Quick, Draw! Doodle Recognition

Kristine Guo
Stanford University
kguo98@stanford.edu

James WoMa
Stanford University
jaywoma@stanford.edu

Eric Xu
Stanford University
ericxu0@stanford.edu

Abstract

Doodle recognition has important consequences in computer vision and pattern recognition, especially in relation to the handling of noisy datasets. In this paper, we build a multi-class classifier to assign hand-drawn doodles from Google's online game Quick, Draw! into 345 unique categories. To do so, we implement and compare multiple variations of k-nearest neighbors and a convolutional neural network, which achieve 35% accuracy and 60% accuracy, respectively. By evaluating the models' performance and learned features, we can identify distinct characteristics of the dataset that will prove important for future work.

1. Introduction

In November 2016, Google released an online game titled Quick, Draw! that challenges players to draw a given object in under 20 seconds. However, this is no ordinary game; while the user is drawing, an advanced neural network attempts to guess the category of the object, and its predictions evolve as the user adds more and more detail.

Beyond just the scope of Quick, Draw!, the ability to recognize and classify hand-drawn doodles has important implications for the development of artificial intelligence at large. For example, research in computer vision and pattern recognition, especially in subfields such as Optical Character Recognition (OCR), would benefit greatly from the advent of a robust classifier on high noise datasets.

For the purposes of this project, we choose to focus on classification of the finished doodles in their entirety. While a simpler premise than that of the original game's, this task remains difficult due to the large number of categories (345), wide variation of doodles within even a single category, and confusing similarity between doodles across multiple categories.

Thus, we create a multi-class classifier whose input is a Quick, Draw! doodle and whose output is the predicted category for the depicted object.

2. Related Work

Similar to our task, Google engineers Ha and Eck used the Quick, Draw! online dataset to train their Recurrent Neural Network (RNN) to learn sketch abstractions. [3] Their goal, however, was to learn to reconstruct, expand, or finish sketches based on input unfinished sketch images rather than to classify sketches. A major strength of their model is their RNN architecture that incorporates ordinal information, which we did not consider for our models.

Kim and Saverese experimented with SVM and KNN performance on image classification, specifically on airplanes, cars, faces, and motorbikes. [4] They found that SVM performed better than KNN, but only because of the KNNs poor performance on car classification. Moreover, they found that the performance for these generative algorithms relied heavily on the characteristics of the data classified, so we attempt to extend KNN to perform better on image classification.

Lu and Tran architected a Convolutional Neural Network (CNN) to tackle sketch classification. [5] Unfortunately, various representations of the same category are indistinguishable for their model, namely sketches of a panda bear either as just the body or as just the face. Our KNN model addresses this issue by using separating out our categories into 5 different representational centroids. Lu and Tran also found that in general, deeper CNNs with moderate dropout to reduce overfitting perform better than shallower networks. We borrow the idea of including dropout, but we do not train very deep CNNs due to the limit of time.

The state-of-the-art as of 2017 comes from a CNN developed by Seddati et al. with their DeepSketch 3 model for sketch classification. [8] Originally attaining a Mean Average Precision (MAP) of 77.64% on the TU-Berlin sketch benchmark from their first DeepSketch model, by adding residuals, they have increased their models performance to 79.18% on the TU-Berlin sketch benchmark as well as 93.02% on the sketchy database. [7] These performance levels are much higher than human MAP of 73% on the TU-Berlin sketch benchmark.

3. Data

Google publicly released a Quick, Draw! dataset containing over 50 million images across 345 categories. There are multiple different representations for the images. One dataset represents each drawing as a series of line vectors, and another contains each image in a 28x28 grayscale matrix. Because we focus on classification of the entire doodle in this project, we use the latter version of the dataset. We treat each 28x28 pixel image as a 784 dimensional vector.

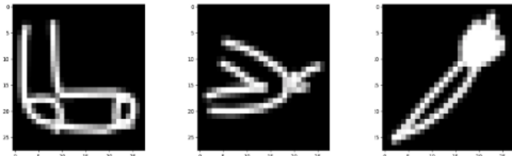

Figure 1. Sample doodles of a sock, elbow, and carrot (left to right) from the training dataset.

To test our models, we split the data into three different folds: 70% for training, 15% for validation, and 15% for testing. To reduce computation time and storage of the data, we decided to create a smaller subset of the original dataset by randomly sampling 1% of the drawings from each category.

