Recipe Scoring with a Recurrent Neural Network Sequence-to-Sequence Model

Kien Wei Siah
kienwei@mit.edu

Paul D. Myers
pdmyers@mit.edu

Abstract

A system for automatically identifying suitable substitutes for ingredients in a given recipe is proposed and implemented. The task is cast as a sequence-to-sequence problem, and is solved using a neural machine translation system based upon an encoder-decoder recurrent neural network architecture. Results obtained from training, validating, and testing on a recipe corpus crawled from the Internet reveal that the model produces results superior to those obtained by the baseline bigram language model. [1]

1 Introduction

Adapting a sequence of instructions is a common task in many application domains. For instance, in the field of chemistry, it is often necessary to modify a series of chemical reactions in order to obtain a variety of desired end products. If the application space is large, the manual modification process may become intractable; however, in many cases, a large subset of the possible modifications are unsuitable. The purpose of this project is to devise a scoring system that will automatically find the space of acceptable modifications to a sequence of instructions, thereby eliminating the need to search the entire space exhaustively. For illustrative purposes, the chosen application domain will be recipe modification. Often, the user of a particular recipe may find it necessary to modify the ingredients present in the recipe for a variety of reasons, such as taste requirements, ingredient availability, and calorie content, among others. In order to automate this procedure, a system will be designed to score each possible modification of a given recipe based upon the criterion that the modified recipe retain semantic similarity to the original recipe. The system consists of a sequence-to-sequence translation model implemented using a recurrent neural network (RNN) based upon a gated recurrent unit (GRU) architecture. The model is trained by applying individual sentences to the RNN and requiring that the network replicate these training sentences. A test recipe is then given to the network, along with a number of modified versions; the system then gives a score to each modification of the original recipe. Experiments reveal that in many cases, the system correctly identifies suitable substitutions in the original recipe.

[1] Implementation code, data sets, and results are available at https://github.mit.edu/pdmyers/DNN-recipe.git.

2 Recurrent Neural Network Sequence-to-Sequence Model

The task will be framed as a machine translation problem in which the model, given an input sequence x, must produce an output y that is identical to the input sequence x; the system therefore functions as an autoencoder. Recently, a neural machine translation model has been proposed in which two RNNs are placed in series with one another (Cho, 2014). The first network, termed the encoder, takes as input at each time step a word in a given sequence and produces a vector representation of the sequence once the end of the sequence has been reached. The second network, called the decoder, takes as input the vector representation of the sequence produced by the encoder and produces a word at each time step, the aggregation of which form the output sequence; Fig. 1 displays a schematic of the encoder-decoder architecture.

Neural networks offer, among other advantages, great flexibility in design. One important design choice to be made for a RNN is the selection of a neuron architecture. The simplest RNN consists of units which simply compute weighted sums of the inputs. While such a model offers simplicity of design, training the model proves

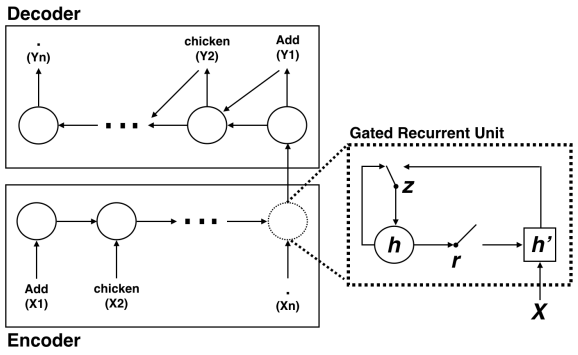

Figure 1: Schematic of the encoder-decoder architecture. The inset image shows the gated recurrent unit design.

to be difficult, as back-propagation must be performed through time, often resulting in gradients that tend towards zero; this result is often referred to as the "vanishing graidents problem." To solve this issue, a number of neuron architectures have been proposed, including long short-term memory (LSTM) (Hochreiter, 1997) and the gated recurrent unit (GRU) (Cho, 2014). Both LSTM and the GRU attempt to solve the vanishing gradients problem by selectively allowing memory to be passed to the next time step or forgotten. In the present project, both the LSTM and GRU architectures were implemented, and no appreciable difference was found between the two; since the GRU is known to require fewer parameters than does LSTM, the GRU model was chosen for the final implementation. The design of the GRU, taken from (Cho, 2014), is shown in the inset in Fig. 1. Mathematically, the GRU may be represented as a set of functions called "gates," which are described by the following set of equations.

