Explorations in Deep Learning for Recipe Generation

Brendan King
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98105
`kingb12@cs.washington.edu`

Antoine Bosselut
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98105
`antoineb@cs.washington.edu`

Ari Holtzman
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98105
`ahai@cs.washington.ed`

Abstract

I spent winter and spring quarters working with Antoine Bosselut and Ari Holtzman on deep learning for natural language generation. We approached the Recipe Generation task in which a title and list of ingredients is translated into a procedural cooking recipe. Beginning with deep learning for NLP fundamentals and building up to a functional generation system, we approached this problem with the ultimate goal of improving our sequence-to-sequence model with deep reinforcement learning. We begin with a language model, followed by an encoder-decoder model for recipe generation. We make several adjustments along the way, such as the including attention mechanism over a list of ingredients and reinforcement training. Included here is a chronological walk-through of our experiments, models, and results.

1 Introduction and Motivation

Natural language generation problems, like many others in natural language processing, involve reasoning over a very large combinatorial space. As such, they provide challenging problems for NLP and deep learning models. We choose to focus on recipe generation, in which given contextual information such as a recipe's title and a list of ingredients with quantities, we must generate a procedure for preparing the dish. This problem is interesting for several reasons. First, the procedural language in a recipe's instructions have an inherent global structure in which dependencies in the language used often mirror physical dependencies in the successful preparation of a dish. For example, baking chicken in the oven involves preheating the oven, preparing the chicken, placing it in the oven for some duration and removing it to serve. Instructions for these steps can be distant from one another within a recipe but all must occur, and not all orders are valid. The dependence relationships among these instructions are grounded in the real-world steps required to cook chicken. Second, a successful model in the recipe generation task must develop some common sense understanding of the kitchen environment, as example recipes often leave quite a lot for the reader to infer or know in advance (for example, that chicken can never be served uncooked). Finally, the recipe generation task provides a fairly intuitive framework with which we can evaluate a proposed set of instructions, but which is challenging to perform with computation. The goal of a recipe's instructions is to guide a reader with an understanding of basic cooking from a set of raw ingredients to a prepared and ready-to-eat meal. In most cases, one can qualitatively evaluate a recipes efficacy using our own basic understanding of cooking. However, to evaluate this quantitatively or computationally is very challenging, let alone in such a way that a model can be trained efficiently to meet this criteria.

We approach recipe generation as a sequence to sequence problem, in which our model provides a mapping from an input sequence of a recipe's title and ingredients to an output sequence of instructions. Previous work in deep learning models for sequence to sequence problems, such as machine translation, has shown considerable success using recurrent neural networks (RNNs) (Sutskever et. al.). Related work has shown success in applying RNN-based model's to recipe generation (Kiddon et. al. 2016). We begin with the RNN-based encoder-decoder approach described in Sutskever et. al. Experiencing only limited success with this approach, we add various improvements to our model, settling on one that uses an attention mechanism over ingredients. Finally, we begin to explore using reinforcement learning as a means for improving the relationship between our model's training criteria and the criteria we use for common sense evaluation of recipes. This remains under-explored and provides a good frontier for future work.

2 Beginning with a Word-level Language Model

After an introduction to the modeling techniques used in deep learning for natural language processing, the first task we approached was building a word-level language model. We approached this task as an achievable introduction to Torch and building models for natural language generation. Using a small subset of the Google Billion Words corpus, we built a number of language models and evaluated their generative performance.

2.1 Problem and data formulation

2.1.1 Language modeling

In language modeling, our task is to compute the probability of a sequence occurring in natural language or in a particular domain. We can model the probability of a word $P(w_i)$ as a distribution over a vocabulary space of all words V. The joint probability of a sequence of m words $P(w_1, ..., w_m)$ is equivalent to the product of the probabilities of each word w_i conditioned on the words which precede it:

$$P(w_1, ..., w_m) = \prod_{i=1}^{i=m} P(w_i|w_1, ..., w_{i-1}) \tag{1}$$

As such, maximizing the likelihood of each word conditioned on those before it is equivalent to maximizing the likelihood of the sequence. We can then define our abstract model as one which takes a sequence of words $w_1...w_i$ as input and outputs a probability distribution $P(w_{i+1} \in V|w_1...w_i)$ for the next word in the sequence.

