Automatic Code Generation using Pre-Trained Language Models

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

Recent advancements in natural language processing [1] [2] have led to near-human performance in multiple natural language tasks. In this paper, we seek to understand whether similar techniques can be applied to a highly structured environment with strict syntax rules. Specifically, we propose an end-to-end machine learning model for code generation in the Python language built on-top of pre-trained language models. We demonstrate that a fine-tuned model can perform well in code generation tasks, achieving a BLEU score of 0.22, an improvement of 46% over a reasonable sequence-to-sequence baseline. All results and related code used for training and data processing are available on GitHub. [1]

1. Introduction

Automating even small parts of software development is an active research area [3], with multiple approaches proposed methods (See Section 1). Succeeding in the automation of even small tasks can save time for countless software engineers, which translates to saved resources across multiple industries. Furthermore, as software continues to eat the world [2] and demand for experienced software developers continues to outpace supply, automatic code generation will become increasingly important.

In this paper, we propose a machine learning model to automate the task of writing code by assisting developers in writing individual units of functionality (or "functions"). 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 the initial lines of code (a function signature) along with a doc-string (function documentation) and generate the corresponding function body. In order to do this, we use a pre-trained language model and fine-tune it on a canonical corpus of Python code scraped from GitHub [4].

2. Background

A primary challenge in code generation is that it is still an active area of research, with many possible solutions and ongoing investigation [5]. State of the art solutions have not yet come close to automating basic tasks software engineers perform on a daily basis.

2.1. Traditional Code Completion

The most traditional and well-known approach used by multiple IDEs across a range of languages simply consists of token completion based on structured information obtained from static analysis of code. For example, when a developer types a sequence of characters, the system will attempt to find near-matching strings corresponding to function definitions and propose completing these function calls. Similarly, for object methods, on the typing of the accessor token (such as "-¿" or "."), the IDE will propose auto-completing different methods belonging to the object.

The biggest drawback of these approaches is that they lack true understanding of the programmers intent, and also lack context relating to the surrounding code other than that from heuristics by the tool's developers.

[1] See shared repository located at https://github.com/kandluis/cs230.
[2] See article by Marc Andreessen.

2.2. Using Machine Learning for Code Search

Another approach taken in multiple papers in the literature [4] involves framing the problem as a code search problem. Rather than trying to generate code or complete the code that the developer is making, we can re-frame the problem as one of searching for relevant pre-existing snippets. This is the primary approach we take in three of our baseline models.

2.3. Using Machine Learning for Code Generation

Other more novel approaches from literature [5] are typically applied to restricted language domains, and have massive 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 [5]. Work in this area has focused on creating more structured models that take advantage of specific architectures [6]. In [7], the authors work to first decompose the input sequence of text tokens for the context into a tree-like structure. Other approaches involve restricting the output of the model to a context-free grammar (CFG) or domain-specific language (DSL) [8]. A code generation model's output must adhere to a very specific form in order to be syntactically correct.

In this paper, we instead focus on taking a different approach. As has been demonstrated by ever-increasing sizes of language models, we focus on improving the performance on the code prediction task by making use of pre-trained language models that are then fine-tuned on code.

2.4. Dataset and Feature

In this project, we are leveraging the CodeSearchNet dataset [4]. The dataset consists of 2 million (comment, code) pairs from open source libraries, ranging in languages from Python to Javascript, PHP, Java, Go and Ruby. 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 3.

We restrict our dataset to samples in the Python programming language rather than the others available. Focusing on Python, there are over 1M methods and approximately 500k (comment, code) pairs that make up our dataset. We make this decision both for practical and modeling reasons. From a practical perspective, restricting to a reasonably-sized dataset focused on a single-language domains permits for more thorough ablation studies. From a modeling perspective, we belief that transfer learning from natural language to a programming language such as Python is an easier task to accomplish.

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. CodeSearchNet Models

Figure 4 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 4), 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:

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

3.2. From Scratch RNN Models

The above baseline is useful only in the sense that it would allow our system to find pre-existing code snippets which might be relevant to the developer. Since our goal is rather to make novel code, we propose a different baseline based on a more traditional sequence-to-sequence model.

