CS230

DoodleNet: Accurate Doodle Sketch Recognition

Jiajing Wang, Yixin Shi
{jjwang17, yxshi}@stanford.edu

1 Introduction

The project was motivated by the Kaggle competition “Quick, Draw! Doodle Recognition Challenge” (link). The goal is to build a better pattern recognition classifier with user generated sketch drawings from the Quick, Draw! dataset. For the team, this is a problem with well defined dataset and evaluation mechanism which would allow us to focus more on learning and applying state-of-art convolutional and recurrent models. And as a whole, solving the doodle recognition problem has a potential huge impact because it can help advance applications involving predictive pattern recognitions such as OCR, speech recognition & NLP.

2 Dataset and Features

We used preprocessed Google Quick Draw! dataset (link), with 50M doodles from users of all countries who were asked to do a sketch within 20 seconds on their mobile phones for a common object (e.g. crab, dishwasher, see Figure 1). Each training data example contains vector forms of the strokes in order of drawing, as well as the labeling of the object if it’s recognized or labeled as unrecognized. There are in total 345 distinct classes (categories).

Figure 1: Sample images from Quick, Draw!

Each image was scaled to 256x256 bitmap and all strokes were compressed using the Ramer–Douglas–Peucker algorithm with an epsilon value of 2.0.

Training on the entire dataset proved to be extremely time-consuming on our single GPU instance (e.g. 23 hours for single run of 100K steps), so we decided to limit the dataset to 30 classes to focus on tuning and architecture explorations. We tuned our model using 360,000 randomly sampled drawings (12k per class) and split to 300k/30k/30k as training/eval/test set.

At later stage of the project, we ran another experiment by transforming the stroke-based data into a sequence of time-elapsed 2-d images feeding a modified DenseNet-121 + LSTM network architecture.

Here’s some statistics of the input data:

Metric Name	Average	Max	Min	Std
# of Points	45.8	1063	1	27.82
# of Strokes	5.88	366	1	4.53
Max length per stroke	20.24	881	1	17.18
Min length per stroke	6.72	881	1	11.61

3 Approaches and Findings

Throughout the project, we experimented with 4 different network architectures with two different perspectives to preprocess the input data (i.e. looking at the data in either 1D stroke sequence or 2D image frames). We rebuilt Google’s Recurrent QuickDraw model and analyzed the performance by

Table 1: Tuning the Recurrent QuickDraw model

Dataset size	Learning rate	Batch size	Other parameters	Accuracy
3k drawings per class, 30 classes in total	0.001	8	Steps=100k	83%
	0.01	8	Steps=100k	3% (-80%)
	0.0001	8	Steps=100k	85% (+2%)
	0.0005	8	Steps=270k	85% (+2%)
	0.00003	8	Steps=400k, CUDA-based LSTM	84% (+1%)
	0.00002	8	Steps=400k, 4 LSTM layers	83% (+0%)
10k drawings per class, 30 classes in total	0.0001	8	Steps=800k	89% (+6%)
	0.0003	32	Steps=400k	89% (+6%)
	0.00006	2	Steps=1.2M	85% (+2%)
100k drawings per class, 30 classes in total	0.0001	8	Steps=1.6M	93% (+10%)
100k drawings per class, 340 classes	0.0001	8	Steps=6M	77% (-6%)

tuning various hyperparameters as well as training dataset size, attempted an Attention mechanism add-on. We then decided to run another experiment by transforming the stroke-based data into a sequence of time-elapsed 2-d images and built a simple handwritten conv2d + RNN network, and eventually wrote a modified DenseNet-121 + RNN neural network to learn. We will elaborate details of each approach in the following sections and our interpretations of the results:

3.1 Conv-1D Recurrent QuickDraw network

Following Google's tutorial, we rebuilt the Recurrent QuickDraw model which is a combination of 1-D convolutional layers, LSTM layers and a Softmax layer for prediction (see Figure 2). Intuitively, this model makes a lot of sense because the input data contains both localization data (i.e. where each stroke is drawn on the canvas) as well as chronological data (i.e. order of strokes). So the convolutional layers' roles are to interpret each stroke coordinates into lower lever features e.g. straight line, curves or higher level ones such as circles or a house, while LSTM layers would help better understand the sequences of strokes. We used Adam optimizer and cross entropy loss.