As a result, we obtain approximately 350,000 examples for the training set and 75,000 examples each for the validation and testing set. Furthermore, the number of drawings in each category is balanced, so this leaves approximately 1000 examples per category in the training dataset.

4. Methodology

4.1. 1-Closest Centroid (1-CC)

For our baseline, we intuitively assume that all the images in a particular category should look relatively similar. Based on this assumption, one way we could determine which category a given drawing belongs to is by looking at which training examples are the most "nearby" to the doodle under test.

This intuition corresponds with the K-Nearest Neighbors (KNN) algorithm. The vanilla KNN algorithm computes the k training examples that are closest in L_1 or L_2 distance to our current drawing. Then it predicts the category that occurs the greatest number of times among those k neighbors. However, because we have 350k training examples and 75k validation examples, this algorithm requires at least $(3.5 \times 10^5)(7.5 \times 10^4)(784) > 2 \times 10^{13}$ operations to evaluate the entire validation set, which is too slow.

Consequently, we propose a less computationally expensive variant of KNN, which we call 1-Closest Centroid (1-CC). At a high level, 1-CC equivalent to supervised k-means clustering, in which we compute a centroid for each category c using the training dataset and classify test examples according to the closest categories.

In more detail, for each category c, we calculate a "centroid" vector, v_c, by taking the average of all of the vectors belonging to category c. Then, to classify a given vector u, we compute arg $\min_c ||u - v_c||_2$, which seeks to minimize the squared difference in pixel values between the two images. Effectively, we are choosing the category whose mean representation vector is closest in Euclidean distance to our given vector u. This reduces the number of points we look at for each u to only 345 (one per category).

4.2. KNN with K-Means++

1-CC makes the simplifying assumption that all doodles in a category will be similar to each other. However, in reality, there are many different ways to draw a given object. For example, "bear" can be drawn with multiple representations as seen in Figure 2. Thus, one common type of misclassification comes from categories with multiple versions of the object.

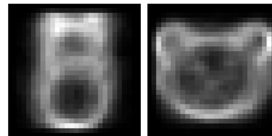

Figure 2. Two of the centroids created by running k-means clustering using k-means++ initialization on the bear training examples.

To remedy this, we hypothesize that creating not one but multiple clusters per category will be able to capture the different variations within a category. To do so, we run k-means clustering on each category's training examples to create 5 sub-centroids per category. In particular, we initialize the centroids using k-means++ initialization to ensure that the final centroids are as different as possible [1]. Then, we follow the KNN algorithm to compare each test example with every generated centroid. Since there are now multiple clusters per category, we determine the final top three classifications using the majority vote of the k closest centroids to a given example, where k is a tunable hyperparameter. We name this method KNN++ for short.

4.3. KNN with K-Means++ and Weighted Voting

We noticed that voting in KNN often ended up with ties. To mitigate ties, we further extend KNN to not only utilize multiple clusters per category, but also to use a weighted voting schema when tallying for final predictions. We name this method KNN++ (weighted) for short.

Intuitively, we wish to count votes from closer centroids more than votes from more distance centroids. Thus, we experiment with two different weighting schemas.

Distance weighting. With distance weighting, each cen-

troid's c's weighted vote w_i is equal to

$$w_i = 1/||x_i - c||_2$$

where x_i is the vector representation of the test example.

Rank weighting. With rank weighting, we first sort all centroids by increasing distance to the test example. Within this sorted order, the centroid c_i at rank i has a weighted vote equal to

$$w_i = 1/\sqrt{i}$$

4.4. Convolutional Neural Network

As a comparison against the above KNN methods, we implement a convolutional neural network (CNN), a state-of-the-art model known for being able to recognize and quickly learn local features within an image.

To achieve the best results, we perform data preprocessing. First, we calculate the mean μ across all training examples as well as the standard deviation σ. We then for each example (training, validation, and test) subtract μ and divide by σ. To account for division by zero errors when dividing by σ, we add an offset of 10 to σ beforehand [2]. Thus, the training data now has zero mean and unit variance, while the validation and test set are shifted so that they are both centered according to the training example distribution.

The model architecture is shown in Figure 3. For a $28 \times 28 \times 1$ doodle, we first run the image through three convolutional filters of size $3 \times 3 \times 5$ with stride one. Furthermore, we add zero padding border around the image so that the resulting outputs have the same width and height. Thus, the dimension of the result after the three convolutional layers is $28 \times 28 \times 5$. The output then goes through a max pooling layer with a kernel size of 2×2, reducing the output to size $14 \times 14 \times 5$.

