NYCU Pattern Recognition Final Project

411551005 徐浩哲

- Environment details
  - CPU: Intel® Core™ i7-6700 4.0GHz * 4
  - GPU: GTX 1060 6 GB
  - Operation System: Windows 10
  - Development tools: Anaconda, **Python 3.9.12**
- Implementation details
  - Model architecture

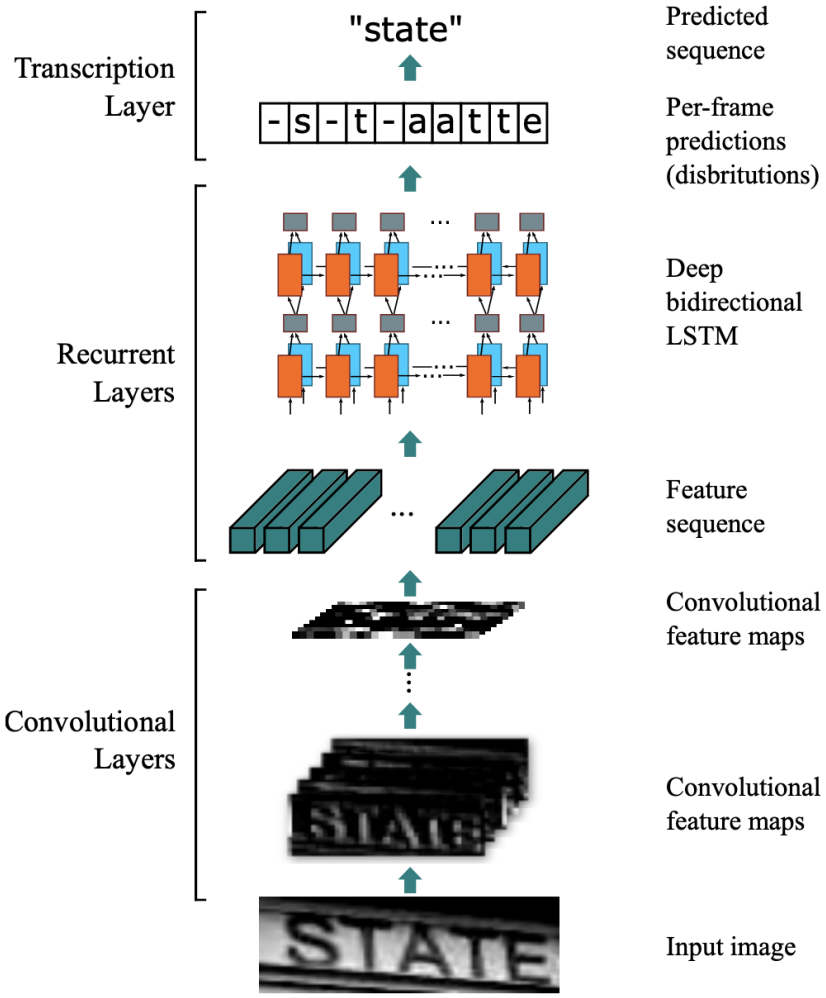
    Convolutional Recurrent Neural Network (CRNN) contain three parts. First is **Convolutional layers**, which extract a feature sequence from the input image follow by **Recurrent layers** which predict a label distribution for each frame. For the end is **Transcription layer**, which translates the per-frame predictions into the final label sequence.

    1. **Convolutional layers**, from a standard VGG-11 model (fully-connected layers are removed). Such component is used to extract a sequential feature representation from an input image. Pass the sequential feature to next part.
    2. **Recurrent layers**, a traditional RNN unit comprises a self-connected hidden layer situated between its input and output layers. Upon receiving a frame x_t in the sequence, it updates its internal state h_t using a non-linear function that takes both the current input x_t and the previous state ht−1 as inputs: $h_t = g(x_t, h_t - 1)$. Subsequently, the prediction yt is generated based on ht. This mechanism allows the RNN unit to capture and utilize past contexts $\{x_t\ 0\}\ t_0<t$ for prediction. However, traditional RNN units are prone to the vanishing gradient problem, which restricts the range of contexts they can store and poses challenges during training. To overcome this limitation, the Long-Short Term Memory (LSTM) was developed as a specialized type of RNN unit. An LSTM consists of a memory cell and three multiplicative gates: the input, output, and forget gates.
    3. **Transcription layer**, Transcription refers to the process of converting the per-frame predictions generated by an RNN into a sequence of labels. Mathematically, transcription involves identifying the label sequence that has the highest probability given the per-frame predictions. In practice, there are two modes of transcription: lexicon-free and lexicon-based.

