CS230

Automatic Code Generation Project Proposal

Lizi Ottens
Department of Computer Science
Stanford University
lottens@stanford.edu

Luis Perez
Department of Computer Science
Stanford University
luis0@stanford.edu

Sudharshan Viswanathan
Department of Computer Science
Stanford University
viswans@stanford.edu

Abstract

We propose developming a machine learning system for automatic code generation given a query string of text (eg, purpose of the program). The progress completed lies around three main axis: (1) refinement of problem statement, (2) in-depth analysis of CodeSearchNet dataset and (3) training of multiple exploratory baselines successfully.

Code used for baseline training and data-preprocessing is available on GitHub [1]

1 Problem Statement

Automating even small parts of software development is an interesting research area [1] of machine learning that can save time for countless software engineers and resources for technology companies to invest into solving other more challenging problems. Automating code generation can take on many forms, from auto-completing lines of source code to generating lines of source code from comments, generating source code from UI images, or generating unit tests from source code. In this project, we aim to take incomplete lines of code as input and generate code to complete it.

2 Background

A primary challenge in code generation is that it's still an active area of research, with many possible solutions and ongoing investigation [2]. State of the art solutions have not yet come close to automating basic tasks software engineers perform on a daily basis.

Other challenges include restricted language domains, complexity in evaluation results, etc. Specifically, while pre-trained models are trained on free-form language data, programming languages often utilize non-natural variable names, function names, and syntax with more structure [2]. Work in this area has focused on creating more structured models that take advantage of specific architectures [3]. Other approaches involve restricting the output of the model to a CFG or DSL [4]. A code generation model's output must adhere to a very specific form in order to be syntactically correct.

Compilers/interpreters could be leveraged to provide a better form of feedback and hence can be used to create labelled data, but it is left to be seen how effectively we can leverage this feedback to aid the learning process.

[1] See shared repository located at https://github.com/kandluis/cs230.

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

3 Methodology

In this section we explain our methodology for multiple experiments and baselines proposed, as well as details on the training data and distribution.

3.1 Data Analysis

In this project, we are leveraging the CodeSearchNet dataset [5]. The dataset consists of 2 million (comment, code) pairs from open source libraries, ranging in languages from Python, Javascript, and PHP. Median code-length consists of 60-100 text tokens, with 95% code-length of up to 350 tokens. Median documentation length consists of 10 text tokens. The distributions of methods and (comment, code) pairs across programming language are visualized in Figure 1.

3.2 Baselines

Figure 2 explains the general architecture of the baseline models from the CodeSearchNet task. We successfully trained and evaluated two baselines: Neural-Bag-Of-Words and an RNN-based baseline. See Section 4.

Generally speaking, the baselines models take as input examples of (comments, code) pairs and learn to retrieve a specific code snippet. Each programming language has its own encoder network (see three columns to the right in Figure 2, which are tasked with encoding a set of candidate code snippets. They are then combined through a dot product operation with the embedding generated by the query (docstring) encoder to produce a matrix comparison.

The matrix diagonal serves as the scores of each query doc string/code snippet. Through this methodology, these baseline models are able to extract meaningful information and learn a joint distribution over the query and comment pairs. We train these models as a baseline since we believe they will be useful in the downstream task of code generation. The models are trained on the following loss function:

$$-\frac{1}{N}\sum_{i}\log\left(\frac{\exp(E_c(\mathbf{c}_i^T)E_q(\mathbf{d}_i))}{\sum_j \exp(E_c(\mathbf{c}_j^T)E_q(\mathbf{d}_j))}\right) \tag{1}$$

4 Results

CodeSearchNet provides a good starting point as we are able to train different models on the input code streams. We trained a simple LSTM model as well as a neural bag of words model on a combination of all the available (code, documentation) pairs.

The Neural Bag of Words and LSTM CodeSearchNet baselines both report metrics in the same fashion. Below, we show the training curves, which correspond to the loss in Equation (1).

