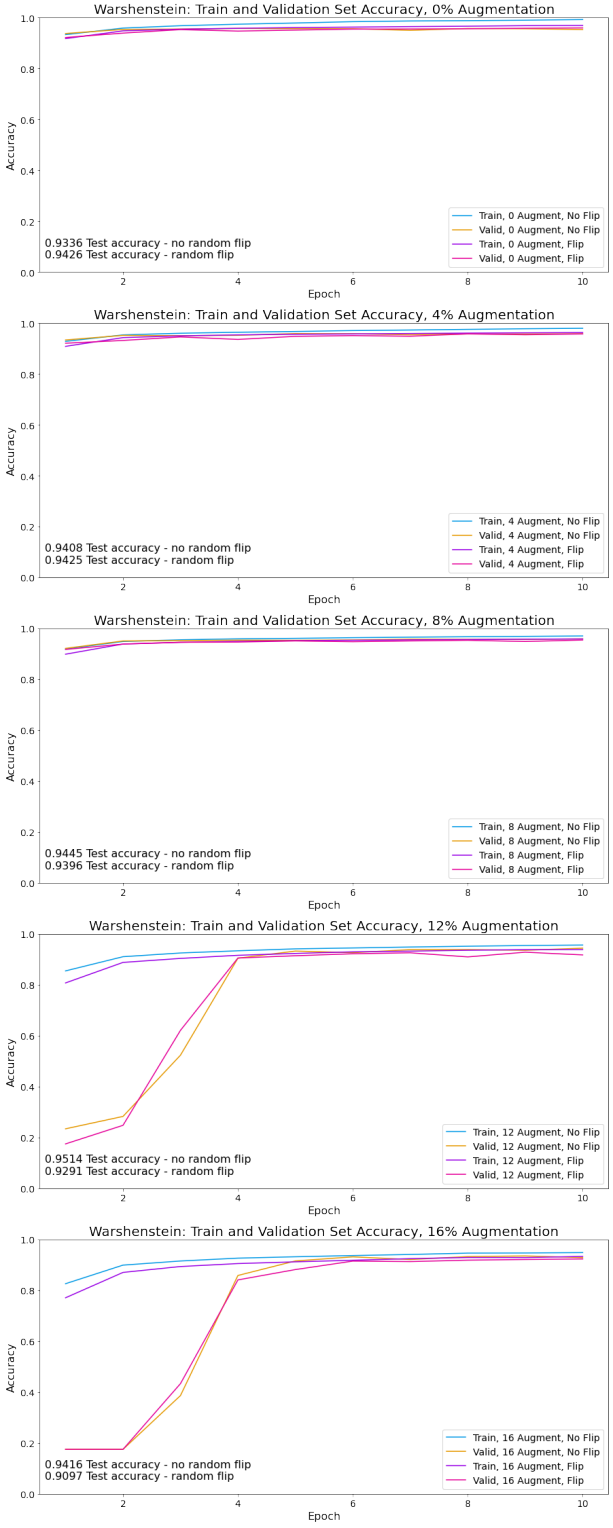

FIG. 5: Train and validation accuracy through training epochs at different levels of augmentation. Each plot is at one level of augmentation with random image flips on or off. Test accuracies are in the bottom left of each plot, corresponding to the values previously shown in Fig 4

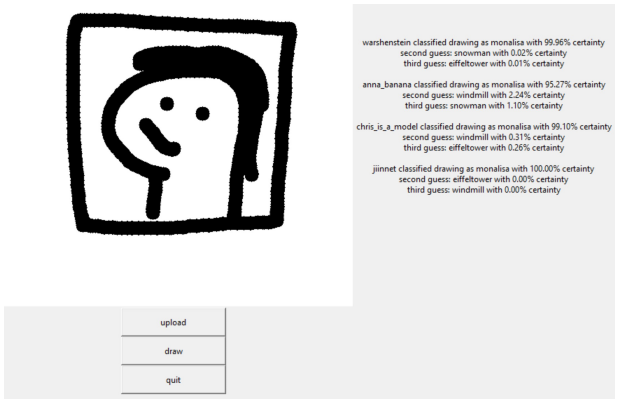

FIG. 6: Image from the Pictionary Program, Warsh's interpretation of the Mona Lisa correctly classified by all of the group's CNNs.

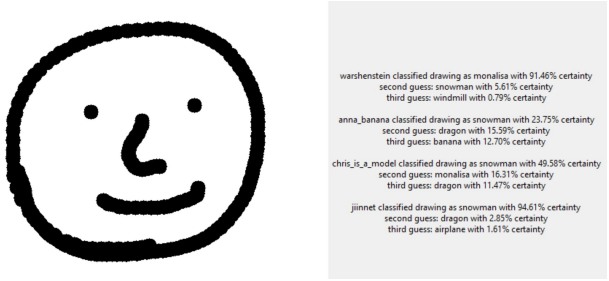

FIG. 7: Image from the Pictionary Program, a simple face interpreted by three of the four group CNNs as a snowman with greatly varying certainty, and classified as the Mona Lisa with great certainty by Warshenstein.

VII. CONCLUSION

In this project, we explored a number of different approaches to supervised learning and convolutional neural networks. From preprocessing data to transfer learning to optimizing training time and parameters, we feel as though we've gained at least a slightly better understanding of the murky process of applying machine learning. By applying the concepts of this class to a common topic such as doodle classification, this group was able to hone in on the specifics of machine learning in order to create CNN architectures that could perform as best as possible. The group member models all had very high accuracies of at least 95 percent when tested on the full dataset, despite the models all having varying characteristics and features to make them unique. Though these fairly high accuracies have been achieved, simple misplaced marks on images can cause high degrees of uncertainty in the models' predictions, showing that these machine learning architectures still have work to do in order to truly recognize human doodles.

VIII. ACKNOWLEDGEMENTS

Special thanks to Professor Pankaj Mehta for the semester, as odd as it was being partially in person and partially online, and thanks the Jupyter notebooks which made this complex topic significantly easier to get hands on with.