We defined accuracy as our primary evaluation metric, i.e. whether the top one softmax predicted class matches the ground truth class. And we evaluated the performance of the model by tuning various hyperparameters including learning rate, batch size, number of layers as well as the training dataset size. We obtained a baseline accuracy of 83% and analyzed effect of various HP tunings as well as training dataset size, see Table 1 above for the findings.

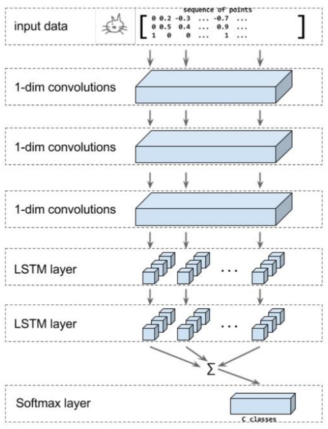


Figure 2: Architecture of Recurrent QuickDraw model

What we have learnt from this exercise:

- Learning rate is a highly influential hyperparameter, where too high a learning rate can lead to non-converging at all (e.g. learning rate = 0.01) and a very low learning rate makes the optimization slower.
- In addition, a reasonable batch size matters, from tensorboard output we noticed that batch size=2 fluctuates the loss curves a lot while a larger batch size makes the optimization smoother.

Table 2: Simple Conv2D + RNN network

Layers	Output size	Architecture / details
Input	8 x 128 x 128 x 1	8 accumulative frames of 128x128 drawings in black & white
Convolution 1	8 x 62 x 62 x 48	5 x 5 conv, stride 2
Convolution 2	8 x 29 x 29 x 64	5 x 5 conv, stride 2
Convolution 3	8 x 14 x 14 x 96	3 x 3 conv, stride 2
LSTM 1 and 2	1 x 256	LSTM layers of 8 stages
Fully connected 2	1 x 30	Softmax output of 30 classes

- But most importantly, deep learning is greedy on training data, as we increase the training data, the accuracy has improved significantly.

3.2 Attention mechanism

Having achieved a reasonable accuracy, we attempted applying attention mechanism to the LSTM model to test whether that would further improve our model. We've implemented fixed length (=4) attention activated LSTM layers and retrained the model with same dataset (10k drawings per class, 30 classes in total) and observed a degradation of accuracy from 89% to 79%. While disappointing, it reinforced our intuition that while attention mechanism works well for applications such as NLP and word sequence generation, the stroke sequence data in this dataset actually would be better understood by looking across a long range (or maybe all) strokes (i.e. how the drawing looks like), and focusing attention to a small segment of strokes would rather hamper the understanding, and thus performance of the model. It is likely that an attention mechanism over a longer span would have better performance, however applying attention seems to be slowing down training significantly (2 -> 0.9 steps/sec by adding a short-span attention), we decided not focus on this experiment.

3.3 Conv-2D understanding of the data

The failure of the attention mechanism prompted us to consider whether there's another way to look at the input data: instead of convolving on the 1-d array of stroke points, would it be better to construct actual snapshots of 2-d drawing over time. (credit to Frank Dai's CS230 project team who inspired us with this idea). This would enable a conv-2d network to "look" at nearby points on the (x,y) coordinates which might be further apart in the sequence of strokes.

In order to preserve the time sequence information, we took 8 snapshots of each drawing as if someone is drawing the doodle in 8 strokes (see Figure 3 for an example).

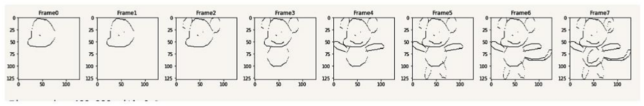


Figure 3: Example sequence of a Monkey doodle

To test out this idea, we wrote a new data preprocessing pipeline to convert 360k strokes based drawings into frames of snapshots. And then trained it with a simple network composed of 3 Conv-2D layers and 3 LSTM layers (see Table 2 for the network architecture). The Conv-2D layers parse the 8 frames of 128x128 image to 8x18816 tensors, which would be passed along into LSTM layers with 8 stages. Not surprisingly, this handwritten Conv-2d network did poorly, only getting 63% accuracy, so we moved on to a more state-of-art architecture: DenseNet.