A lexicon refers to a predefined set of label sequences that the prediction process is constrained to, similar to a spell-checking dictionary. In lexicon-free mode, predictions are made without utilizing any lexicon. This means that the RNN generates label sequences solely based on the per-frame predictions without any external constraints.

On the other hand, in lexicon-based mode, predictions are made by selecting the label sequence that has the highest probability among all the possible sequences within the lexicon. The lexicon serves as a reference to guide the prediction process, and the RNN chooses the label sequence that maximizes the probability according to the available lexicon.

Overall, transcription involves finding the most probable label sequence based on the per-frame predictions, and it can be performed either in a lexicon-free mode or a lexicon-based mode depending on whether a predefined set of labels is considered during the prediction process.



- Hyperparameters
  1. Epoch = 1000
  2. Train and Validation ratio = 8:2
  3. Batch size = 512
  4. Optimizer: RMSprop
  5. Loss function: CTCloss (Connectionist Temporal Classification loss)
  6. Initial learning rate: 0.0005
- A detailed description of your experimental design, including the methodology and procedures employed in your study.

  In my endeavor to develop a more effective strategy for captcha recognition, I encountered some significant challenges pertaining to the variable length of captcha text. One of the traditional strategies in machine learning involves the use of a one-hot encoding scheme. However, this approach seems ill-suited for variable length data such as captcha text. The main issue lies in the fact that one-hot encoding can potentially lead to an insufficient number of samples for each category.

  To illustrate this issue, consider the case of two images with respective labels "7A" and "7Ajs". Using a one-hot encoding scheme would result in two entirely distinct categories, rather than a system that learns individual character features. This limitation was made evident through our initial implementation on a VGG-19 model, which yielded a disappointingly low accuracy of around 25%, essentially amounting to random guessing.

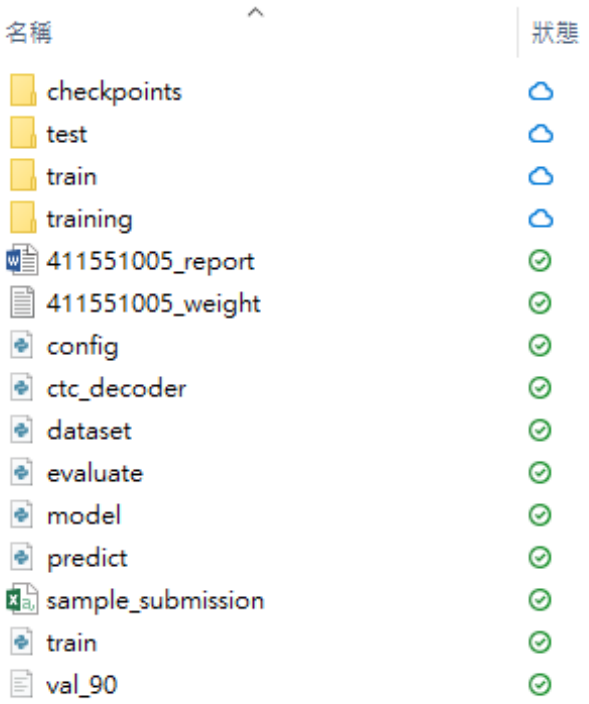
  This observation brought to light the human approach to captcha recognition, which is primarily a left-to-right process. However, the lack of fixed distances between characters presents a further obstacle in data preprocessing and character segmentation.

  Inspired by technical literature on captcha and license plate recognition https://arxiv.org/pdf/1507.05717.pdf, I identified Convolutional Recurrent Neural Network (CRNN) as a promising approach to overcome these challenges. This method utilizes the VGG model for feature extraction, followed by an LSTM network to analyze sequential data. Finally, it applies a Connectionist Temporal Classification (CTC) loss function to estimate the probability of character presence.

  This hybrid approach leverages the strengths of both CNN and RNN, potentially providing a robust solution for variable-length captcha recognition. I anticipate further exploration and optimization of this approach could substantially improve the current state of captcha recognition technologies.

