Motion-Based Handwriting Recognition and Word Reconstruction

Junshen Kevin Chen, Wanze Xie, Yutong He | {jkc1, wanze.xie, kellyyhe}@stanford.edu

Video Presentation Link: https://youtu.be/sYK3znT5nI

Overview
This project attempts motion-based handwriting recognition to explore an alternative to a vision-based approach, for various use cases where there is no convenient surface to write on (e.g. VR).

For this project, we build the collection hardware with Arduino and motion sensor, collect our own original dataset of written individual letters as training data, and written words as test data. We solve this problem by building a word reconstruction pipeline that splits a given sequence to segments, search through combinations of segments for optimal trajectories, then auto-correct to produce a production word. Finally, we experiment with domain adaptation to handle unseen data distribution.

Data Augmentation
With a small dataset of individually written letter, we augment the training set to approach the distribution continuous writing:
- shape modification adding noise, stretching, rotating
- prepend / append frames from other letter samples
- trimming off small number of frames off current sample

We also create non-class labeled samples by taking a combination of noise and partial letter samples.

Word Reconstruction Pipeline
The character classifier is an encoder-decoder, with a LSTM encoder that encodes frames of motion data into states, then a decoder that decodes sequences of characters. The auto-correct module uses a character classifier to predict the most likely character at each step. The trajectory search module searches through combinations of segments for optimal trajectories, then auto-corrects to produce a production word. The word reconstruction model learns to produce the most accurate prediction, and contributes to performance by introducing noise and more false predictions.

Character Classifier
The character classifier is an encoder-decoder, with a LSTM encoder that encodes frames of motion data into states, then a decoder that decodes sequences of characters. The auto-correct module uses a character classifier to predict the most likely character at each step. The trajectory search module searches through combinations of segments for optimal trajectories, then auto-corrects to produce a production word. The word reconstruction model learns to produce the most accurate prediction, and contributes to performance by introducing noise and more false predictions.

Trajectory Search
We search through all candidates of all segments to form trajectories, combination of candidates through the sequence. At each split-point, we keep the top ranking sub-trajectories ("beams") by average logit over constituting candidates. Trajectory search is a dynamic programming problem, where optimal sub-trajectories starting at a split-point must be optimal for all sub-trajectory arriving at this split-point.

Auto Correction
We implemented our auto-correction model based on Symmetric Delete spelling correction (SymSpell) algorithm. However, instead of directly auto-correcting the Top 1 result from the Trajectory Search, we auto-correct all available predictions from word search and use the confidence $c_i$ from the word search, the frequency $f_i$ and edit distance $d_i$ from auto-correction lookup result, to pick the final predicted word. We experiment with four different techniques, where we finalize word based on MaxVote (max # of repeated occurrence), SumConf (max sum confidence $c_i$), Division Combination (max sum of $\alpha_i = \frac{c_i}{f_i}$), or Power Combination (max sum of $\alpha_i = \frac{c_i}{f_i^{1/\gamma}}$).

Domain Adaptation
To compensate the lack of generalized dataset, we use domain adaptation to transfer the knowledge and feature extractor the model learned from the limited dataset into any new user of the device. We train the model with the following loss function:

$$L_{da}(\theta_{da}) = \sum_{i=1}^{n_{da}} \sum_{c} \text{logit}(P(c|y^{(i)}(\theta_{da})))$$

$$L_{dum}(\theta_{dum}) = \sum_{i=1}^{n_{dum}} \text{logit}(P(c|y^{(i)}(\theta_{dum}))) + (1 - \text{logit}(P(c|y^{(i)}(\theta_{dum}))))$$

$$L(\theta_{da}, \theta_{dum}) = L_{da}(\theta_{da}) - L_{dum}(\theta_{dum})$$

Future Work
- Apply multi-threading to parallelize trajectory search
- Improve data collection strategy, and collect more, cleaner data to train a better base model
- Experiment with other auto-correction and domain adaptation techniques

Experiment Results
Character classifier hyperparameter search: we perform a random search to find the best performing model has 8 LSTM layers of 275 dimensional hidden states, and 86 FC hidden units. Word reconstruction hyperparameter search: we perform a grid search on number of splits per letter ($G$), and number of beams for trajectory search ($K$), and evaluate on average edit distance.

Out-of-Domain Character Classification:

<table>
<thead>
<tr>
<th></th>
<th>Train Acc</th>
<th>Dev Acc</th>
<th>Test Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>-</td>
<td>-</td>
<td>-0.13725</td>
</tr>
<tr>
<td>Fine-Tuning</td>
<td>0.96323</td>
<td>0.49057</td>
<td>0.49020</td>
</tr>
<tr>
<td>Domain Adaptation</td>
<td>0.99185</td>
<td>0.64780</td>
<td>0.70688</td>
</tr>
</tbody>
</table>

Analysis and Discussion
- Character classifier with a complex LSTM encoder and simple FC decoder works better, to encode complex features from raw data, while avoiding overfitting.
- Trajectory search with a higher number of beams produces higher accuracy by producing more "confident options" to auto-correct, while a higher number of split worsens performance by introducing noise and more false predictions.
- Auto-correction with the Direct Combination (D.C.) technique is able to produce the most accurate prediction, and contributes to our final word reconstruction pipeline's accuracy of 0.888.
- Domain adaptation is able to boost accuracy significantly for the character classifier, but is unable to improve sufficiently for accurate trajectory search. With a base model trained with better data, DA is a data-efficient way to apply the model to real world.

Experiment Results
Character classifier hyperparameter search: we perform a random search to find the best performing model has 8 LSTM layers of 275 dimensional hidden states, and 86 FC hidden units. Word reconstruction hyperparameter search: we perform a grid search on number of splits per letter ($G$), and number of beams for trajectory search ($K$), and evaluate on average edit distance.

Auto-correction:

<table>
<thead>
<tr>
<th></th>
<th>Top 1</th>
<th>MaxVote</th>
<th>SumConf</th>
<th>D.C.</th>
<th>P.C.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOD</td>
<td>0.299</td>
<td>0.208</td>
<td>0.299</td>
<td>0.292</td>
<td>0.260</td>
</tr>
<tr>
<td>ID-1</td>
<td>0.674</td>
<td>0.640</td>
<td>0.685</td>
<td>0.832</td>
<td>0.787</td>
</tr>
<tr>
<td>ID-2</td>
<td>0.678</td>
<td>0.711</td>
<td>0.733</td>
<td>0.944</td>
<td>0.900</td>
</tr>
<tr>
<td>ID-avg</td>
<td>0.676</td>
<td>0.676</td>
<td>0.710</td>
<td>0.888</td>
<td>0.844</td>
</tr>
</tbody>
</table>

Out-of-Domain Character Classification:

<table>
<thead>
<tr>
<th></th>
<th>Train Acc</th>
<th>Dev Acc</th>
<th>Test Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>-</td>
<td>-</td>
<td>-0.13725</td>
</tr>
<tr>
<td>Fine-Tuning</td>
<td>0.96323</td>
<td>0.49057</td>
<td>0.49020</td>
</tr>
<tr>
<td>Domain Adaptation</td>
<td>0.99185</td>
<td>0.64780</td>
<td>0.70688</td>
</tr>
</tbody>
</table>