Additionally, given that the baselines models for CodeSearchNet focus on code snippet retrieval, we also report the achieved mean reciprocal rank. The MRR is a statistic measure for evaluating any process that produces a list of possible responses to a sample of queries, ordered by probability of correctness. The reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct answer: 1 for first place, $\frac{1}{2}$ for second place, $\frac{1}{3}$ for third place and so on. The mean reciprocal rank is the average of the reciprocal ranks of results for a sample of queries, as in Equation (2).

$$\text{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{\text{rank}_i} \tag{2}$$

4.1 Neural Bag of Words Baselines

This baseline consists of a simple encoder architecture which takes as input bag-of-words representation of the code and using a single neural network encodes these token representation into an

embedding [5]. This baseline actually performs the best, achieving the lowest overall training and validation losses (see Figure 3) as well as the highest MRR on the validation set (See Figure 4).

4.2 Bi-directional RNN Model

In this model, we employ the GRU cell [6] to summarize the input sequence. This baseline performs significantly worse, suffering from what appears to be obvious over-fitting. In Figure 5, we can see that while the training loss appears to plateau, the validation loss begins quickly climbing. While this behavior does not appear to affect the overall MRR achieved on the validation set, it's still clear that the model performs worse than the bag of words baseline as per Figure 6.

4.3 Code Generation with Char-RNN

As both of the above baselines focus on understanding and extracting useful embedding for our overall task, our primary baseline consists of a straight-forward sequence-to-sequence model. Given that code typically does not consists of English words (but instead can have quite a varied syntax), our baseline model is a model which uses character level embedding, so it is character aware [7].

Due to computational constraints, we train only on the Python subset of the data and only on 10% of the total data available. For the char-rnn model [7], this corresponds to around 50MB of raw text, or 78,357,395 characters with 1,618 distinct symbols. Figure 7 shows the training and validation losses on the model. The loss is simply a softmax loss on the 1,618 characters for a sequence of length 128 (the model is trained on sequences of length 128 by default). Figure 8 shows the perplexity, or the amount of meaninful information encoder.

We include a sample generated from the best performing model for reference.

```
Downloads Dailymotion videos by URL. (x or pillares (if macks), style as a bool to you extner met
       to oio instance of that Seir to be two categorical String to mandation :attr:‘Columnserver
                zr, j)
        dimensed_source
        also = ‘‘axis‘‘. Element. This repr_dures a substimcle of

        """
        code = item
    return self.filename
            restroxig, self.get_channels():
    """Get the function as firmap_list {1} and a :attracted_coordint: value of Time
    end_aes:
            target_ext = int(cmd_dict):
            # In data applicate horinad(string): the channel for decoded collfile
            Runnee network number of element. What you’s associates py ch of List does request. (

        ‘‘index‘‘.
def vert_event_notify_channel(self, infna=1)
                    elif trying.format(
        comps + ’,’ % (random) + 1,
            H.data:
            if list + lookbing.get(’get_dict.get’]}
```

5 Next Steps

For our next steps, we plan on working to figure out how to make use of the CodeSearchNet baselines with our generative model. It's not yet clear to use how to do this, or how to merge the two. However, at the minimum we believe we can improve the performance of the baseline models by tuning some of the hyperparameters, as it's clear that for our dataset there appears to be overfitting.

CodeBERT [8] is a transformer method to generate documentation from code and builds on top of the CodeSearchNet paper. It improves the results compared to the NBow and LSTM results that we have

discussed above. On the other hand, the above problems mainly focus on "searching/ranking code given multiple snippets and a NL query" and also the work focuses on generating documentation.

The Char-RNN work that we were able to test out also shows the shortcomings that we had expected, where the generated "code" is not in a compiler-acceptable form. The CodeSearchNet attempt starts with translating code to a tree structure before applying Seq2Seq models on top of it, and we can hypothesize that a Tree modelling would be a necessary condition for modelling and eventually generating code.

In that spirit, TreeGAN [9] focuses on building a tree (where a tree itself is represented as a sequence) and a GAN is used to generate the code sequences with Policy Gradient used to tackle the problem of finding a differentiable cost function in such instances.

As next steps, we propose to investigate and attempt to run the GAN models (either TreeGAN or SeqGAN) and verify the claims, and in the case that we are able to verify that they indeed work we can proceed to extend to the case of generating unit tests.