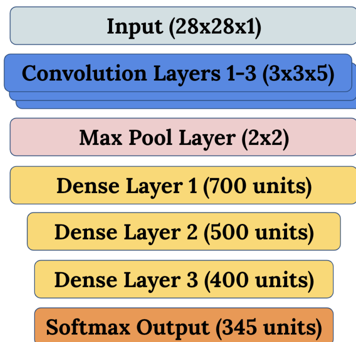

Figure 3. CNN Architecture

Following this, we flatten the tensor so that it becomes a 980-dimensional vector. Finally, we feed the result through three fully-connected or dense layers. Each layer uses the ReLu activation function as well as dropout. The output then goes through one more affine transformation to produce logits of dimension 345 (number of categories) before we apply softmax to generate probabilities for each class. For training the model, we use cross-entropy loss.

4.5. Evaluation Metric

While raw accuracy is a good measure of a model's performance, it penalizes harshly for an incorrect prediction (wrong predictions receive 0 points and right predictions receive 1 point). Since we have so many categories, including some that are extremely similar such as "cake" and "birthday cake", we evaluate our methods not only with raw accuracy but also with a scoring metric that is more lenient of incorrect predictions.

Thus, predictions are evaluated using Mean Average Precision @ 3 (MAP@3):

$$MAP@3 = \frac{1}{U} \sum_{u=1}^{U} \sum_{k=1}^{min(n,3)} P(k)$$

where U is the number of drawings in the test set, $P(k)$ is the precision at cutoff k, and n is the number of predictions per drawing.

Put more intuitively, the equation considers the top 3 predictions (P_1, P_2, P_3) that the model makes for a given drawing. It then assigns a score of $\frac{1}{i}$ if P_i is the correct label for the image and a score of 0 if the correct label is not in the top 3 guesses. Note that MAP@1 is equivalent to single-prediction accuracy.

5. Results and Discussion

Table 1 shows the respective MAP@1 and MAP@3 scores for each model. The best KNN model achieves almost 35% MAP@3 accuracy while the CNN model outperforms all of the other methods with a MAP@3 score of 62%. However, it is worth noting that all methods significantly outperform randomness. Predicting a category uniformly at random would achieve a MAP@3 score of $345(1 + \frac{1}{2} + \frac{1}{3}) \approx 0.5\%$ and a MAP@1 score (single-prediction accuracy) of approximately 0.3%.

5.1. 1-CC and KNN++ Analysis

As seen in Table 1, creating multiple centroids for each category and using KNN created an increase of 4.1% for the classifier's MAP@3 score. However, both still cannot achieve greater than 30% accuracy. For analyzing why this is, we first computed the MAP@3 scores per category. We can then compare the computed centroids for those categories.

1-CC performed best on the categories stairs, circle, and door. KNN++ performed best on the categories stairs, The Eiffel Tower, and bowtie. For these categories, the centroids are either simplistic (circle, door) or are distinct in shape

	1-CC	KNN++	KNN++ (distance)	KNN++ (rank)	CNN
MAP@1 (Train)	18.4%	29.2%	28.3%	29.3%	52.9%
MAP@3 (Train)	24.2%	36.7%	36.3%	37.1%	61.8%
MAP@1 (Dev)	18.2%	17.3%	26.3%	26.8%	53.5%
MAP@3 (Dev)	23.9%	28.0%	33.8%	34.5%	62.2%
MAP@1 (Test)	17.9%	17.0%	26.2%	26.7%	53.4%
MAP@3 (Test)	23.6%	27.7%	33.7%	34.4%	62.1%

Table 1. MAP@1 and MAP@3 scores for each method across train/dev/test

(stairs, The Eiffel Tower, bowtie), which causes the doodles to have less variance. Thus, the centroids are generally contain a clear outline of the object.

On the other hand, 1-CC performed worst on the categories "flip flops", "garden hose", and "wrist watch", and KNN++ performed worst on "dog", "string bean", and "peas". The centroids for these bottom 3 categories are much more vague. For example, "dog" was often confused with other four-legged animals, such as "horse" and "cow".

Furthermore, some categories produced nearly identical centroids, such as "circle" and "octagon" in Figure 4, making it difficult to classify drawings by only comparing pixels with L_2 distance in KNN.