2.1.2 Data

Our training data set is composed of sentences from the Google Billion Words corpus, a large natural language dataset derived from news articles. Our goal is to build a language model over sentence-length examples, . For a given example, our training example is composed of an input and output sequence:

input: $beg, w_1, ...w_m$
output: $w_1, ...w_m, end$

where at each time-step $0 <= i <= m$, our goal is to take the sub-sequence from $0...i$ in the input and predict the ith token in the output. beg and end are special padding tokens used so that the ith

word in the output follows the ith word of the input in the original example sequence. In total, we train on over six hundred thousand sentences.

In order to make training training tractable, a couple of data transformations were considered. The first concerns the data set's very large vocabulary size of eight hundred thousand words. As we will discuss further in subsequent sections, a *softmax operation*, which we use to output our probability distribution over vocabulary words, requires linear time with respect to vocabulary size, which can make this operation the rate-limiting step for large vocabularies. To resolve this, we chose to limit the vocabulary to twenty five thousand words, replacing unknown words with an *unk* token. The second transformation we made concerns sequence length. Our recurrent architecture performs a number of back-propagation through time steps proportional to sequence length, and a significant number of our training examples were sentences of several hundred words. To improve training speed and make training tractable on available hardware, we chose to focus on sequences of twenty five tokens or less.

Given our intent to approach recipe generation with subsequent models, we later ran some of our language modeling experiments with a similarly constructed data set of recipe procedures. This recipe data set has a vocabulary size of six thousand, making stronger performance more tractable.

2.2 Model Formulation

We chose to use a recurrent neural network for our language model. The primary strength of recurrent models over simpler n-gram models is their capacity to consider the full history of preceding words instead of a fixed-length history. Deep architectures are also trainable end-to-end, given an architecture of only randomly initialized parameters and a data set of sequences. Below we describe the components of our architecture in more detail:

2.2.1 Word Embeddings

Dense word embeddings give our model the capacity to learn relationships among words, and to model attributes of each word (Goldberg 2015). We chose to represent each word w_i with a vector x_i in a learned embedding space $\mathbb{R}^D$, where D is a hyper-parameter. Many available tools can provide pre-trained embeddings, but for simplicity we choose to initialize these at random and learn them from data. Returning to our abstract task of producing a probability distribution $P(w_{t+1}|w_1...w_t)$ given a partial input sequence $w_1...w_t$ of length T, our first step is to transform this into a series of word embeddings, a tensor of size $T * D$, which we accomplish with a lookup-table.

2.2.2 LSTM recurrent module

The general recurrent neural network module reads a sequence of input $x_1...x_t$ and both updates and outputs a hidden state h_t at each time step according to the following equations:

$$h_t = \sigma(W^{(hh)}h_{t-1} + W^{(hx)}x_t) \tag{2}$$

where $W^{(hh)}$ and $W^{(hx)}$ are matrices of learned parameters, σ is a non-linear activation function such as the $sigmoid$ function, h_{t-1} is the previous hidden state, and x_t is the current input, in our case a word embedding. While models constructed this way are powerful, they often struggle to manage longer term dependencies. We choose to use another form of recurrent network designed to address this, the Long Short Term Memory (LSTM) module (Hochreiter and Schmidhuber 1997). The LSTM similarly operates over a sequence of inputs but maintains both a hidden state and a cell state in $\mathbb{R}^H$, where H is a hyper-parameter. The hidden state can be thought of as short-term memory, and the cell state as long term. These states are updated with the following set of equations:

$$f_t = \sigma(W_f \dot{[} h_{t-1}, x_t] + b_f) \tag{3}$$

$$i_t = \sigma(W_i \dot{[} h_{t-1}, x_t] + b_i) \tag{4}$$

$$\tilde{C}_t = \tanh(W_C \dot{[} h_{t-1}, x_t] + b_C) \tag{5}$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{6}$$

$$o_t = \sigma(W_o \dot{[} h_{t-1}, x_t] + b_o) \tag{7}$$

$$h_t = o_t * \tanh(C_t) \tag{8}$$

The cell state is updated at each time step using equations (3) - (6). a candidate cell state $\tilde{C_t}$ is proposed, and gating mechanisms decide what information can be forgotten from C_{t-1}, and what new information should be added from $\tilde{C_t}$. In equations (7-8), the output gate determines what should be extracted from the cell and hidden states and outputted at this time step.