References

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. A survey of machine learning for big code and naturalness. *CoRR*, abs/1709.06182, 2017.

[2] Yasir Hussain, Zhiqiu Huang, Senzhang Wang, and Yu Zhou. Codegru: Context-aware deep learning with gated recurrent unit for source code modeling. *CoRR*, abs/1903.00884, 2019.

[3] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. Treegen: A tree-based transformer architecture for code generation, 2019.

[4] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. A grammar-based structural CNN decoder for code generation. *CoRR*, abs/1811.06837, 2018.

[5] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2019.

[6] KyungHyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259, 2014.

[7] Yoon Kim, Yacine Jernite, David A. Sontag, and Alexander M. Rush. Character-aware neural language models. *CoRR*, abs/1508.06615, 2015.

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[9] Xinyue Liu, Xiangnan Kong, Lei Liu, and Kuorong Chiang. Treegan: Syntax-aware sequence generation with generative adversarial networks. *CoRR*, abs/1808.07582, 2018.

Appendix and Figures

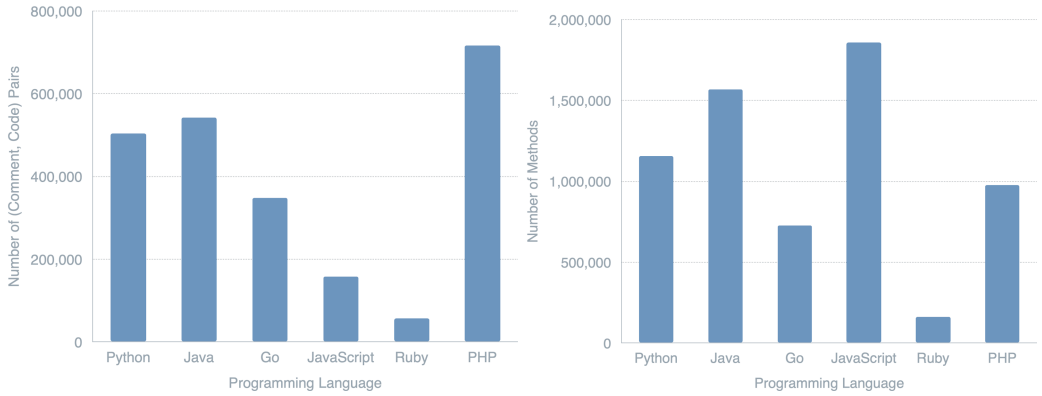

Figure 1: Histogram of the the number of (comment, code) pairs available in our dataset, as well as the number of unique function methods for each language.

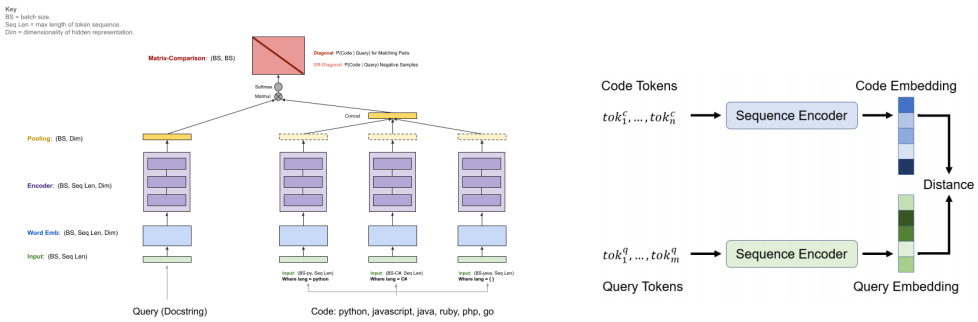

Figure 2: General CodeSearchNet architecture for all of our baselines. Each language is processed through different encoder mechanisms. The query encoder is shared (an NLP encoder), and the purpose of the CodeSearchNet tasks is to retrieve the most relevant code snippets subject to the natural language query.

