Trans2CNN: A Transformer-Rasterization-CNN Architecture for Freehand Sketch Recognition

CS420 Machine Learning Project

Zenan Li, Qi Liu
{emiyali, purewhite}@sjtu.edu.cn

Abstract

Freehand sketch recognition remains a challenging task for machine learning models such as CNNs due to their high level of abstraction and iconic representations. Recent works propose to capture the unique temporal information of stroke sequences using RNNs, which lay the foundation for sketch representation learning. However, the limited temporal extent of RNNs will restrict the sketch recognition performance. In this paper, we propose Trans2CNN, a Transformer-Rasterization-CNN end-to-end network architecture, which can better capture information from both vector and pixel format sketches. We also develop data augmentation strategies to mitigate the overfitting problem on the sketch dataset. Besides, a transformer decoder is introduced to reconstruct the input, which further boosts the recognition capability. We implement multiple baselines and conduct extensive experiments on the QuickDraw dataset, which verifies the superiority of Trans2CNN. Our code is available at `https://github.com/Emiyalzn/CS420-Sketch-Recognition`.

1 Introduction

Freehand sketching is an easy and quick means of communication because of its simplicity and expressiveness. While human beings have the inherent ability to interpret drawing semantics, it is still a challenging task for machines [9] due to their high level of abstraction and iconic representations. Such unique characteristics, together with the emergence of large sketch dataset like QuickDraw [6], has promoted recent research on sketches, spanning from sketch recognition [10, 21, 20, 18], sketch-based image retrieval (SBIR) [19, 12], to sketch generation and synthesis [22, 6, 11].

With the quick development of deep learning, large number of CNNs [7, 13] have been proposed to solve image recognition problems. Sketch-specified CNNs such as Sktech-a-Net [21, 20] have also been proposed to better deal with abstract sketch images. However, human sketches are often acquired digitally and stored in a vector format, including (1) positional information of points, (2) temporal order (stoke order and point order within each stroke), and (3) grouping of points as strokes (or pen states). The latter two types of information cannot be efficiently accessed by existing CNNs, which deal with the rasterization version of vector sketches.

The recently proposed SketchRNN [6] and its following up studies have shown that RNNs [5, 2] can directly take vector sketches as inputs to learn descriptive feature representations. Motivated by this, researchers have also incorporated the vector format data, serving as a complement to the pixel format data in sketch based tasks. For example, SketchMate [17] adopts a two-branch networks structure: a CNN branch for the pixel sketch and an RNN branch for the vector sketch; a concatenation layer at last is used to fuse feature representations from the two branches. Sketch-R2CNN [10] proposes a novel rasterization layer to build an single-branch RNN-Rasterization-CNN network architecture. Its end-to-end training mechanism boosts the synergy between the RNN and CNN and claims as the state-of-the-art (SOTA) method for vector sketch recognition.

Yet, the limited temporal extent of RNNs restricts the structural complexity of sketches that may be accommodated in sequence embeddings. In the field of neural language processing (NLP), this shortcoming has been addressed through the emergence of Transformer networks [4, 15] in which the self-attention blocks enhances the ability to learn better and longer term temporal structure in the language sequence.

So in this paper, we propose a *transformer encoder* to learn more descriptive representations from the stroke sequence input. Afterwards, we use the *neural line rasterization* (NLR) module as in [10] to convert the vector sketch with the per-point features to multi-channel point feature maps in a differentiable way. Subsequently, an *off-the-shelf CNN* (e.g. EfficientNet [13]) consumes the resulting point feature maps and predicts the target object category as output. This Transformer-Rasterization-CNN (referred as *Trans2CNN*) architecture can promote the information flow between vector and pixel formats of sketches, benefited from both sides to learn more discriminative representations for abstract sketches. Besides, as the sketch dataset is easy to be overfitted due to the distribution shifts [1] in human sketching styles (for example, a cat can be represented by its full body or just a head as well), we also propose *augmentation* strategies [21, 20] based on affine transformation and stroke removal to mitigate this issue. Moreover, a *transformer decoder* is introduced to reconstruct the stroke sequence input, which guides the model to learn more distinguishing embeddings.

In a nutshell, the contributions of this paper are as follows:

- We propose Trans2CNN, a Transformer-Rasterization-CNN end-to-end network architecture for sketch recognition, which can learn from both vector and pixel format sketches.
- We do data augmentation based on affine transformation and stroke removal, in order to tackle the overfitting problem on the sketch dataset.
- We introduce a transformer decoder to reconstruct the input, which further boosts the model's recognition capability. It also brings about generation power to our model.
- We implement multiple baselines, including RNN-based, CNN-based, RNN&CNN combined methods and human tests. Based on these, we conduct extensive experiments and visualization on the QuickDraw dataset (details in Section 4) to show the superiority of our Trans2CNN.

2 Preliminary

In this section, we will briefly introduce existing sketch recognition algorithms. They lay the foundation for our proposed Trans2CNN, and will serve as baselines in the experiments.

2.1 CNN-Based Sketch Recognition

Convolutional neural networks (CNNs) can learn the structural patterns of an image I. Their representation power is further explored after the appearance of ResNet [7], which utilizes so called "skip connections" to enable deep stacking of convolutional layers. The stacked convolutional and pooling layers enable CNNs to learn different abstract level of structural patterns in different layers. Generally, the recognition process can be formualted as:

$$\mathbf{h} = \text{CNN}(I),\ p(\mathbf{y}) = \text{Softmax}(\text{FC}(\mathbf{h})), \tag{1}$$

where $\mathbf{h}$ is the extracted feature embedding from the CNN. It will be fed into a fully connected (FC) layer, using the Softmax operation to get per-class probability $p(\mathbf{y}) \in \mathbb{R}^K$ where K is the total number of classes. However, freehand sketches are highly iconic and abstract, lacking visual cues such as color and texture that CNNs usually capture for classification.

To mitigate this issue, a specially designed CNN architecture is proposed in Sketch-a-Net [21, 20]. Typically, it uses **larger first later filters**. As sketches lack texture information, larger filters can help to capture more structured context rather than textured information. The filter size is set as 15×15, different from the widely adopted tiny 3×3 filters. Besides, it adopts **larger pooling size**. Specifically, a 3×3 pooling size with stride 2 is adopted rather than the widely used 2×2 pooling layer. However, CNNs ignore the unique temporal property of strokes, thus they will inevitably encounter bottlenecks for recognition.