LSTM modules output a sequence of hidden states $h_1...h_t$ which can be used in downstream computations such as computing a probability distribution over words. Notice that the forward pass requires T updates, and likewise the backward pass will require T parameter updates. This necessitates our cap on sequence length for input sentences. We feed our sequence of word embeddings into the LSTM module, and use the sequence of hidden states to compute a sequence of distributions over the vocabulary.

2.2.3 Computing a distribution over the vocabulary

Our LSTM module(s) output a sequence of hidden states $h_1...h_t$, which we transform into sequence of discreet probability distributions over the vocabulary space. We do this by first using a linear projection to put our sequence in the dimensions of the vocabulary space:

$$h'_t = W_p h_t + b_p \tag{9}$$

where W_p is a $|V| * h$ matrix of learned parameters and b_p is a vector of learned parameters of size $|V|$. This gives us a sequence $h'_1...h'_t$ of length $|V|$ vectors which we can transform into valid probability distributions using the *softmax* function:

$$p(x_k) = \frac{exp(x_k))}{\sum_{j=1}^{|V|} exp(x_j))} \tag{10}$$

Applying this function to each of our vectors $h'_1...h'_t$ will give us a sequence of valid probability distributions where $h_{i,j}$ gives the probability of word j being the next word conditioned on the input sequence $w_1...w_{i-1}$. The sum in the denominator of the *softmax* function necessitates the reduction in vocabulary size we previously discussed, as this step requires $O(|V|)$ computation time for each training example.

Combining these components, we can assemble a model capable of modeling the probability of words in our vocabulary conditioned on the words which precede them, as shown in Figure 1.

2.3 Training the model

In the previous section, we outlined an architecture for a neural network model with the capacity for modeling the probability distributions over sentences in a language modeling task. Here, we discuss how these parameters are learned with back-propagation, using *stochastic gradient descent (SGD).*

Given a gradient update with respect to our output module (the *softmax* computation), we can update all of the learn-able parameters in the LSTM module as well as our learned word embeddings using back-propagation. To compute this outermost gradient and apply appropriate updates within the model, we need both a loss function and a training algorithm.

2.3.1 Cross-entropy loss function

We chose to use the cross-entropy loss to compute our gradients. Cross entropy is used to compare to probability distributions, one which we have calculated with our model, and the true distribution

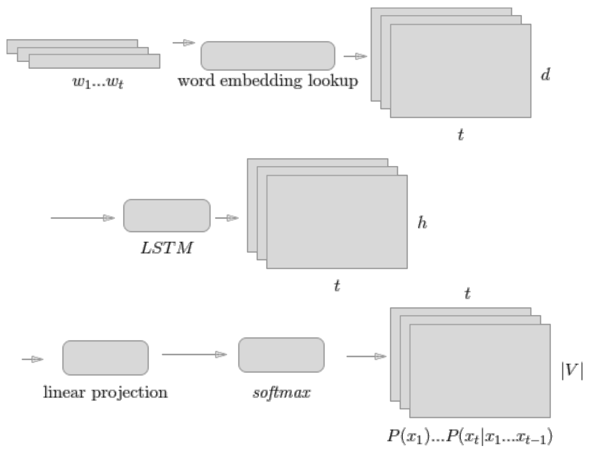

Figure 1: An overall look at our general model architecture, beginning with the learned word embeddings, followed by a LSTM recurrent module, and finally a linear projection to the vocabulary space and *softmax* application to produce valid probability distributions.

which is approximated by the correct word. the average cross-entropy C of a sequence of T words is calculated as follows:

$$C = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{|V|} y_{t,j} \times log(\hat{y}_{t,j}) \tag{11}$$

where y is the true distribution of possible words, and $\hat{y}$ is our calculated distribution. Cross-entropy loss is useful particularly for this task since our goal in language modeling is to best estimate y.

2.3.2 Stochastic gradient descent and Adam optimization

We use two training algorithms to update our parameters, both a form of *SGD*. The first is a *mini-batch stochastic gradient descent*, in which a batch of N training examples are used to calculate a single average loss across a batch. This loss is the back-propagated through the model according to the SGD update:

$$W^* := W^* - \alpha * \sum_{i=1}^{N} \nabla C(W^*, x_{i,1}...x_{i,t}) \tag{12}$$

where α is a step-size or learning rate hyper-parameter and W^* signifies the model's learn-able parameters. Using batches of a reasonable size both increases training speed and provides a means of regularizing training, preventing wild and sporadic parameter updates as individual examples may differ significantly. Further, we experiment with another SGD-based training algorithm which smooths updates between batches using momentum, *Adam* (Kingma and Ba 2014).