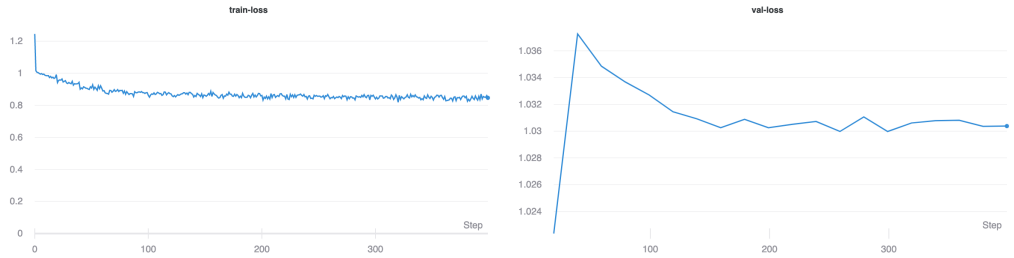

Figure 3: Training and Validation losses for the Neural Bag of Words model in CodeSearchNet.

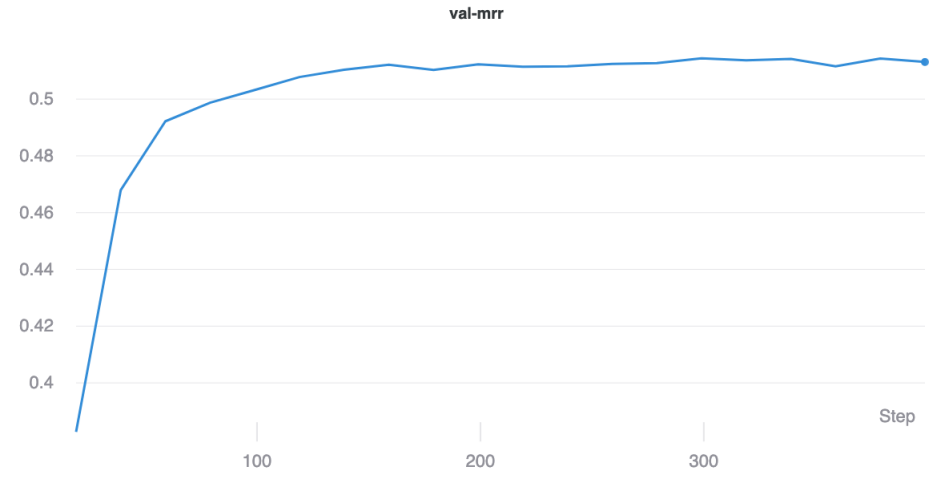

Figure 4: MRR on validation set for the baseline neural bag of words model in the CodeSearchNet Challenge.

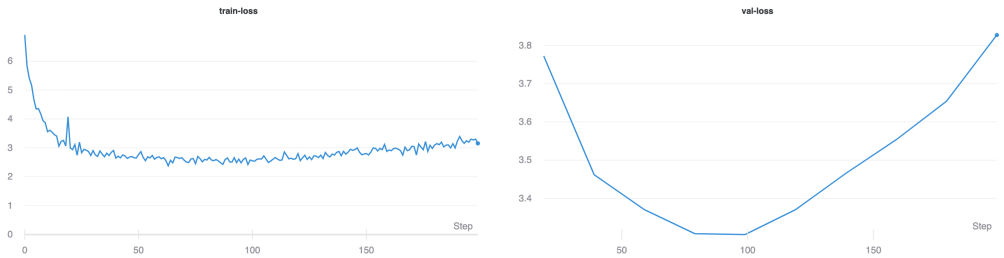

Figure 5: Training and Validation losses for the RNN model in CodeSearchNet.