2.2 RNN-Based Sketch Recognition

We consider the input vector sketch S to be an ordered sequence of strokes, each stroke comprising of a sequence of points $S = \{\mathbf{p}_i = (x_i, y_i, s_i)\}_{i=1...n}$, where x_i and y_i are the 2D coordinates of point $\mathbf{p}_i$, s_i is the binary pen state, and n is the total number of points in all strokes. Specifically, state $s_i = 0$ indicates that the current stroke has not ended and that the stroke connects $\mathbf{p}_i$ to $\mathbf{p}_{i+1}$; $s_i = 1$ indicates that $\mathbf{p}_i$ is the last point of current stroke and $\mathbf{p}_{i+1}$ is the starting point of another stroke.

SketchRNN [6] pioneers to exploit the temporal information of vector format sketches utilizing the power of RNNs. Typically, an RNN is adopted to perform analysis on the point sequence of S and then produce a feature vector for each point $\mathbf{p}_i$. At time step i, the recurrent operation of the RNN can be expressed in a general form as:

$$[\mathbf{h}_i; \mathbf{c}_i] = \mathcal{G}_r(\mathbf{p}_i, [\mathbf{h}_{i-1}; \mathbf{c}_{i-1}]),\ \mathbf{f}_i = \mathcal{G}_f(\mathbf{h}_i), \tag{2}$$

where $\mathbf{h}$ represents the hidden states of the RNN, $\mathbf{c}$ is the optional cell states, and $\mathbf{f}_i \in \mathbb{R}^d$ is a d-dimensional point feature output for $\mathbf{p}_i$. The symbol $\mathcal{G}_r$ denotes a nonlinear mapping for recurrently updating of internal states, and $\mathcal{G}_f$ denotes a nonlinear function (usually a fully connected layer) that projects the hidden states to the desired outputs. As in SketchRNN, a bidirectional LSTM [5] unit with two layers are adopted as $\mathcal{G}_r$. As the original SketchRNN is used for sketch generation, we abandon its decoder, and utilize the final hidden state $\mathbf{h}_n^{\rightarrow}$ and $\mathbf{h}_n^{\leftarrow}$ for classification. The arrows in the superscripts denote the two directions of the LSTM, respectively. Typically, we use a fully connected (FC) layer to get the logits:

$$p(\mathbf{y}) = \text{Softmax}(\text{FC}([\mathbf{h}_n^{\rightarrow}; \mathbf{h}_n^{\leftarrow}])), \tag{3}$$

where $p(\mathbf{y}) \in \mathbb{R}^K$ is the output per-class probability for sequence S.

2.3 CNN&RNN Combined Sketch Recognition

Generally, CNNs and RNNs have different advantages in the field of sketch recognition (CNNs focus on image structure while RNNs pay attention to the temporal information). So a natural idea to achieve better recognition performance is to combine the advantage of the two fields.

SketchMate [17] implements this straightforward by input stroke sequence and sketch image simultaneously. A CNN is adopted to embed the image, while an RNN is used to embed the sequence. Finally, the two kinds of embeddings are concatenated and fed into an FC layer to get logits.

However, as pointed out by Sketch-R2CNN [10], the RNN and CNN barely have learning interactions in such a embedding-then-fusion design, and it demands the networks to learn to balance contributing weights of the two types of features in the concatenated feature space.

To compensate for this issue, Sketch-R2CNN only takes in the vector format sketch input and proposes a neural line rasterization (NLR) module to perform in-network vector-to-pixel sketch conversion. Typically, the NLR module takes as input the point sequence of S with per-point features output by the RNN $\{(\mathbf{p}_i, \mathbf{f}_i)\}_{i=1...n}$. Let $f_i^c \in \mathbb{R}$ ($c \in [1, d]$) denotes the c-th component of $\mathbf{f}_i$, and $I^c \in \mathbb{R}^{h \times w}$ be the c-th channel of the resulting feature maps (that will go through a CNN to get final classification result). Here we describe the rasterization process of $\{(\mathbf{p}_i, f_i^c)\}$ to I^c as it is also utilized in our Trans2CNN model, and the symbol c is omitted in the remainder part for simplicity.

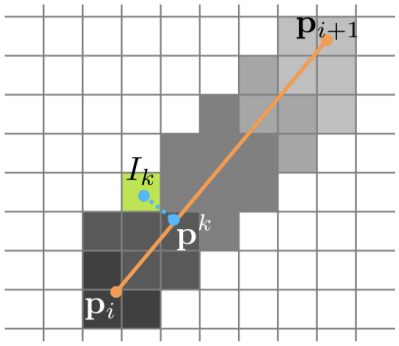


Figure 1: Rasterization of line segment $\mathbf{p}_i\mathbf{p}_{i+1}$ and linear interpolation of the feature value for stroke pixel I_k [10].

In the forward pass, the basic operation of NLR is to draw each valid line segment $\mathbf{p}_i\mathbf{p}_{i+1}$ (i.e. for those $s_i = 0$) on to the canvas I. Similar to the conventional line rasterization, to determine whether or not a pixel I_k is covered by the line segment $\mathbf{p}_i\mathbf{p}_{i+1}$, we simply compute the distance from the pixel's center to the line segment and check whether it is smaller than a predefined threshold ϵ (set as $\epsilon = 1$ in our experiments). If I_k is a stroke pixel, we compute its feature value by

linear interpolation; otherwise its feature value is set to zero. More specifically, as shown in Figure 1, let $\mathbf{p}^k$ be the projection point of I_k's center onto $\mathbf{p}_i\mathbf{p}_{i+1}$, and the feature value of I_k is defined as:

$$I_k = (1 - \alpha_k) \cdot f_i + \alpha_k \cdot f_{i+1}. \tag{4}$$

where $\alpha_k = ||\mathbf{p}^k - \mathbf{p}_i||_2 / ||\mathbf{p}_{i+1} - \mathbf{p}_i||_2$, and $\mathbf{p}^k$, $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$ are in absolute 2D coordinates.