3.4 DenseNet-121 + RNN

To replace the handwritten Conv-2D layers, we built a modified DenseNet-121 network followed by the same LSTM recurrent layers (see Table 3 for detailed network layout). We knew the DenseNet would perform better without gradient vanishing or exploding, and indded the DenseNet + RNN has performed considerably better than our simple Conv-2D network, achieving a 88% accuracy, which is on par with the best result (89%) of a well tuned Conv-1D + RNN model.

Due to time constraint, we did not have time to further tune this model, but the result looks pretty promising. We believe that there are improvement potentials if we can reduce compression losses

Table 3: A modified DenseNet-121 + RNN network

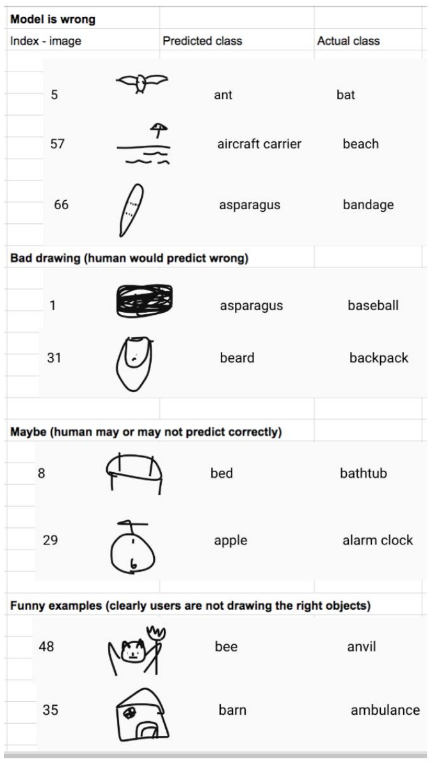
	Layers	Output size	Architecture / details
	Input	8 x 128 x 128 x 1	8 accumulative frames of 128x128 drawings in black & white
Modified DenseNet-121	Convolution	8 x 64 x 64 x 64	7 x 7 conv, stride 2
	Pooling & ReLu	8 x 32 x 32 x 64	3 x 3 max pool, stride 2
	Dense Block 1	8 x 32 x 32 x 256	$\begin{Bmatrix} \text{1 x 1 conv} \\ \text{3 x 3 conv} \end{Bmatrix} \times 6$
	Transition Layer 1	8 x 16 x 16 x 128	1 x 1 conv & 2 x 2 avg. pool, stride 2
	Dense Block 2	8 x 16 x 16 x 320	$\begin{Bmatrix} \text{1 x 1 conv} \\ \text{3 x 3 conv} \end{Bmatrix} \times 12$
	Transition Layer 2	8 x 8 x 8 x 160	1 x 1 conv & 2 x 2 avg. pool, stride 2
	Dense Block 3	8 x 8 x 8 x 352	$\begin{Bmatrix} \text{1 x 1 conv} \\ \text{3 x 3 conv} \end{Bmatrix} \times 24$
	Transition Layer 3	8 x 4 x 4 x 176	1 x 1 conv & 2 x 2 avg. pool, stride 2
	Dense Block 4	8 x 4 x 4 x 368	$\begin{Bmatrix} \text{1 x 1 conv} \\ \text{3 x 3 conv} \end{Bmatrix} \times 16$
	Pooling	8 x 1 x 1 x 368	4 x 4 avg. pool
	Fully connected 1	8 x 128	8 frames of vectors of 128 nodes
	LSTM 1 and 2	1 x 256	LSTM layers of 8 stages
	Fully connected 2	1 x 30	Softmax output of 30 classes

during preprocessing, e.g. by increase number of image frames, and by more thorough hyperparameter tunings.

4 Error Analysis

In addition to exploring these network architectures, we operated an error analysis for one instance of the model (on 10k training data) with a 84% accuracy and manually inspected the 16% of the drawings that were wrongly predicted, comparing to a human evaluation which can be approximated as the bayes error for this doodle recognition problem. To help execute the error analysis, we wrote a simple web application using AngularJS and d3 to render samples of drawings with its ground truth and wrong predictions of the model. Here's what we found and some illustrative examples (Figure 4):

- Within the wrong predictions, a human would not be able to predict 58% of them because they are either incomplete, very bad drawings or user had mistakenly drawn a wrong object;
- Another 23% look difficult to human and would need some luck to guess them right;
- The reminder 19% are obvious for human but the model got it wrong;
- So the estimated bayes error rate would be ~9%, i.e. an upper bound of 91% accuracy, which is close to what our models were able to achieve.