$$z = \sigma(U^z x_t + W^z s_{t-1}) \tag{1}$$

$$r = \sigma(U^r x_t + W^r s_{t-1}) \tag{2}$$

$$h = tanh(U^h x_t + W^h (s_{t-1} \cdot r)) \tag{3}$$

$$s_t = (1 - z) \cdot h + z \cdot s_{t-1} \tag{4}$$

$$o_t = \sigma(V s_t) \tag{5}$$

Here, z is known as the update gate, r is known as the reset gate, and o is known as the output gate. t denotes the current time step, σ is the logistic sigmoid function, s is the hidden state, and U, V, and W are weights of the model that are shared between time steps and are learned during training; note that $\cdot$ denotes element-wise multiplication. The weights may be shared between the encoder and decoder in a so-called "tied" configuration, or may be set independently. In order to reduce the number of parameters to be learned by the model to reduce the computation time, the tied configuration only was implemented in this project.

The neural machine translation model described above produces as output the probability of an output sequence y given an input sequence x and parameters θ as follows:

$$P(y_1, y_2, ..., y_n, < END > | x_1, x_2, ..., x_m, \theta) = \prod_{i=1}^{n+1} P(y_i | y_1, ..., y_{i-1}, x, \theta) \tag{6}$$

Here, $< END >$ is a special symbol that denotes the end of a sequence, n is the length of the output sequence, and m is the length of the input sequence; note that in the present model, n and m should be equivalent if the model is functioning properly. The above probability may be interpreted as a score for the given sequence, as it denotes how likely the sequence is to occur given the training set.

With the basic RNN model in place, a number of customizations particular to this application were implemented in *Python* using the *TensorFlow* software package; many of these customizations were inspired by work done by the *TensorFlow* design team (Abadi, 2015). First, during training, the model was given the correct decoded word from the previous time step regardless of whether the model actually produced this word. For example, if the sentence to be decoded was "add chicken to the bowl." and the model decoded "chicken" into "salmon," the model would be given "chicken" to ensure that the remainder of the sentence was decoded correctly. During testing, the model was not supplied with the correct output, and was therefore required to use the its own generated results. The predictions made by the decoder during testing are done in a greedy fashion, where the decoded word is generated by taking the argmax of the logistic output. Another feature of the model design was that inputs to the encoder were padded and applied to the encoder

in reverse, following the approach suggested by (Sutskever, 2014). For instance, if the input sentence is "add chicken to the bowl." and the sentence is set to be of length seven, the input would be [PAD PAD "." "bowl" "the" "to" "chicken" "add"] and the correctly decoded sentence would be [GO "add" "chicken" "to" "the" "bowl" "." EOS PAD PAD]; here, "GO" and "EOS" mark the beginning and end of the sequence, respectively. To improve the efficiency of the model, sentences were permitted to be of certain fixed lengths. To accomplish this task, a number of buckets representing fixed sentences lengths were chosen, and sentences were placed into these buckets based upon their lengths; PAD symbols were appended to sentences that did not fit into any one bucket. The buckets chosen were for sentences of lengths 5, 10, 20, and 40, where length is measured by the number of tokens in the sentence. For further efficiency improvements, the sampled softmax and output projection techniques were used to manage the large vocabulary size (Jean, 2014). If the vocabulary size is large, such as in the present case where the vocabulary size was set to $3,000$, the outputs of the model will require a large amount of memory to store, thereby rendering the computation intractable for commercial computers. Sampled sotmaxes and output projections may be used to efficiently reduce the dimensionality of these outputs for storage and manipulation, and allow the original outputs to be recovered when necessary.