Through the above process, a vector sketch can be easily converted into a pixel image (or point feature maps) in the forward pass. In order to propagate the gradients (w.r.t. the loss function) from CNN to RNN in the backward optimization process. Owing to the simplicity of linear interpolation in Equation 4, the gradients for the rasterization of $\mathbf{p}_i\mathbf{p}_{i+1}$ with f_i and f_{i+1} can be computed as:

$$\frac{\partial I_k}{\partial f_i} = 1 - \alpha_k, \ \frac{\partial I_k}{\partial f_{i+1}} = \alpha_k. \tag{5}$$

Let L be the loss function and δ_k^I be the gradient w.r.t. L back-propagated into I_k through CNN. By the chain rule, we have:

$$\frac{\partial L}{\partial f_i} = \sum_k \delta_k^I \cdot (1 - \alpha_k), \ \frac{\partial L}{\partial f_{i+1}} = \sum_k \delta_k^I \cdot \alpha_k, \tag{6}$$

where k iterates over all stroke pixels covered by the line segment $\mathbf{p}_i\mathbf{p}_{i+1}$. With the computation in Eq. 6, the gradients can continue to flow into RNN for optimizing the learning of point features in the vector sketch space.

The NLR module is non-parametric as it emulates the conventional line rasterization. NLR enables the unification of the two sketch spaces (image and sequence) in a single RNN-Rasterization-CNN archtecture network, and its differentiability allows learning interactions between the RNN and CNN. In this sense, RNN complements the CNN with feature representations extracted from a sequential data format. On the other hand, the CNN informs the RNN with 2D spatial relationships of points, which aid the RNN in learning correlations of temporally-distant but spatially-close points.

3 Methodology

In this section, we will discuss our model in details. An overview of Trans2CNN is shown in Figure 2.

3.1 Transformer-Rasterization-CNN Architecture

As discussed in Section 1, we utilize transformers [15] to compensate for the limited temporal extent of LSTM. Suppose we have input sequence $S = \{\mathbf{p}_i = (x_i, y_i, s_i)\}$, our target is to deduce the points-wise features $\mathbf{f}_i$ to be fed into the NLR module.

Typically, as shown in Figure 2, the transformer encoder is comprised of several encoder blocks. First, each multihead attention (MHA) layer is formulated as such:

$$\begin{aligned} \text{SHA}(\mathbf{k}, \mathbf{q}, \mathbf{v}) &= \text{softmax}(\alpha \mathbf{q}\mathbf{k}^\top)\mathbf{v}, \\ \text{MHA}(\mathbf{k}, \mathbf{q}, \mathbf{v}) &= [\text{SHA}_0(\mathbf{k}W_0^k, \mathbf{q}W_0^q, \mathbf{v}W_0^v); ...; \text{SHA}_m(\mathbf{k}W_m^k, \mathbf{q}W_m^q, \mathbf{v}W_m^v)]W^0, \end{aligned} \tag{7}$$

where $\mathbf{k}$, $\mathbf{q}$ and $\mathbf{v}$ are respective *Key, Query* and *Value* inputs to the single head attention (SHA) module. This module computes the similarity between pairs of Query and Key features, normalizes those scores and finally uses them as a projection matrix for the Value features. The MHA module concatenates the output of multiple single heads and projects the result to a lower dimension. α is a scaling constant and $W_{(\cdot)}^{(\cdot)}$ are learnable weight matrices.

After the MHA module, the feature is added by the "skip connection" as in the ResNet [7] and goes through a layer normalization layer (in short, Add&Norm layer). Then, the output is fed to a positional feed forward network (FFN), which consists of two fully connected layers with ReLU activation:

$$\text{FFN}(\mathbf{x}) = \max(0, \mathbf{x}W_1^f + b_1^f)W_2^f + b_2^f, \tag{8}$$

where $W_{(\cdot)}^f$ and $b_{(\cdot)}^f$ denote the learnable weight matrices and biases, respectively. To summarize, the transformer encoder does:

$$F_{enc}(\mathbf{x}) = \overline{\text{FFN}}(\overline{\text{MHA}}(\mathbf{x}, \mathbf{x}, \mathbf{x})), \ E(\mathbf{x}) = F_N(..(F_1(\mathbf{x}))), \tag{9}$$

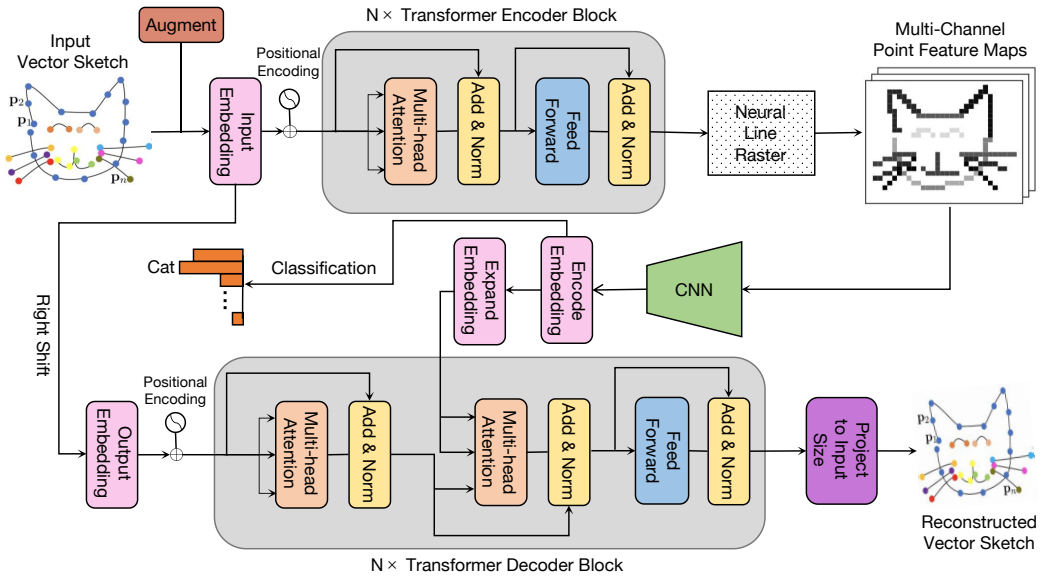


Figure 2: Framework of our proposed Trans2CNN. Typically, the input goes through the transformer encoder to get per-point features, which will be fed into the NLR module to get the multi-channel point feature maps. Subsequently, a CNN is adopted to get the encoded embedding. The Encoded embedding is used for classification, and will be passed into the transformer decoder after expansion. The transformer decoder then tries to recover the vector sketch based on previous strokes.

where $\mathbf{x}$ is the input sequence embedding, $\overline{X}$ denotes the output of X after going through the Add&Norm layer and N is the number of encoder blocks $F_{enc}(\cdot)$.