In this case, we use a traditional RNN architecture which takes as input individual characters. The reason we take this approach is to circumvent the need to learn word-level embeddings. Furthermore, we hypothesize that making use of entire words, from NLP models, will actually harm the performance of the model for code generation. The primary reason for this being that most of the syntax involved in writing code does not generally map directly to the English language. Concretely, we encode each character present in the training data as a 1-of-k encoding (one-hot encoding) and feed them into an RNN one at a time. Our output will be a k-dimensional output vector corresponding to a probability distribution over the entire set of characters.

For the model architecture, we sweep over multiple types of RNN cells, including LSTM, RNN, and GRU. We find the best performing model to consists of an LSTM-based

model using a hidden state size of 128 with two hidden layers in the internal RNN cell. Our training takes place using sequences of 50 characters, sampled at random from our input code. Given a sequence from i to $i + 50$, the model is trained to predict the sequence from $i + 1$ to $i + 51$. This means we have a many-to-many sequence model (See Figure 6.2.1). We use batch size of 50 and train for a total of 50 epochs.

To avoid issues with gradient explosion and stabilize training, we make liberal use of gradient-clipping. In particular, we clip all gradients to an absolute size of 5.

We sweep over learning rates and find that a started learning rate of 0.002 with an exponentially decaying schedule appears to perform best as measured by a held-out validation set. We use a decay rate of 0.97 per epoch. We also experiment with the use of dropout, but find little impact on final performance.

3.3. Fine-Tuned Pre-Trained Large Language Models

Our final approach relies on the use of pre-trained language models. We fine tune our code generator using the small GPT-2 model with 117 million parameters. Using such a large backbone and continuing to fine tune allows us to generate synthetic code samples with even higher quality, treating programming languages as another specific domain alongside encyclopedia articles, news or books.

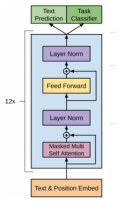


Figure 1. Decoder-Only Architecture used by GPT-2.

The general architecture of the GPT-2 model consists of a sequence-to-sequence predictive task based on the transformer architecture [9] [1]. However, it consists solely of the 12-layer decoder-only, as visualized in Figure 1. Each layer has 12 independent attention heads, leading to 144 distinct attention patterns. By making use of an attention-based framework, the model is more adept at dealing with long-range dependencies. This is because the attention mechanism allows the model to focus on the encoding of any of the input sequence tokens.

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. For details on these simple baselines, please see Appendix Section 6.1.

4.1. Code Generation with Char-RNN

As both of the above baselines focus on understanding and extracting useful embeddings for our overall task, our primary baseline consists of a straight-forward sequence-to-sequence model. Given that code typically does not consist of English words and can instead have quite a varied syntax, our baseline model is a model which uses character level embedding, so it is character aware [10].

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 [10], this corresponds to around 50MB of raw text, or 78,357,395 characters with 1,618 distinct symbols. Figure 9 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 10 shows the perplexity, or the amount of meaningful information encoded.

We include a sample generated from the best performing model for reference (See Section 2 in Appendix). A hyper-parameter tuning of learning rate and batch side for a total of 20 epochs has final measured performance as shown in Table 4.1.

Batch Size	**Starter Learning Rate**	**Regularization Weight**	**BLEU Score on Train**	**BLEU Score on Eval**
64	0.02	0.1	0.022	0.012
64	0.02	0.01	0.023	0.019
64	0.002	0.1	0.034	0.028
64	0.002	0.01	0.037	0.012
64	0.0002	0.1	0.09	0.073
64	0.0002	0.01	0.094	0.014
128	0.02	0.1	0.024	0.021
128	0.02	0.01	0.021	0.013
128	0.002	0.1	0.033	0.029
128	0.002	0.01	0.038	0.011
128	0.0002	0.1	**0.117**	**0.093**
128	0.0002	0.01	0.113	0.034

4.2. Code Generation with GPT-2

We have been working with the publicly available small GPT-2 model with 117 million parameters. We trained using the small GPT-2 model for 100,000 mini-batch iterations with a batch size of 2. We have included some sample code that our model generated directly in the report. Qualitatively, our model generates code which is far more reasonable than our baseline. The generated code is novel,

as verified by doing n-gram overlap analysis between the generated code and the training dataset. We also note that the model learns appropriate understanding of Python syntax, with uses of if-statements, function and method calls, as well as regularly commented code. For full output, see Appendix Section 6.2.