2.4 Generative sampling

Since our motivating task is natural language generation, we additionally need a means of generating a sequence from our model. In our experiments, we take in a variable length sequence of words $w_1...w_t, t >= 1$ representing an incomplete sentence, and generate a sequence of words $w_{t+1}...w_T, T >= t + 1$ to complete the sentence. We experiment with two straight-forward means of generating sequences from our incomplete input sequence.

In *argmax* sampling, we use our model and input sequence $w_1...w_t$ to generate a probability distribution for the $P(w_{t+1}|w_1...w_t)$. Given this conditional distribution, we choose the next word as the most likely word from this distribution:

$$w_{t+1} = \arg\max_{w \in V} P(w|w_1...w_t) \tag{13}$$

We then append this word to our incomplete sequence and repeat this sampling technique until we reach the end of the sequence (a period).

In cases when our model's generations are not diverse enough, we use a stochastic sampling technique, in which instead of choosing the most likely word, we sample a word from $P(w|w_1...w_t)$, treating it as a categorical distribution.

2.5 Results and Adjustments

Given this general model architecture, set of training algorithms, and generative sampling techniques, we can produce and compare a variety of language models and their efficacy as a generative model. Below we look at two of our better language models, discuss their strengths and weaknesses, and move on to our motivating task, recipe generation.

2.5.1 Google billion words language model

We trained a language model on the GBW data set as described previously, seeing only limited success. Table 1 shows the hyper-parameters used in our model.

Table 1: Hyper-parameters for our GBW language model

HYPER-PARAMETER	VALUE
Word embedding size (d)	100
Hidden state size (h)	300
Learning Rate	(varies by epoch) $10^{-6} \leq lr \leq 5^{-3}$
Training algorithm	*Adam*
Drop probability	0.1
Dropout placement	After LSTM

We evaluate our language models on their *perplexity*, which is the exponentiation of *cross entropy loss* and measures how well a particular sample is predicted by a probability distribution. Table 2 shows our perplexity across all examples in each data partition. Our perplexity scores indicate some over-fitting, as training perplexity is significantly lower than test perplexity.

Table 2: Average perplexity across our data partitions

Train Perplexity	143.27
Valid Perplexity	228.64
Test Perplexity	229.81

We produce generations from this model by choosing random examples from each data partition and using a prefix of random length as input, generating the conclusion of the sentence with *argmax* sampling. Ellipses indicate a generation did not conclude with a period. Below are some generations from the validation set:

Input: Even
Generated conclusion: the assumption that the UNK is not the only way ...
True conclusion: your parents would probably agree with me .
Input: Blake finished with 17 assists

Generated conclusion: and a UNK UNK in the UNK minute .
True conclusion: .
Input: You
Generated conclusion: UNK UNK , UNK , UNK , UNK , UNK ...
True conclusion: have certainly turned things around since then .

We can see our model did learn some grammatical coherency, and even occasional recognition of context, as it when given a prefix about a sports game, its generation matches the context. However, our generative capabilities with this model are certainly limited, as we can see when only the prefix ”You” is given, and the model repeats only the unknown token.

In the interest of moving on to our motivating task of recipe generation, we decided to train a language model on our data set of recipe instructions, which we detail below:

2.5.2 Language modeling recipe instructions

We train a language model on sequences of recipe instructions. Each set of instructions for an example recipe can encompass more than one sentence and up to three hundred words. This increase in long-term dependencies provides a modeling challenge, but our reduced vocabulary of only six thousand words in a more specific domain of language relaxes some of the difficulties in modeling the GBW corpus. Table 3 gives the hyper-parameters used for this model.

Table 3: Hyper-parameters for our recipe language model

HYPER-PARAMETER	**VALUE**
Word embedding size (d)	300
Hidden state size (h)	800
Learning Rate	10^{-5}
Training algorithm	*Adam*
Drop probability	0.4
Dropout placement	Before/After LSTM

Table 4: Average perplexity across our data partitions

Train Perplexity	23.85
Valid Perplexity	24.12
Test Perplexity	24.21

As we can see in Table 4, our language model experiences less over-fitting, as we perform more or less equivalently across data partitions. To extract generations from our model, we again use a randomly sliced input fragment with *argmax* sampling to produce a generation. While they are somewhat interpret-able, our model’s generations incur frequent grammatical mistakes and are by no means effective instructions for producing a recipe. This motivates us to look for improved recipe generations in sequence-to-sequence models, which make use of an learned encoding of contextual input information, in our case the recipe’s title and ingredients.