For each input sequence S, it is first fed into an FC layer to get point-wise embedding, then added by the *positional encoding* (cosine encoding in our experiments) to better capture temporal information. Eventually, it is fed into several encoder blocks described above to get the per-point features that will go through the NLR module to get the multi-channel point feature maps. The remained operation is just the same as that in Sketch-R2CNN [10], the multi-channel point feature maps will be input to a CNN to get encoded embedding, which will finally go through an FC layer to get classification logits.

3.2 Data Augmentation

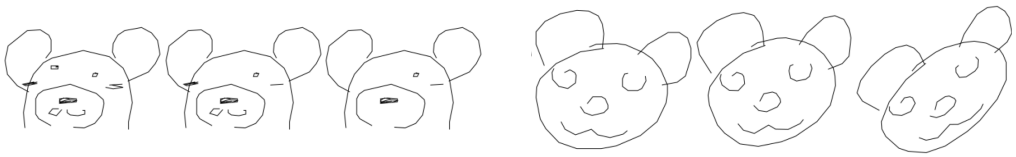
As we found in experiment results that it's easy to overfit on sketch datasets, we propose two data augmentation methods to mitigate this problem. Specifically, as shown in Figure 3, the augmentation strategies are adopted to enrich the diversity of the dataset, which is of critical need due to the diverse sketch styles of humans.

The first strategy is to do **affine transformation**. As this concept is widely used in existing works, we omit its mathematical details here. Generally speaking, it will scale and translate x and y coordinates of the input sequence points, as well as rotate the points by a certain angle.

The second strategy is to do **stroke removal**. Specifically, given a sketch consisting of a set of N ordered strokes $S = \{s\}_i^N$ indexed by i, the order of the stroke and its length are used together to compute the probability of removing the i-th stroke as:

$$p(i) = \frac{1}{Z} \cdot e^{\alpha * o_i} / e^{\beta \times l_i},\ s.t. Z = \sum_i e^{\alpha \times o_i} / e^{\beta \times l_i}, \tag{10}$$

where o_i and l_i are the sequence order and length of the i-th stroke, α and β are weights for these two factors. Overall, the later and the shorter a stroke is, the more likely it will be removed. This removal strategy can create sketches of different abstract levels while maintain the most significant stroke information.



(a) Stroke removal. (b) Affine transformation

Figure 3: An illustration of our data augmentation strategies.

3.3 Transformer Decoder

We also develop a transformer decoder to reconstruct our input stroke sequence. On the one hand, the reconstruction loss can act as a regularization for the encode embedding to capture more distinguishing sketch information. On the other hand, it can also bring generation capability to our Trans2CNN.

Typically, the decoder takes as inputs the encode embedding (expanded to per-point embeddings) and target sequence in an auto-regressive manner. In our case we are learning a transformer autoencoder so the target sequence is also the input sequence shifted forward by 1. In other words, similar to that in NLP, we are going to predict the next stroke based on all the previous strokes. This can be implemented by a MHA block with a *lookahead mask* that filters out the attentions on all the subsequent strokes following a certain stroke.

As shown in Figure 2, the transformer decoder block is just the same as that in the conventional Transformer [15], including two MHA modules which takes the sequence and encode embedding as input, and a FFN module to get per-point features. The complete decoder is comprised of several decoder blocks, and the output is fed into an FC layer to project the per-point features into the input size, which is our final reconstructed result. To summarize, the transformer decoder does:

$$F_{dec}(\mathbf{h},\mathbf{x}) = \overline{\text{FFN}}(\overline{MHA}_2(\mathbf{h},\mathbf{h},\overline{\text{MHA}_1}(\mathbf{x},\mathbf{x},\mathbf{x}))),\ D(\mathbf{h},\mathbf{x}^*) = G_N(\mathbf{h}, G_{N-1}(,...G_1(\mathbf{h},\mathbf{x}^*))), \tag{11}$$

where $\mathbf{h}$ is the encode embedding, $\mathbf{x}^*$ is the shifted forward version of input sequence embedding $\mathbf{x}$, N denotes the total number of decoder blocks $F_{dec}(\cdot)$.

Generally, we employ two losses for training Trans2CNN. First, a **Cross Entropy loss** L_{cls} is used for classification. Second, a **reconstruction loss** L_{recon} is adopted to guide the transformer decoder to reconstruct the input sequence. Our final loss function is $L = L_{cls} + \lambda L_{recon}$, where λ acts as a weight for the reconstruction loss.

It remains to specify the calculation of the reconstruction loss. Specifically, is consists of a L^2 loss term modeling the points positions (x_i, y_i) and a a cross entropy loss term modeling the pen states s_i. We found these losses simple yet effective in learning a robust sketch embedding.

4 Experiments

In this section, we conduct extensive experiments to answer the following questions:

- **Q1:** How effective is Trans2CNN in sketch recognition?
- **Q2:** How do the composed modules impact Trans2CNN's performance? Besides, how robust is Tran2CNN to recognize "bad" sketches, does it learn meaningful embedding for the input sketches?

4.1 Experiment Setup

We briefly introduce our experiment settings in this section, leaving the hyperparameter and implementation details to Appendix A to keep a concise main text.

For the environment requirements, we implement all our models with Python 3.8, Pytorch 1.8 and TorchVision 0.12.0 on Linux 18.04. Besides, CUDA support is needed to run the NLR module, and our CUDA version is 11.2. All of our experiments are run on a NVIDIA RTX 3090.