- Compare with different method, ablation study, result analysis
  I compared the accuracy of general CNN and CRNN to 25% and 90% respectively on the validation data set. On the input image it was found that a width of 100 is better than the original width (32). And in the learning rate, try to set the learning rate scheduler, but try to halve the learning rate every 10, 50, 100, and 200 epochs. It seems that there is no way to get better results.
- How to implementation

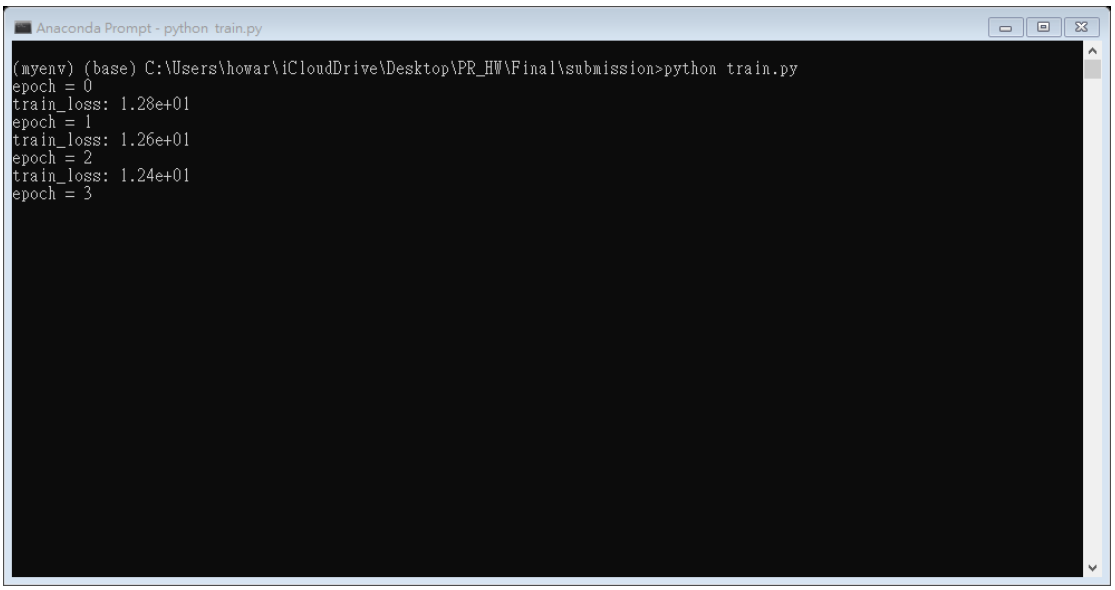
1. Move everything from the training folder to the previous level, as shown in the image below.



2. Run:

   python train.py

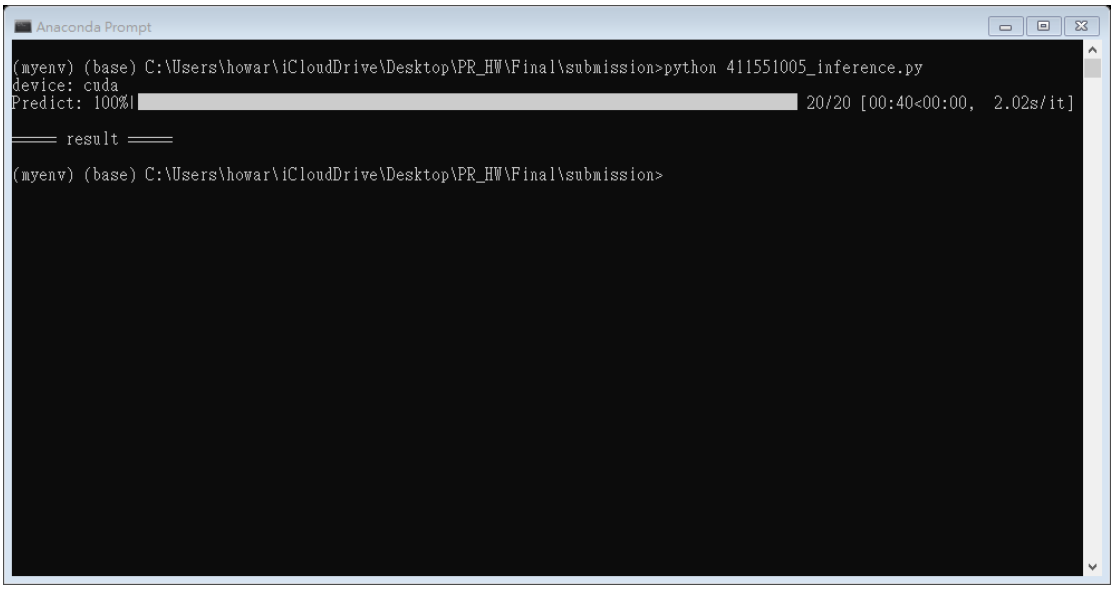
   For training.



Run:

python 411551005_inference.py

For inference, it will generate a “output1.csv” that is my prediction for Kaggle.



If you want to run weight file form checkpoints:

Python 411551005_inference.py -m ./checkpoints/filename