We observed that the idea of using Byte Pair encoding as used in GPT-2 is a much better strategy to generate code than just using characters, while of course the size of the models itself has a very observable effect in generating Python-like code.

Overall, the GPT-2 model quickly achieves performance that's much better than the baseline. Continued training of the model shows that our BLEU score performance will continue to increase, as seen in Figure 2

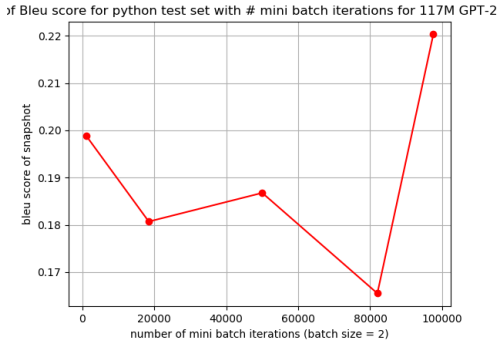


Figure 2. BLEU Score During Training of GPT-2 Based Model for Python Code Generation

5. Conclusions

In this paper, we explore the problem of automatically completing a function from the given function signature and human-readable documentation. We find the best performing model to be a fine-tuned version GPT-2, a transformer-based NLP model which is trained to generate natural text on an extremely large dataset. Despite the fact that our model focuses specifically on code rather than natural language, we hypothesize that it is able to treat programming language as another specific domain alongside the encyclopedia articles, news or books that its backbone has been trained on. We are able to achieve a BLEU score of 0.22, improving our baseline by ¿40%.

6. Contributions

All team member contributed equally to this project. Baselines from the CodeSearchNet models for code search were trained and tuned by Luis Perez and Sudharshan Viswanathan. Data analysis and understanding of the features (including histograms, distribution of tokens, and other data insights) was primarily performed by Lizi Ottens.

Training of the baseline char-rnn model, as well as analysis of results and discussion was contributed primarily by Luis Perez. Fine-tuning and training with the small and medium GPT-2 models was primarily explored and analyzed by Lizi Ottens and Sudharshan Viswanathan.

All written submissions were co-written by all three authors.

References

[1] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *None*, 2019. 1, 3

[2] Kenton Lee Kristina Toutanova Jacob Devlin, Ming-Wei Chang. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. 1

[3] 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. 1

[4] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2019. 1, 2, 6

[5] 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. 1, 2

[6] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. Treegen: A tree-based transformer architecture for code generation, 2019. 2

[7] Xinyue Liu, Xiangnan Kong, Lei Liu, and Kuorong Chiang. Treegan: Syntax-aware sequence generation with generative adversarial networks. *CoRR*, abs/1808.07582, 2018. 2

[8] 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. 2

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. 3

[10] Yoon Kim, Yacine Jernite, David A. Sontag, and Alexander M. Rush. Character-aware neural language models. *CoRR*, abs/1508.06615, 2015. 3

[11] 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. 6

Appendix and Figures

6.1. CodeSearchNet Results

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).

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

6.1.1 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 [4]. This baseline actually performs the best, achieving the lowest overall training and validation losses (see Figure 5) as well as the highest MRR on the validation set (See Figure 6).

6.1.2 Bi-directional RNN Model

In this model, we employ the GRU cell [11] to summarize the input sequence. This baseline performs significantly worse, suffering from what appears to be obvious over-fitting. In Figure 7, 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 is still clear that the model performs worse than the bag of words baseline as per Figure 8.