For the dataset, we adopt QuickDraw [6] which includes 345 categories of common objects, and each one contains 70k training, 2.5k validation and 2.5k testing samples. In this project, we

choose 25 categories with full samples as demanded in the project requirement from QuickDraw for the sketch classification problem. Note that original sketches in QuickDraw are described as vectorized sequences $S = \{s_i = (\Delta x_i, \Delta y_i, s_i)\}_{i=1..n}$. They are translated to 28×28 images as the CNN input. Besides, as described in SketchRNN [6], the 3-element stroke tuple is translated to $(\Delta x_i, \Delta y_i, p_1, p_2, p_3)$ as the RNN input (also the input to our transformer encoder). The last 3 elements represents a binary one-hot vector of 3 possible states. The first pen state, p_1, indicates that the pen is currently touching the paper, and that a line will be drawn connecting the next point with the current point. The second pen state, p_2, indicates that the pen will be lifted from the paper after the current point, and that no line will be drawn next. The final pen state, p_3, indicates that the drawing has ended, and subsequent points, including the current point, will not be rendered.

For the recognition models, we have discussed the general framework of all the baselines and our Trans2CNN in details in Section 2 and Section 3. Specifically, we adopt 7 baselines, including 2 RNN-based methods: SketchGRU and SketchLSTM (derived from SketchRNN [6]), 3 CNN-based methods: Sketch-a-Net [20], ResNet50 [7], and EfficientNet [13], and 2 RNN&CNN combined methods: SketchMate [17] and Sketch-R2CNN [10]. We also write a simple interactive GUI to do human recognition tests. See more details (e.g. hyperparameters) about them in Appendix A.

For the main recognition results, we run all our models on 5 different random seeds to calculate the mean and standard deviation. The models are trained on all 25×70k samples, validated and tested on the 25×2.5k validation/testing datasets, respectively. We report the testing accuracy achieved by the epoch that gives the highest accuracy on the validation dataset. Besides, as we find that the overfitting issue is serious on the sketch dataset (for example, the training accuracy of Sketch-R2CNN can achieve 94%, while the testing accuracy stops boosting at around 87%), we adopt the *early stopping* strategy to save time during training. Specifically, if the validation accuracy doesn't improve for 5 continuous epochs, we break and start training on the next random seed.

4.2 Main Results (Q1)

To answer the question Q1, we will discuss about the main experiment results in this section.

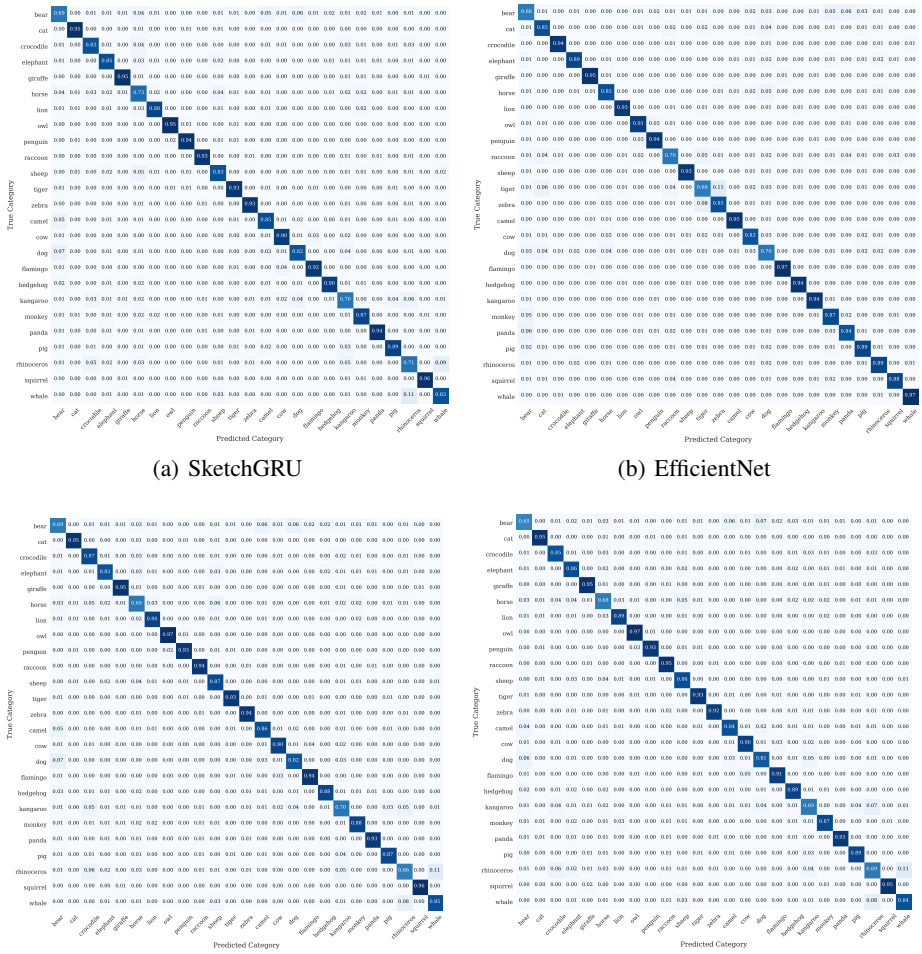
Table 1: Recognition results on QuickDraw. It records the mean accuracy and standard deviation on the testing dataset. For the binary dataset metrics: precision, recall, and F1 score, they are averaged over the 25 categories. The top-5 accuracy is non-sense for human tests as we only test on 5 categories. Our Trans2CNN model is composed of the transformer encoder, decoder and augmentation modules. Best results are in bold.