In addition to the approach described above, a number of other approaches were explored in the hopes of achiveing improved performance. One such approach involvedusing the encoder-decoder architecture in a manner different from that described above. Since the encoder produces a compressed vector representaiton of each sentence in a given corpus, one approach might be to use the similarity between the vector reprepresentation of the original sentence and the modified sentence. Modified sentences that are semantically similar to the original sentence should produce vector representations that are similar to the the vector representation of the original sentence. The cosine similarity was used as the comparison metric. While the approach seemed reasonable, it was found to produce poor results, likely because the assumption that good substitions should produce encoded vectors similar to the original vector was faulty. Since the RNN architecture is quite complex, it is not clear how semantic relationships are encoded in the state vector; perhaps, some components of the vector possess more information than others do, thereby rendering the cosine similarity a poor metric for scoring. An alternative procedure was implemented in the *Python* package called *Keras* (Chollet, 2015). This implementation was largely a simplified precursor to the more sophistaced model described above, in that it did not include a method to provide the model with correctly decoded words during training, contained no bucketing mechanism, used the mean squared error for evaluateion rather than sampled softmax, did not learn word embeddings, but instead used pre-trained word vectors directly, and contained no feedback in the decoder, as shown in Fig. 1. This model was found to produce inadequate results.

3 Recipe Data Collection and Processing

In order to train, validate, and test the model it was first necessary to collect recipe data. To do so, a web-crawling program was written using the *Scrapy* (Scrapy Team, 2015) software package and designed to extract recipe instructions and ingredients from the websites *AllRecipes.com* (AllRecipes.com, 2015) and *Food.com* (Food.com, 2015). In order to ensure that no conflicting contexts would occur for a given ingredient due to differing uses in disparate recipe families, the domain of recipe types was limited to pasta; in total, approximately $4,000$ recipes were collected, giving $19,000$ training sentences, $6,500$ validation sentences, and $6,500$ test sentences. While many recipes offered reasonably good training data, a large number of recipes contained inconsistencies, such as spelling errors, non-standard abbreviations, and obscure ingredient names; thus, it was necessary to apply significant preprocessing techniques to the data. Examples of errors include misspelling certain ingredient names, such as "spaghetti," which was sometimes incorrectly spelled as "spagheti," and abbreviating measurement quantities, such as "Qt." and "Qrt." in place of "Quart." Although much of the preprocessing was automated, the random nature of the inconsistencies in the data required that a significant amount of the preprocessing effort needed to be done manually. Preprocessing the text was shown to improve the quality of the results noticeably.

4 Word Embeddings Data

As noted above, the employed model takes as input a sequence of words that compose a sentence; these words are presented to the model in the form of word vectors, so it is therefore necessary to consider the means by which the word vectors may be generated. One option is to allow the model to learn the word embeddings from the training data; this approach can be advantageous, as it will produce word vectors that capture semantic relationships specific to the domain of interest. The primary disadvantage of learning the word embeddings from the training data is that a large training corpus is required to obtain robust word vectors. Another option is to use pre-trained word vectors. Although these word vectors were generated from corpora outside of the domain of interest, it is likely that they will still capture meaningful semantic relationships for the present application; this option avoids the need to generate a large training corpus to learn the embeddings. A third option is to use a combination of the preceding two approaches, wherein pre-trained word vectors are used to initialize the vectors to be learned and then the model is allowed to refine these embeddings. This approach is advantageous because it reduces the effects of having a small training corpus on the quality of the word embeddings, and allows the model to learn embeddings for words not found in the pre-trained word vectors. The *Global Vectors for Word Representation* (GloVe) word vectors trained on 2014 *Wikipedia* and *Gigaword 5* articles with six billion tokens, a vocabulary size of $400,000$, and 50 dimensions were used for initialization (Pennington, 2014).

5 Testing Procedure

As discussed briefly in the introduction, the objective of this study is to design a system capable of identifying suitable substitutes for ingredients in a recipe. The high-level procedure is as follows:

For a given recipe:
1. Choose an ingredient to replace
2. Choose a replacement criterion
3. Rank the possible substitutes
4. Generate a modified recipe using the best substitute

Table 1: Description of the high-level recipe generation procedure.