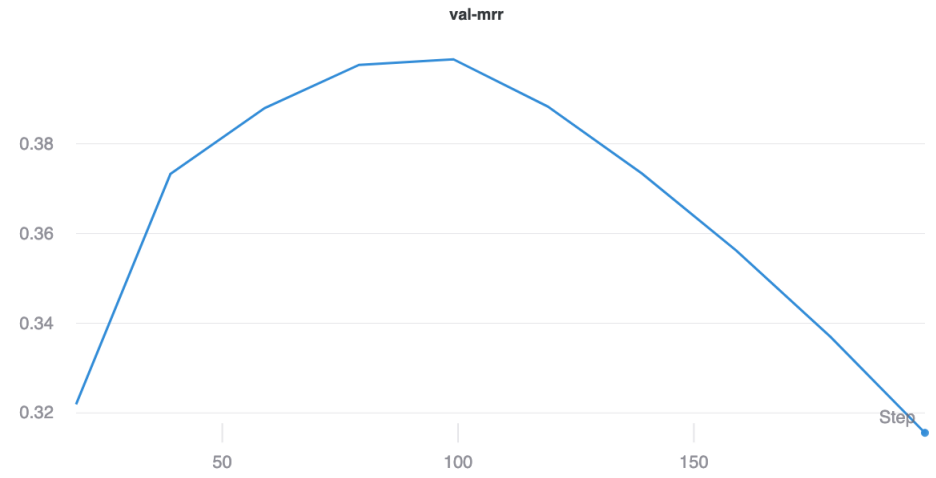

Figure 6: MRR on validation set for the baseline RNN in the CodeSearchNet Challenge.

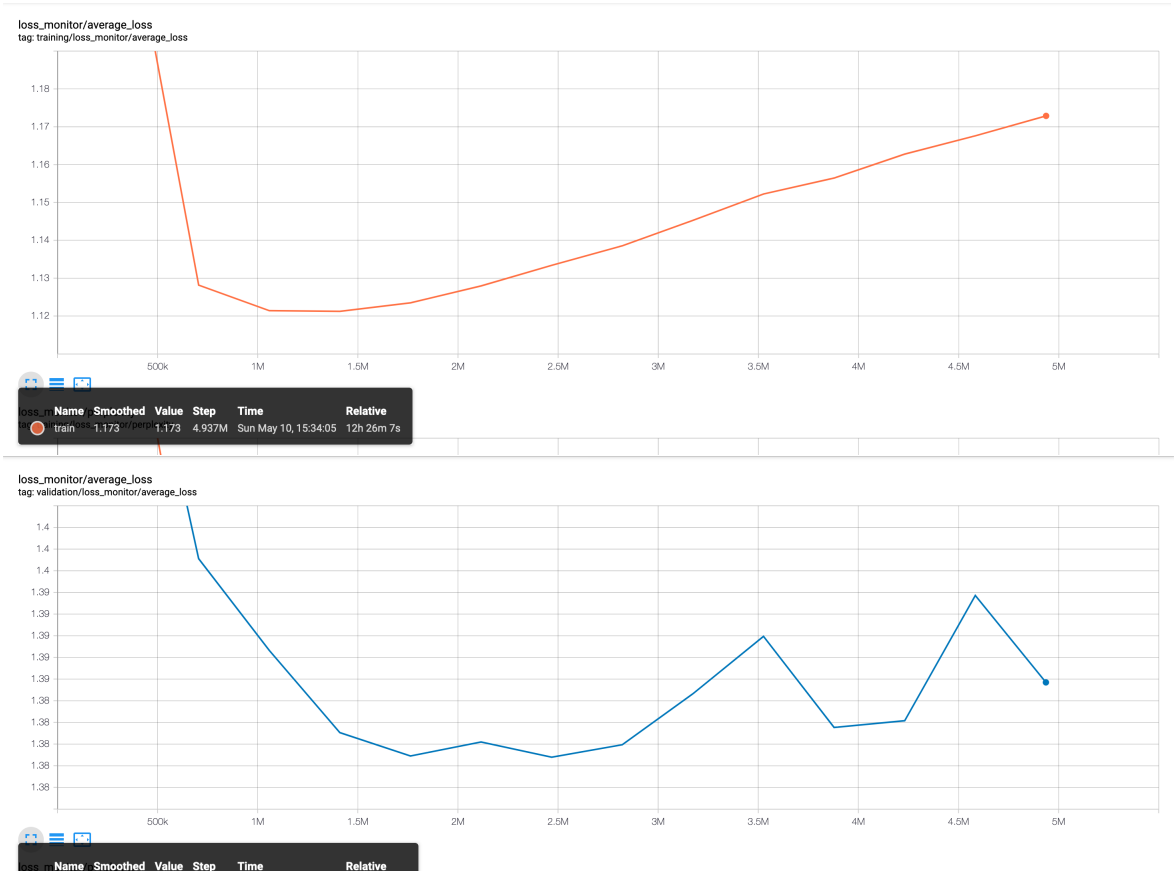

Figure 7: Training and Validation Losses on the Baseline Char-RNN Model. This is the cross-entropy loss over 128 predicted character sequence.