Model	Top-1 Accuracy↑	Top-5 Accuracy↑	Precision↑	Recall↑	F1 Score↑
SketchGRU [6]	0.8673±0.0034	0.9795±0.0028	0.8688±0.0031	0.8673±0.0034	0.8677±0.0033
SketchLSTM [6]	0.8583±0.0030	0.9778±0.0027	0.8571±0.0032	0.8583±0.0030	0.8573±0.0031
Sketch-a-Net [20]	0.8314±0.0025	0.9690±0.0023	0.8299±0.0027	0.8314±0.0025	0.8299±0.0027
ResNet50 [7]	0.8672±0.0020	0.9792±0.0020	0.8675±0.0022	0.8673±0.0020	0.8669±0.0021
EfficientNet [13]	0.8718±0.0022	0.9807±0.0022	0.8714±0.0027	0.8718±0.0022	0.8714±0.0024
SketchMate [17]	0.8729±0.0029	0.9827±0.0022	0.8730±0.0040	0.8730±0.0029	0.8726±0.0034
Sketch-R2CNN [10]	0.8751±0.0025	0.9811±0.0024	0.8752±0.0020	0.8750±0.0026	0.8747±0.0021
Human Test	0.8237±0.0876	/	0.8311±0.1045	0.8260±0.1475	0.8187±0.1015
Trans2CNN (ours)	**0.8839±0.0020**	**0.9837±0.0017**	**0.8836±0.0022**	**0.8839±0.0024**	**0.8834±0.0018**

Table 1 records the main recognition results of 5 commonly used metrics for the baselines and our Trans2CNN model. Generally, we can see that all the deep learning based models have achieved super-human average recognition performance, though the human test is of high variance. Generally, the RNN-based and CNN-based models have all achieved satisfying recognition accuracy with low variance. Note that the performance of Sketch-a-Net is relatively poor since we only adopt its network architecture, without its augmentation and fusion techniques in the original paper [21, 20].

Besides, we can also find in Table 1 that the RNN&CNN combined methods can really boost the recognition performance. However, their improvements are somewhat marginal. For example, Sketch-R2CNN only outperforms the EfficientNet backbone by approximately 0.3% (top-1 accuracy), which may not be enough to make up for the loss of computation resource it takes up. In comparison, our proposed **Trans2CNN outperforms all the baselines by a clear margin, across all the 5 metrics.**

Specifically, the top-1 accuracy of Trans2CNN outperforms Sketch-R2CNN by almost 1%. These results strongly support that our proposed Trans2CNN is strong at distinguishing sketches into the true class.



(a) SketchGRU (b) EfficientNet

(c) Sketch-R2CNN (d) Trans2CNN

Figure 4: Recognition confusion matrix for 4 different models on QuickDraw.

We also visualize the confusion matrixes to have more intuition about models' recognition performance. Typically, we can see from Figure 4(b) that EfficientNet tends to misclassify tigers as zebras. This result is consistent with the property of CNNs, as they prone to capture texture and structural patterns to recognize images (tigers and zebras all have speckles on the body). Besides, we find all the four models can't perform well in distinguishing between rhinoceros and whales. This may because that these two kinds of animals are rarely seen in daily lives so they are rather difficult to be sketched in a short time. For Trans2CNN, we can see in Figure 4(d) that it prones to misclassify bears as dogs, which is in line with our common sense as they do look like each other. We randomly visualize some misclassified samples in Figure 5. As we can see, they are rather abstract and meaningless, which are also difficult to be recognized by human eyes. Therefore, we guess that a 90% top-1 accuracy may be the bottleneck on the testing dataset.

4.3 In-Depth Study (Q2)

To answer Question Q2, we do some advanced experiments in this section.

First, we conduct ablation study as shown in Table 2. As we can see from the table, removing either the augmentation or the reconstruction module will cause a performance drop obviously, validating

Figure 5: Some misclassified samples of Trans2CNN. In the title "A"→"B", A and B denotes the ground truth and misclassified class, respectively.

Table 2: Ablation study results on QuickDraw. It records the mean accuracy and standard deviation on the testing dataset. The short term "aug" and "recon" denote the augmentation and reconstruction module, respectively.

Model	Top-1 Accuracy↑	Top-5 Accuracy↑	Precision↑	Recall↑	F1 Score↑
Trans2CNN w/o aug&recon	0.8799±0.0020	0.9806±0.0017	0.8794±0.0019	0.8799±0.0017	0.8794±0.0017
Trans2CNN w/o aug	0.8819±0.0037	0.9812±0.0016	0.8818±0.0031	0.8814±0.0037	0.8818±0.0030
Trans2CNN w/o recon	0.8807±0.0027	0.9806±0.0016	0.8801±0.0011	0.8804±0.0027	0.8808±0.0020
Trans2CNN	**0.8839±0.0020**	**0.9837±0.0017**	**0.8836±0.0022**	**0.8839±0.0024**	**0.8834±0.0018**

their capability of promoting recognition accuracy. In comparison, the reconstruction module is more important to achieve a high recognition accuracy. Notably, the "Trans2CNN w/o aug&recon" model still has a 0.5% accuracy improvement over Sketch-R2CNN, proving that the transformer encoder can learn better temporal information owing to its self-attention mechanism. Besides, we find that the augmentation module is important for mitigating overfitting. Specifically, the training and testing accuracy gap reduced from 3% to 1% during training after adding the augmentation module.

Second, we test our model's robustness by recognizing incomplete and deformed sketches. As illustrated in Figure 6, the performance of the three models all drop obviously w.r.t. larger deformed and incomplete ratio. On both experiments, Trans2CNN consistently outperforms the baselines by a clear margin. Especially when tested on the deformed sketches (left in Figure 6), the accuracy gap between Trans2CNN and Sketch-R2CNN can be up to 10% when the scale factor and rotation angle are set as 0.6 and 60°, respectively. These results strongly support that our model is more robust than the previous works, which may be benefited from its transformer architecture and the two data augmentation strategies.

Finally, we visualize the feature embeddings to compare the representation learning power of different models. Specifically, we select the feature embedding before the final FC layer for