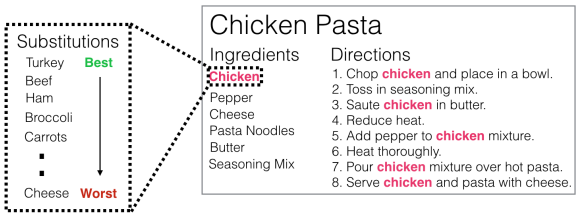

Figure 2: Graphical depiction of the objective of this study.

Fig. 2 presents a graphical display of the desired task.

As discussed above, the model is trained to reproduce the recipes in the training corpus. The training procedure is as follows: Tokenize the recipes into sentences; input each sentence to the model; and repeat until the model reconstructs the input sentences; Fig. 3 shows the training procedure on a test sentence. Each variation on the model was trained for at least $150,000$ epochs, which ensured that the models sufficiently convereged.

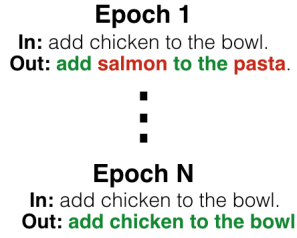

Figure 3: Example of the training procedure with a sample sentence.

After training, the simulation procedure follows that outlined in Table 1. In order to obtain statistically meaningful results, it was necessary to use a testing corpus of adequate size, which further required that the testing procedure be automated to the fullest extent possible. Thus, to accomplish Step 1 in Table 1, a random ingredient was chosen to be replaced in each recipe; in practice, the user would select the desired ingredient. For Step 2, semantic similarity was chosen as the replacement criterion. For example, if the ingredient to be replaced is chicken, then a reasonable substitution might be turkey. Less acceptable substitutions might include beef or pork, which satisfy the requirement that the replacement ingredient be a meat, but fail to satisfy the requirement that the replacement ingredient be poultry. Other selection criteria are possible, but would require a more

specialized training corpus than the one utilized in this study. For instance, the user might wish to perform a substitution based on the calorie content of the ingredient. The system here utilized could likely perform this task given appropriate training data.

With the selection criterion selected, Step 3 may be accomplished as described in Table 2; Fig. 4 provides a graphical depiction of this procedure.

For a given recipe and a given ingredient:
1. Replace all occurrences of the chosen ingredient with a template word $< WORD >$
2. For each ingredient in the test corpus, replace all occurrences of $< WORD >$ with the that ingredient
3. Calculate the average perplexity of the recipe with the substituted ingredient in place

Table 2: Description of the testing procedure.

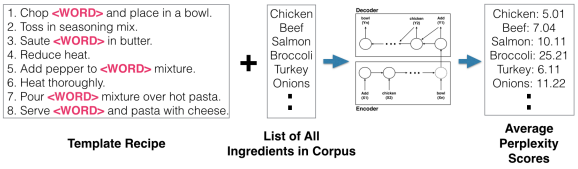

Figure 4: Graphical depiction of the testing procedure.

Following the procedure outlined in Table 2, an ingredient in a chosen recipe is chosen at random and all occurrences of this ingredient in the recipe are replaced with a template word, denoted by $< WORD >$. All ingredients in the test corpus across all recipes are then collected into a single list, and the template word is iteratively replaced by each word in this list. The model then calculates the average perplexity of the recipe with each substitute ingredient in place. After all ingredients are scored, the ingredients are ranked from lowest perplexity to highest perplexity, and the top five ingredients are examined. If at least one of the top five ingredients is found to be an acceptable substitute for the original ingredient, as determined by a human annotator, then the model is said to have succeeded for that recipe; the model is likewise said to fail if none of the top five ingredients is found to be suitable. The accuracy is then computed by taking the ratio of the number of successfully modified recipes to the total number of recipes tested, as follows:

$$Accuracy = \frac{\#\ Successfully\ Modified\ Recipes}{\#\ Recipes} \tag{7}$$

6 Results and Analysis

The evaluation procedure described above was applied to the proposed model. For comparison, a bigram language model was implemented as a baseline. The bigram language model computes the perplexity of a given recipe by calculating the product of the probabilities obtained from the training corpus of all bigrams in the given recipe; add-α smoothing was employed and a validation set was used to choose the smoothing parameter. The perplexity calculated by both the bigram model is defined as follows:

$$Perplexity = 2^{-\frac{1}{N}\sum_{i=1}^{N} logP(w_i|w_{i-2},w_{i-1})} \tag{8}$$