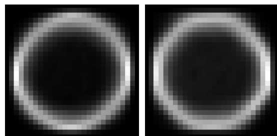

Figure 4. Centroids for the circle and octagon categories.

While creating multiple clusters per category did boost performance, the model still achieved a misclassification rate. To evaluate what types of misclassifications the classifier was making now, we ran a confusion analysis that grouped together categories that were often guessed for each other.

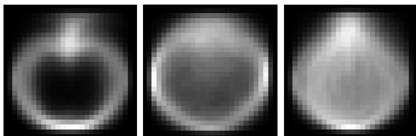

Figure 5. Centroids for apple, blueberry, and onion (left to right).

For example, one group of categories was "apple", "blueberry", "and onion", all of which had generally circular shapes (Figure 5). From analyzing these groups and the category centroids, we deduce that KNN++ was able to generally differentiate between the general structures of doodles, but the local details that differentiated the objects within those groups were often lost, which kept the MAP@3 score lower.

5.2. KNN++ Weighted Analysis

KNN++ with weighted votes by rank produced the highest MAP@3 and MAP@1 scores out of all the KNN++ models. Figure 6 shows the distribution of per-category accuracies running KNN++ with rank weighting and $k = 29$, which is the value of k that gave the highest MAP@3 score on the validation set. Inspecting this distribution, we notice that the distribution is skewed left.

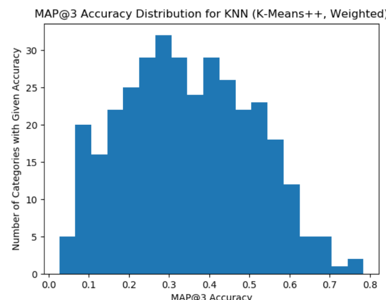

Figure 6. KNN++ (rank weighting, $k = 29$) MAP@3 score distribution by category.

Comparing the two weighting schemas, KNN++ with rank weighting outperformed KNN++ with distance weighting. In particular, Figure 7 shows the MAP@3 scores of KNN++ with both weighting schemes for multiple values of k. As seen, KNN++ with rank weighting produces better, more stable performance at high values of k.

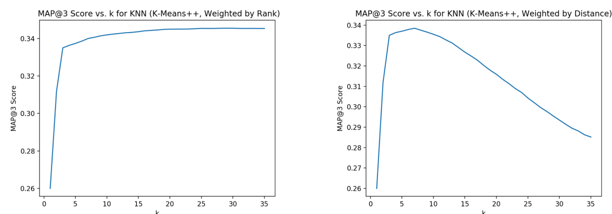

Figure 7. MAP@3 scores plotted against different values of k for KNN++ with weighted voting (rank, distance).

5.3. CNN Analysis

To achieve the best performance for the CNN model, we tuned various hyperparameters including the number

of units in each dense layer, dropout rate, and learning rate. Overall, we found that the model producing the best MAP@3 score on the validation set had three dense layers with 700, 500, and 400 units with each layer having a dropout rate of 0.2. Furthermore, we trained our model with learning rate of 1×10^{-3} and batch size of 32 across 20 epochs.

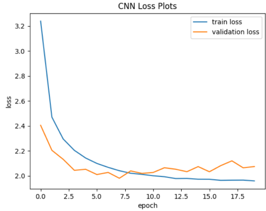

Figure 8. CNN loss.

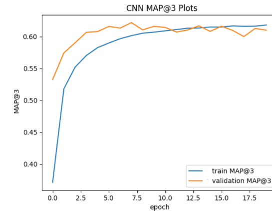

Figure 9. CNN MAP@3.

The end architecture fits the data well, as we see from the loss plot in Figure 8 that the training loss has more or less converged by the 16th epoch. In addition, validation loss slowly begins to increase beginning after the 10th epoch, suggesting that the model has started to overfit the training set. This is further reinforced by Figure 10, in which we see that the training MAP@3 score plateaus while the validation MAP@3 begins to drop after 12 epochs.

Inspecting the accuracy distribution across individual category, we note from Figure 10 that most of the classes have relatively high MAP@3 scores with the median class accuracy being closer to 70%. As a result, the weight of the distribution is shifted towards the right and there is a longer lower tail which drags down the average MAP@3 score to around 62%.