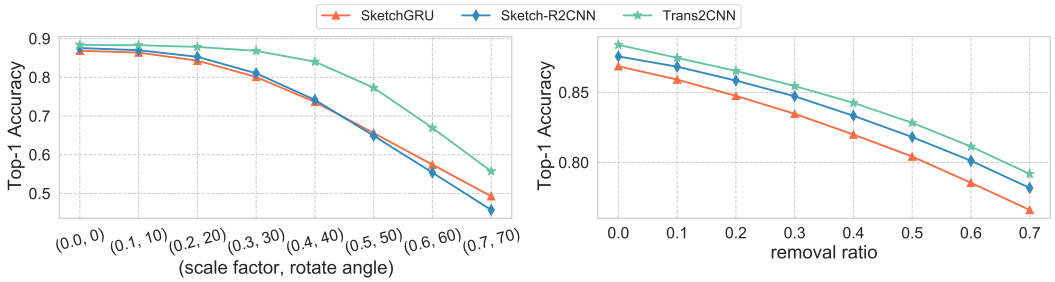
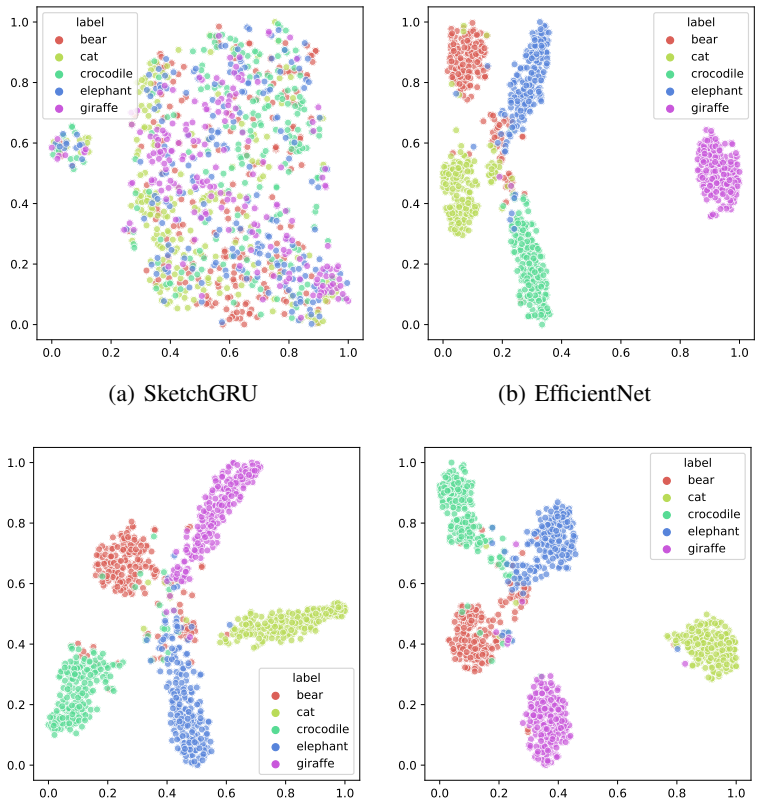


Figure 6: Robustness comparison for 3 different models. The left figure denotes the experiments on deformed sketches, while the right one shows the results on incomplete ones.



(a) SketchGRU (b) EfficientNet

(c) Sketch-R2CNN (d) Trans2CNN

Figure 7: TSNE Embedding visualization for 4 different models on QuickDraw.

fair comparison, using tSNE [14] to embed and cluster them on a 2D space. As shown in Figure 7, EfficientNet, Sketch-R2CNN, and Trans2CNN have all achieved satisfying embedding performance. Figure 7(a) shows that SketchGRU doesn't perform well in feature embedding, which may because that the hidden states from two directions have not been fused well before the final FC layer. In comparison, the embeddings are best separated out by our Trans2CNN, with crocodile and giraffe embeddings far away from the center.

5 Conclusion

In this work, we propose Trans2CNN, a Transformer-Rasterization-CNN end-to-end network architecture to deal with the limited temporal extent of RNNs, as well as capture information from both vector and pixel format sketches. We also develop data augmentation strategies based on affine transformation and stroke removal to mitigate the overfitting issue on the sketch dataset. Besides, we introduce a transformer decoder, which further boosts the model's recognition performance.

We implement RNN-based, CNN-based, RNN&CNN combined, and human testing baselines and conduct extensive experiments on the QuickDraw dataset. Experiment results show that Trans2CNN outperforms all the baselines by a notable margin on sketch recognition tasks. We also study models' performance on recognizing incomplete and deformed sketches, which shows that Trans2CNN is robust to these kinds of perturbations. The ablation study proves the significance of each module in our proposed framework.

One defect of Trans2CNN is that though it runs faster than Sketch-R2CNN [10] and SketchMate [17] during our experiments, it still owns high time and space overhead. One possible improvement is to incorporate with advanced transformer variants such as Linformer [16] or Performer [3] to do self-attention operation with linear complexity. Another important future direction is to tackle the

overfitting issue. We believe that pretraining on large datasets and preprocessing operations such as filtering out dirty sketches can help, which we leave for future work. It is also an interesting direction to conduct further generation and interpolation experiments based on our transformer decoder.

Contribution and Acknowledgements

In this work, Zenan Li is responsible for preliminary paper survey, idea formulation and code implementation, and paper writing. Qi Liu is responsible for the implementation of the baselines, experiments and visualization. Our contributions are approximately equivalent, so we recommend Zenan Li 60%, Qi Liu 40% for grading. Check our code at `https://github.com/Emiyalzn/CS420-Sketch-Recognition`.

At the end of this paper, we would like to thank every team member for our dedication to this project. We two have really cooperated well to complete this work efficiently. Second, we sincerely thank Prof.Tu for setting up this interesting project and his excellent teaching. Thanks to TA Zhao's valuable advice on our project. Finally, thank you for reading till this end, hope this paper can inspire you to some extent.

References

[1] M. Arjovsky, L. Bottou, I. Gulrajani, and D. Lopez-Paz. Invariant risk minimization. *arXiv preprint arXiv:1907.02893*, 2019.

[2] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[3] K. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlós, P. Hawkins, J. Davis, A. Mohiuddin, L. Kaiser, D. Belanger, L. J. Colwell, and A. Weller. Rethinking attention with performers. *CoRR*, abs/2009.14794, 2020.

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[5] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[6] D. Ha and D. Eck. A neural representation of sketch drawings. *arXiv preprint arXiv:1704.03477*, 2017.

[7] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[9] A. Lamb, S. Ozair, V. Verma, and D. Ha. Sketchtransfer: A new dataset for exploring detail-invariance and the abstractions learned by deep networks. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 963–972, 2020.

[10] L. Li, C. Zou, Y. Zheng, Q. Su, H. Fu, and C.-L. Tai. Sketch-r2cnn: An rnn-rasterization-cnn architecture for vector sketch recognition. *IEEE transactions on visualization and computer graphics*, 27(9):3745–3754, 2020.

[11] F. Liu, X. Deng, Y.-K. Lai, Y.-J. Liu, C. Ma, and H. Wang. Sketchgan: Joint sketch completion and recognition with generative adversarial network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5830–5839, 2019.