Index - image	Predicted class	Actual class
Model is wrong		
5	ant	bat
57	aircraft carrier	beach
66	asparagus	bandage
Bad drawing (human would predict wrong)		
1	asparagus	baseball
31	beard	backpack
Maybe (human may or may not predict correctly)		
8	bed	bathtub
29	apple	alarm clock
Funny examples (clearly users are not drawing the right objects)		
48	bee	anvil
35	barn	ambulance

Figure 4: Example of error analysis

Some the mistakes were quite entertaining, for example when user drew two bananas instead of one, model got visibly confused :)

5 Conclusion/Future Work

We are happy that we have achieved all the project milestones we set out to do initially. First we've established a baseline accuracy of 83% by rebuilding Google's Recurrent QuickDraw model with a dataset limited to 30 classes. After tuning various hyperparameters, as well as increasing the training set to 10k drawings / class, we improved the model accuracy to 89%. Then we experimented Attention mechanism as well as simple Conv-2D network which did not perform as well because of various reasons mentioned in section 3. In the end, we built a DenseNet + RNN model which had a good performance of 88% by looking at the data as a sequence of drawing snapshots. We also built a web app to perform error analysis and our human evaluation can achieve an accuracy of ~91% due to some bad input data, so in general we were satisfied with our model's performance.

Some future work that we wish we could have done:

- Test an attention mechanism with much longer attention span
- Reduce the preprocessing information loss for the DenseNet+RNN model, e.g. having higher resolution image, more frames of images in a sequence
- Scale the model up for the entire 340 classes and submit result for Kaggle competition
- Experiment with more NN architectures, e.g. WaveNet

6 Contributions

Since we were a small team of 2, lots of the planning, brainstorming, debugging happened as a joint effort during our weekly sync / TA sessions / github collaboration. But we also tried to split larger areas of work for each team member:

Jiajing Wang:

- Wrote a web app for error analysis and did human evaluations of model output
- Some coding changes to enable CUDA and accelerated model training
- Hyperparameter training
- Built the Conv-2d model and DenseNet+RNN model
- Project report writeup

Yixin Shi:

- Graciously sponsored Google Cloud GPU instances for model training
- Built all the input data preprocessing pipelines
- Rebuilt Google's recurrent quickdraw model and setup the model on cloud
- Setup evaluation/ test data pipeline
- Attention mechanism
- Project poster preparation and presentation
- GitHub repository setup / documentations

Project code

All the code of this project can be found on GitHub: https://github.com/yixinshi/tensorflow-test/tree/master/cs230

References

[1] David Ha & Douglas Eck (2017) A Neural Representation of Sketch Drawings. *ArXiv*

[2] David Ha (2017) Teaching Machines to Draw. *Google AI Blog*

[3] Google (2018) Recurrent Neural Networks for Drawing Classification. *tensorflow.org*

[4] David Ha, Jonas Jongejan & Ian Johnson (2017) Draw Together with a Neural Network. *magenta.tensorflow.org*

[5] Yeephycho (2017) densenet-tensorflow *https://github.com/yeephycho/densenet-tensorflow*

[6] Gao Huang, Zhuang Liu, Laurens van der Maate & Kilian Q. Weinberger (2017) Densely Connected Convolutional Networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*

[7] Thang Luong, Hieu Pham & Christopher D. Manning (2015) Effective Approaches to Attention-based Neural Machine Translation. *EMNLP*

[8] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior & Koray Kavukcuoglu (2016) WaveNet: A Generative Model for Raw Audio. *SSW*

[9] Alex Lamb, Anirudh Goyal, Ying Zhang, Saizheng Zhang, Aaron Courville & Yoshua Bengio (2016) Professor Forcing: A New Algorithm for Training Recurrent Networks. *ArXiv*

[10] Python framework used: Tensorflow, Matpotlib