Here, N is the number of bigrams in the corpus, and w is a word. The perplexity was used to rank the ingredients substituted into the given recipe; the ingredients with the lowest perplexities were considered to be the best substitutions, since those ingredients produce bigrams that are more likely to appear in similar contexts in the original recipe. Eq. (6) was used in a similar fashion for the RNN model, but with some variations. Since the RNN model is designed to decode an input sequence into an output sequence during testing, evaluation of the model is based on the perplexity, which is derived from Eq. (6), of the original sentence in the recipe as compared with the perplexity of the modified sentence; the model therefore calculates the probability of decoding the modified sentence into the original sentence. Modifications with higher probabilities represent semantically similar substitutions as a result. For example, if the original sentence is "Add chicken to the bowl.", the modification "Add beef to the bowl." is more likely to decode into the original sentence than would "Add water to the bowl.", since beef and chicken are more similar than are water and chicken.

As is the case with most statistical learning algorithms, the RNN model used here possesses a number of hyperparameters that may be adjusted to improve the performance of the model. Due to the long training and testing times required to obtain meaningful results, only a subset of these

hyperparameters was investigated thoroughly; the subset includes the number of layers in the network, the number of which was chosen to be either two or three, and whether or not the generated word embeddings were initialized with pretrained word vectors. Other hyperparameters such as the number of hidden units, batch size, and initial learning rate were held constant across all training and testing sessions. The number of hidden units was set to 50 to match the dimensions of the word vectors, the batch size was set to 64, and the initial learning rate, which was modified adaptively during training, was set to 0.5.

A total of 200 recipes from the test set were evaluated using four variations on the proposed model and the baseline. Running the testing procedure outlined above was found to require a great deal of time; coupled with the need to manually examine the results for each recipe in the test set, it was not possible to score more than 200 recipes given the short time-frame, shortage of man-power, and limited computing resources.

Table 3 presents the results of the testing procedure. A number of trends may be surmised from the table. First, it appears that using word vector initializations dramatically improves the results, since for both the two-layer and three-layer networks, the models with word vector initializations performed markedly better than the models without word vector initializations. This result is consistent with expectation, since the relatively small training set used in this study was perhaps insufficient for properly learning the word embeddings for all of the ingredients. Second, it appears that, at least for the models with word vector initializations, increasing the number of layers improves the results. An interesting results obtained during the evaluation procedure was that the model with three layers and word vector initializations was quite adept at recognizing suitable substitutions for the various types of pasta noodles in the recipes. For example, if a recipe used fusilli, a spiral-shaped noodle, the model correctly identified linguine, a long, thick noodle, as a valid substitution. All other models, including the baseline, were unable to perform this task reliably in the majority of the cases. Moreover, the three-layer model with word vector initializations was able to identify suitable substitutes for onions much of the time, such as shallots and chives, while the other models were unable to do so. Both the three-layer network with initializations and the baseline model were generally capable of finding suitable substitutes for common meats, such as chicken, and some seafoods, such as shrimp, while the other models were less able to do so.

To gain further insight into the capabilities of the model, the three-layer network with initializations, which was the optimal model, and the baseline model were further examined on a number of hand-crafted toy recipes. One such recipe is shown in Fig. 5. The ingredient to be replaced in this task was chicken, and the models were given four other ingredients from which to choose a substitute: Turkey, beef, broccoli, and onions. Interestingly, the optimal network model selected beef rather than turkey, which would presumably be the optimal substitution; the bigram model chose broccoli. To gain insight into the choice made by the model, the training corpus was examined in detail. All sentences containing the ingredients chicken, turkey, and beef were compared with the sentences found in the toy test recipe. It was found that in the training corpus, the of sentences in which chicken and beef occurred generally contained similar context words, such as saute and chop, both of which were found in the toy recipe; the ingredient turkey did not appear in similar contexts anywhere in the training corpus, so it is therefore reasonable that the model chose beef instead of turkey as the best substitution. This simple exercise revealed the importance of using a training set of good quality and adequate size. Given additional time, a number of other models would have been trained with much larger and much more carefully pre-processed training corpora. Upon further retrospection, restricting the recipe domain to pasta was perhaps unnecessary; it is likely that using recipes from a wider variety of domains would have further improved the results by introducing additional context information for each ingredient.