[12] P. Sangkloy, N. Burnell, C. Ham, and J. Hays. The sketchy database: learning to retrieve badly drawn bunnies. *ACM Transactions on Graphics (TOG)*, 35(4):1–12, 2016.

[13] M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

[14] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[16] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma. Linformer: Self-attention with linear complexity. *CoRR*, abs/2006.04768, 2020.

[17] P. Xu, Y. Huang, T. Yuan, K. Pang, Y.-Z. Song, T. Xiang, T. M. Hospedales, Z. Ma, and J. Guo. Sketchmate: Deep hashing for million-scale human sketch retrieval. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8090–8098, 2018.

[18] L. Yang, K. Pang, H. Zhang, and Y.-Z. Song. Sketchaa: Abstract representation for abstract sketches. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10097–10106, 2021.

[19] Q. Yu, F. Liu, Y.-Z. Song, T. Xiang, T. M. Hospedales, and C.-C. Loy. Sketch me that shoe. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 799–807, 2016.

[20] Q. Yu, Y. Yang, F. Liu, Y.-Z. Song, T. Xiang, and T. M. Hospedales. Sketch-a-net: A deep neural network that beats humans. *International journal of computer vision*, 122(3):411–425, 2017.

[21] Q. Yu, Y. Yang, Y.-Z. Song, T. Xiang, and T. Hospedales. Sketch-a-net that beats humans. *arXiv preprint arXiv:1501.07873*, 2015.

[22] S. Zang, S. Tu, and L. Xu. Controllable stroke-based sketch synthesis from a self-organized latent space. *Neural Networks*, 137:138–150, 2021.

A Implementation Details

We present implementation details of the recognition models in this section for reproducibility.

A.1 Details for RNN-Based Models

For the RNN-based models, the input is 5-element tuple $(\Delta x_i, \Delta y_i, p_1, p_2, p_3)$ sequence. To enable mini-batch training, we pad the sequence to length 226, which is the max length in the dataset. Next, the training details and hyperparameter settings are listed as follows:

- The Cross Entropy loss is adopted for classification. We use mini-batch gradient descent to optimize the model, and the batch size is set as 128. The Adam optimizer is selected for optimization.
- We adopt a 2-layer, bi-direction architecture with a hidden size of 256 for both GRU [2] and LSTM [5] models, the final hidden state in each direction will be concatenated and output as the RNN embedding. The embedding will be fed into an FC layer for final classification.
- The dropout rate is set as 0.1, the learning rate is set as 0.001. We do not use weight decay regularization during training.
- The max training epoch is set as 20.

A.2 Details for CNN-Based Models

The input to CNN-based models is pre-translated 28×28 images. The images will be rescaled to size of 224×224 during training, and they will be normalized w.r.t. their mean and standard deviations. Next, the training details and hyperparameter settings are listed as follows:

- The Cross Entropy loss is adopted for classification. We use mini-batch gradient descent to optimize the model, and the batch size is set as 48. The Adam optimizer is selected for optimization.

- The architectures of ResNet50 [7], EfficientNet [13], and Sketch-a-Net [20] are the same as their original paper. Typically, for ResNet50 and EfficientNet, we use their ImageNet [8] pretrained version in the TorchVision. The CNN output will be fed into an FC layer for final classification.
- No dropout and weight decay are adopted during training, and the learning rate is set as 0.0001.
- The max training epoch is set as 20.

A.3 Details for RNN&CNN Combined Models

For both SketchMate [17] and Sketch-R2CNN [10], the vector format input is 3-element (rather than 5) tuple $(\Delta x_i, \Delta y_i, s_i)$ sequence (for RNN) to speed up training. SketchMate also takes the pre-translated 28×28 images as additional input to the CNN module. For Sketch-R2CNN, the sequence will also be translated into point coordinate sequence (x_i, y_i, s_i) in the collate function for rasterization. Next, the training details and hyperparameter settings are listed as follows:

- The Cross Entropy loss is adopted for classification. We use mini-batch gradient descent to optimize the model, and the batch size is set as 64. The Adam optimizer is selected for optimization.
- For both SketchMate and Sketch-R2CNN, the RNN backbone is chosen as LSTM, while the CNN backbone is chosen as EfficientNet. Their architectures are the same as that in the RNN and CNN based models. For Sketch-R2CNN, we use 8-channel point feature maps for the CNN. The final output will be fed into an FC layer for final classification.
- No dropout and weight decay are adopted during training, and the learning rate is set as 0.001.
- The max training epoch is set as 20.

A.4 Details for Trans2CNN

The input to Trans2CNN is 5-element tuple $(\Delta x_i, \Delta y_i, p_1, p_2, p_3)$ sequence (for transformer), which will also be translated into point coordinate sequence in the collate function for rasterization. Next, the training details and hyperparameter settings are listed as follows:

- The Cross Entropy loss is adopted for classification. Additionally, we turn on the reconstruction loss as described in Section 3 if we use the transformer decoder. And the reconstruction weight λ is set as 1.0. We use mini-batch gradient descent to optimize the model, and the batch size is set as 128. The Adam optimizer is selected for optimization.
- The CNN backbone is chosen as EfficientNet with the same architecture as in the CNN-based models.
- For the transformer architecture, both the encoder and decoder are comprised of 4 blocks. We use 8 attention heads with a total hidden dimension of 128 for the MHA blocks. The hidden size of the FFN block is set as 512 with ReLU activation. Besides, we use 8-channel point feature maps for the CNN. The final output will be fed into an FC layer for classification.
- The dropout rate is set as 0.1 for both transformer encoder and decoder. The learning rate is set as 0.001. No weight decay regularization is adopted during training.
- For the data augmentation strategies, we set $\alpha = 2$, and $\beta = 0.5$, the default stroke removal probability is 0.15. For the affine transformation, the default scale factor is 0.1, while the default rotation angle is set as 20°.
- The max training epoch is set as 20.

A.5 Details for Human Test

We write a simple interactive UI (which can be found in our github repository) in Python to conduct human recognition tests. Specifically, we invite 3 irrelevant participants to test on 5 categories (100 sketches each): bear, camel, cat, cow, and crocodile. We compute the average results and record them in Table 1.