6.2. Example Code

Listing 1. Sample generated using Char-RNN model.

```
Downloads Dailymotion videos by URL. (x or pillares (if macks), style as a bool to you extner
        to oio instance of that Seir to be two categorical String to mandation :attr:`Columnser
                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']}
```

Listing 2. Sample generated using small (117M) GPT-2 model after 40000 iterations.

```
input_snippet = def semilocal_linear_trend_transition_matrix(autoregressive_coef):
  """Build the transition matrix for a semi-local linear trend model.""",
output_code =
  # Create the matrix
  # (autoregressive_linear_trend_matrix) where :param semilocal_linear_trend_matrix:
  #                 (n_samples) is length of the semi-sparse linear regression matrix;
  #                 i, j = semilocal_linear_trend

    # Create the columns
    columns = np.asarray(columns)
    if auto_correlated_trend_matrix:
        column = np.asarray(columns)

    # Create the indices for the columns
    i = column.min()
    j = column.max()
    # Ensure the columns have uniformity and the axis position
    assert hasattr(
        column, "sigma")
    assert hasattr(axes, "axis")

    # Create the columns
    i = columns.copy()
    j = columns.copy()

    # Ensure the indices for the indices for the columns
    ax = ColumnSetAxes(i, j)
    axes.set_axes(ax)

    # Create the columns
    ax.set_xlabel(self.default_label)
    ax.set_ylabel(self.default_ylabel)
    axes.set_title(self._title)
    axes.text(u"Subsim"]

    # Create the column index
    self.get_index(axis = columns, cols = cols)

    # Create the columns if not already provided by this function
    if not columns:
        if auto_correlated_trend_matrix:
            ax.columns = columns
        else:
            if i < 0 or j < 0 or i+1 <= i
```

6.2.1 Figures

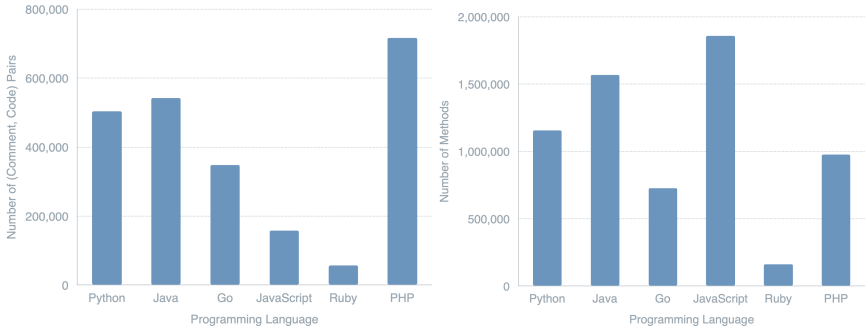


Figure 3. Histogram of the the number of (comment, code) pairs available in our dataset, as well as the number of unique function methods for each language.

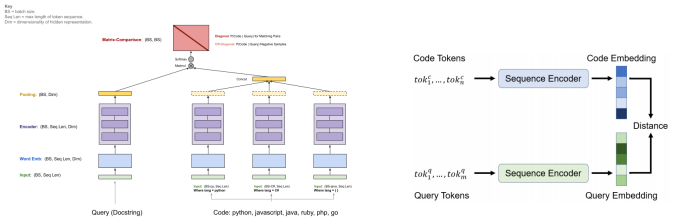


Figure 4. General CodeSearchNet architecture for all of our baselines. Each language is processed through different encoder mechanisms. The query encoder is shared (an NLP encoder), and the purpose of the CodeSearchNet tasks is to retrieve the most relevant code snippets subject to the natural language query.

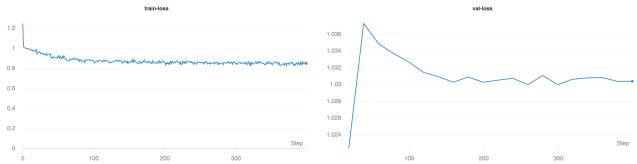


Figure 5. Training and Validation losses for the Neural Bag of Words model in CodeSearchNet.

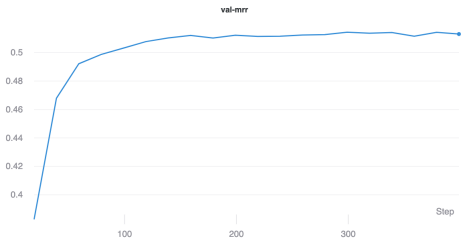


Figure 6. MRR on validation set for the baseline neural bag of words model in the CodeSearchNet Challenge.