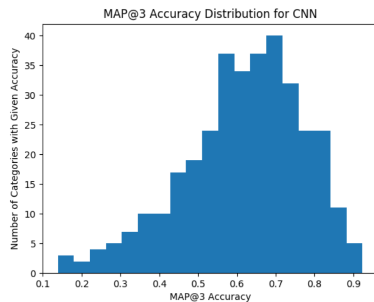

Figure 10. CNN MAP@3 score distribution by category.

Two categories for which the CNN produced very low MAP@3 scores are “garden hose” and “raccoon”. For the category “garden hose”, the model still predicted “garden hose” as most often, but the two other common guesses were “snake” and “snail”. This suggests that while the model learned that a coil is perhaps an important feature for “garden hose”, it was still unable to find more subtle features. Similarly, the CNN commonly predicted “tiger” and “cat” for “raccoon”, implying that it discovered having stripes as a local feature. But given that doodles may be poorly drawn, other distinguishing features of a raccoon are difficult to discern even for a human eye.

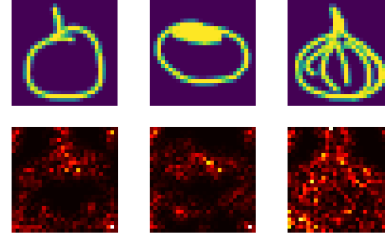
Figure 11. Saliency maps for apple, blueberry, and onion examples.

Nevertheless, we observe that our CNN model recognizes some local features that help correctly classify doodles. Recall that the KNN models only learn a general shape of each category; in the case of apples, blueberries, and onions, it looks for a circle shape. However, looking at the saliency maps [6] produced by the neural net for examples in those three categories in Figure 11, we see that the CNN learns the stem of the apple, flatter top of the blueberry, and the lines drawn within the circle for the onion.

6. Conclusion and Future Work

We found that our CNN outperformed our extended KNN++ algorithm with MAP@3 values of 62.1% and 34.4% respectively, although both algorithms perform much better than random guessing of 0.5% but lower than human guessing of 73.0%. Although KNN++ was able to identify multiple representations of the same category, which increased accuracy compared to 1-NN, KNN++ still came short compared to our CNN due to its inability to recognize features and distinguish between apples and blueberries due to the presence of a stem.

For future work, we would like to experiment with advanced CNN architectures such as VGG-Net and ResNet, which have already reached state-of-the-art levels of image classification performance, although not for sketches in particular. Additionally, we have only used approximately 1% of the total Quick, Draw! dataset, and we believe training our models on the complete dataset would improve accuracy, as well incorporating stroke order information and extract features such as velocity and acceleration. Finally, we believe that ensembling techniques are interesting, particularly for lighterweight methods such as KNN.

7. Contributions

Kristine Guo: Centroid calculation, kmeans++, weighted voting, confusion and accuracy analysis
James WoMa: KNN for kmeans++, categories dictionary, related work
Eric Xu: KNN for kmeans, CNN, evalution metrics and analysis

8. Code

https://github.com/kristineguo/quickdraw

9. References

[1] Arthur, D., & Vassilvitskii, S. (2007, January). k-means++: The advantages of careful seeding. In Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms (pp. 1027-1035). Society for Industrial and Applied Mathematics.

[2] Coates, A., & Ng, A. Learning Feature Representations with K-means (2012).

[3] Ha, D., & Eck, D. (2017). A neural representation of sketch drawings. arXiv preprint arXiv:1704.03477.

[4] Kim, J., Kim, B. S., & Savarese, S. (2012). Comparing image classification methods: K-nearest-neighbor and support-vector-machines. Ann Arbor, 1001, 48109-2122.

[5] Lu, W., & Tran, E. (2017). Free-hand Sketch Recognition Classification.

[6] Simonyan, K., Vedaldi, A., & Zisserman, A. 2013. Deep inside convolutional networks: Visualising image classification models and saliency maps. arXiv preprint arXiv:1312.6034 (2013).

[7] O. Seddati, S. Dupont, and S. Mahmoudi. Deepsketch: deep convolutional neural networks for sketch recognition and similarity search. In Content-Based Multimedia Indexing (CBMI), 2015 13th International Workshop on, pages 16. IEEE, 2015. 14th International Workshop on, pages 16. IEEE, 2016.

[8] Seddati, O., Dupont, S. & Mahmoudi, S. Multimed Tools Appl (2017) 76: 22333. https://doi.org/10.1007/s11042-017-4799-2