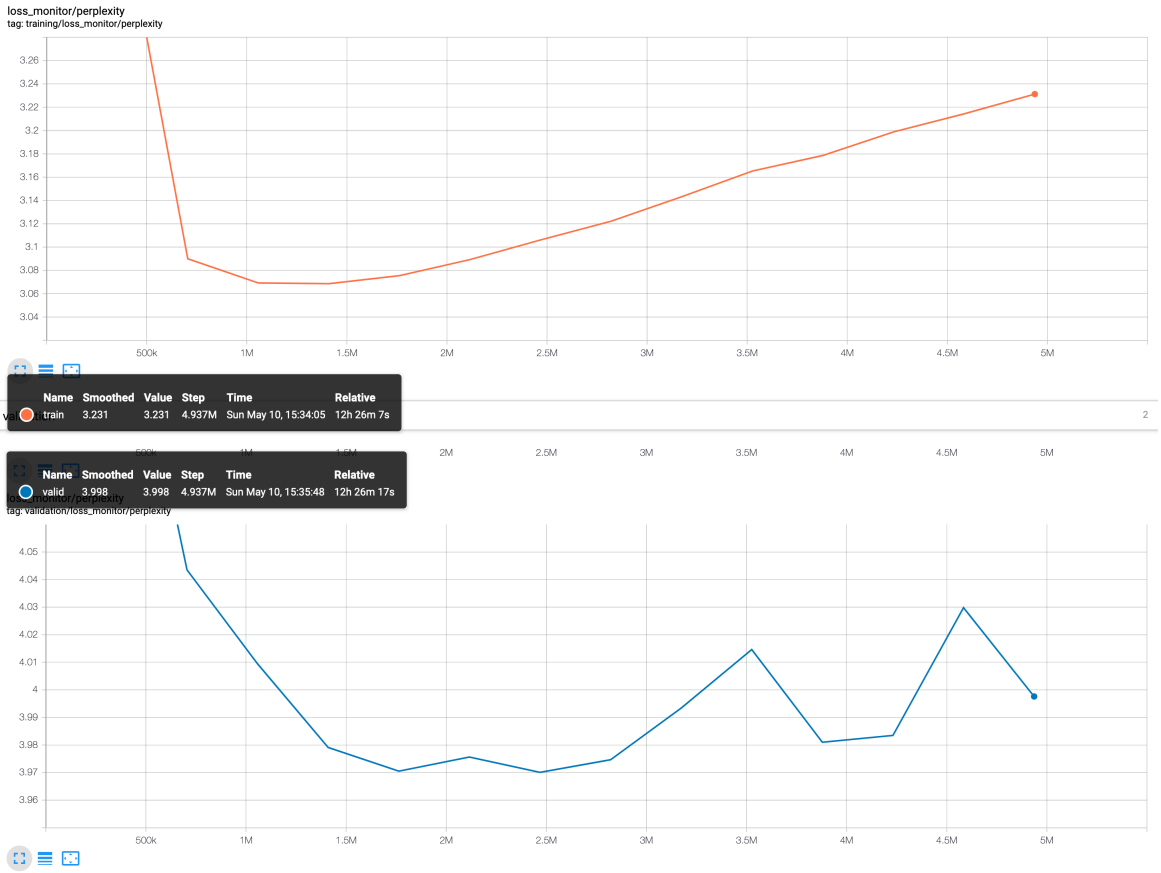

Figure 8: Training and Validation Perplexity on the Baseline Char-RNN Model. This is the cross-entropy loss over 128 predicted character sequence.