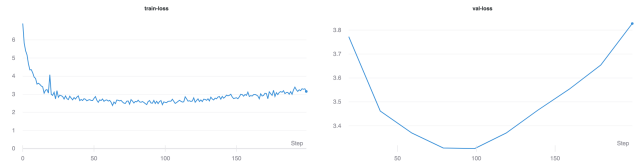


Figure 7. Training and Validation losses for the RNN model in CodeSearchNet.

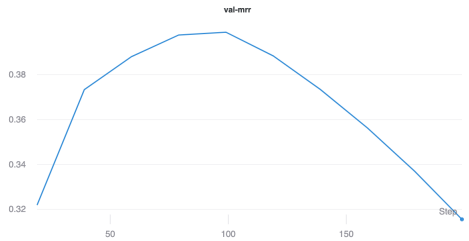


Figure 8. MRR on validation set for the baseline RNN in the CodeSearchNet Challenge.

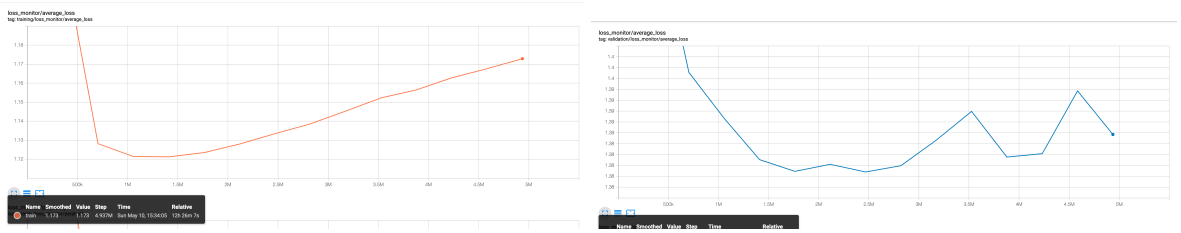


Figure 9. Training and Validation Losses on the Baseline Char-RNN Model. This is the cross-entropy loss over 128 predicted character sequence.

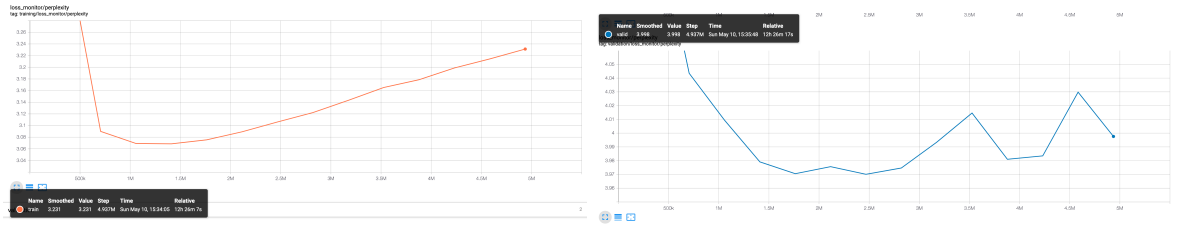
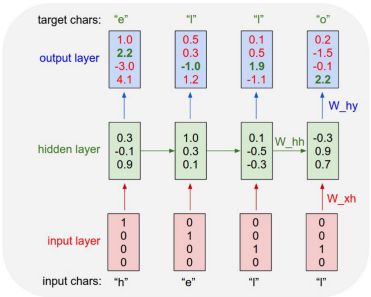


Figure 10. Training and Validation Perplexity on the Baseline Char-RNN Model. This is the cross-entropy loss over 128 predicted character sequence.



An example RNN with 4-dimensional input and output layers, and a hidden layer of 3 units (neurons). This diagram shows the activations in the forward pass when the RNN is fed the characters "hell" as input. The output layer contains confidences the RNN assigns for the next character (vocabulary is "h,e,l,o"); We want the green numbers to be high and red numbers to be low.