7 Conclusion and Future Work

The findings of this study underscore the challenges and opportunities inherent in designing recommender systems, of which the proposed recipe scoring system is an example. While the proposed system achieved adequate results on the specified task, there is a great deal of margin for improvement. Besides increasing the size and quality of the training corpus, further improvements

# Layers	Initialization?	Accuracy
2	No	20.30%
2	Yes	24.10%
3	No	19.35%
3	Yes	**33.86%**
Baseline	–	19.47%

Table 3: Results of the testing procedure. The *Initialization?* column refers to whether the word embeddings were initialized with GloVe word vectors.

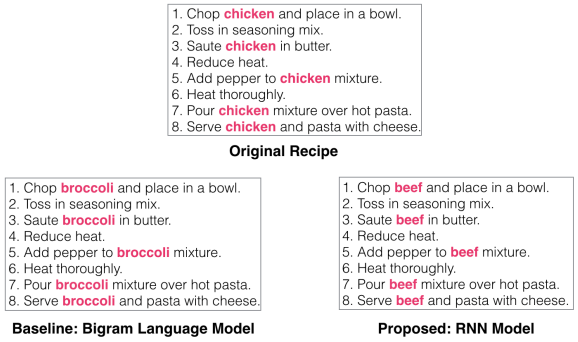

Figure 5: An example of results generated by the proposed RNN model and the baseline bigram model.

might include more detailed tuning of the network hyperparameters, particularly the number of layers, and deeper exploration into the word embedding training procedure. Although 50-dimensional word vectors were used in this study for computational reasons, vectors of higher dimensionality could also have been used and may have further improved the results. Moreover, examining the use of word vectors trained on sources other than *Wikipedia* and *Gigaword*, such as *Google News*, might have yielded interesting results had time allowed for this exploration; perhaps even more interesting would have been the results obtained from learning the word embeddings from a large recipe corpus without initialization using pre-trained vectors. Implicit in these considerations is the importance of attention to the system setup on the performance of the model. While the encoder-decoder model discussed in this study seems well suited to the proposed task, the details of the implementation are what allow the model to work well. The rapidly growing interest in the application of neural network models to a variety of problem domains will likely result in clever solutions to these implementation considerations in the near future, allowing for the design of high-performance models for recipe scoring and beyond.

Acknowledgments

The authors would like to thank Professor Regina Barzilay, Professor Tommi Jaakkola, and Mr. Karthik Narasimhan for their valuable advice throughout the semester.

References

K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *EMNLP 2014*, 2014.

S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, Vol. 9, no. 8, pp.1735-1780, 1997.

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D.M., M Schuster, R. Monga, S. Moore, D. Murray, C. Olah, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Vigas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," *Software available from tensorflow.org*, 2015.

I. Sutskever, O. Vinyals and Q.V. Le, "Sequence to sequence learning with neural networks," *NIPS 2014*, 2014.

S.Jean, K. Cho, R. Memisevic, and Y. Bengio, "On using very large target vocabulary for neural machine translation," 2014.

F. Chollet, "Keras," *GitHub Repository, https://github.com/fchollet/keras*, 2015.

Scrapy Team, "Scrapy," *GitHub Repository, https://github.com/scrapy/scrapy/tree/1.0*, 2015.

Retrieved from http://allrecipes.com/search/results/pasta, 2015.

Retrieved from http://www.food.com/search/pasta, 2015.

J. Pennington, R. Socher, and C.D. Manning, "GloVe: Global Vectors for Word Representation," 2014.