Input: To make the
Generated conclusion: wrapper , lots of the beef , onion , kosher lime juice , cornstarch Prick and incorporate in a bowl , until the chicken is all UNK ! jam the fish to make your pico crust ...
True conclusion: marinade , combine the yogurt , olive oil , minced garlic , oregano , salt , and some freshly cracked pepper in a bowl . Use a fine UNK cheese grater or a UNK to scrape the thin layer of yellow zest from the lemon skin into the bowl ...
Input: Wash the spinach
Generated conclusion: are been peppers and place them seeds in a food processor slow cooker chard ...

True conclusion: Wash the spinach well and cook very briefly in half a cup of water until wilted . Drain well , squeeze dry and roughly chop ...

3 Building a sequence to sequence model for recipe generation

3.1 Problem and data formulation

Recipe generation is a particular problem in natural language generation in which the modeling goal is to produce a procedural recipe given context such as the recipe's title, ingredients, and quantities. It follows a general pattern of sequence-to-sequence problems, where given an input sequence $w_1...w_{t_1}$, our goal is to generate a meaningful output sequence $w'_1...w'_{t_2}$. In machine translation for example, given a sequence in one language, the goal is to output a semantically equivalent sequence in another language. In our approach to recipe generation, we read in a sequence $w_1...w_{t_1}$ which encodes a recipe's title and ingredients. Our modeling goal is to output a natural language sequence $w'_1...w'_{t_2}$ providing procedural instructions for preparing a complete dish.

3.1.1 Data

Our data set consists of over eighty thousand full recipes, each of which forms a training example. We partition the recipes into a training set of sixty thousand examples and validation/test sets of ten thousand examples each. For each recipe we begin our input sequence with the title, followed by the ingredients, delimited with *beginIngredients* and *ing* tokens. Our output sequence is simply the procedural instructions delimited by *step* tokens, where each sentence in the recipe's procedure is assumed to be a single instruction. Together, our input and output sequence form an example like so:

Input: $w_1...w_i$, *beginIngredients,* $w_{i+1}...w_{i+k}$, *ing, ... ing ...*w_{t_1}
Output: $w_1...w_i$ *step, ... step, ...*w_{t_2}

We chose to reduce the vocabulary to only include words which occur in the data set ten or more times, resulting in a vocabulary size of six thousand one hundred eight.

3.2 Model Formulation

We follow the general design architecture of the RNN-based encoder-decoder for sequence to sequence learning (Sutskever et. al. 2014). The encoder-decoder is a pair of RNN based sub-networks which encode the input sequence into some intermediate state which is used to decode the output sequence.

3.2.1 The encoder

The modeling task for our encoder is to take the input sequence $w_1...w_{t_1}$ and return a standardized representation of its contents, in our case a fixed-length vector, which can be used by the decoder to generate the output sequence. Just as in our language model, we learn an embedding representation for each word in $\mathbb{R}^D$. These learned embeddings are shared between the encoder and decoder.

Our encoder uses an LSTM to read in the sequence of input word embeddings and output a sequence of hidden states $h_1...h_{t_1}$, where each is a vector in $\mathbb{R}^H$ and H is the hyper-parameter for LSTM hidden size. Thus, h_{t_1} represents our full encoding of the input sequence, which is passed on to the decoder.

3.2.2 The decoder

Although a language model is not always applied to sequence generation, we saw in the previous section how one could train a language model with *cross entropy loss* over words and sample from its distribution to produce an output sequence. We will take use this approach again here, as one can notice that our decoder architecture is almost identical to that of our language model. The decoder's task at training time is to take in two inputs: (1) the final encoding of the input sequence h_{t_1}, and (2) a portion of the output sequence $w_1...w_i$ for which it produces a probability distribution over V for the next word in the output sequence $P(w_{i+1}|w_1...w_i)$.

We accomplish this using using essentially the same architecture as our language model, in which we first retrieve word embeddings for each word in the supplied portion of the output sequence. The key distinction is that when passing the sequence of embeddings into the LSTM, our input sequence encoding h_{t_1} is used the as the initial hidden state in the LSTM update cycle. This allows our encoding of the input sequence to influence the eventual outputted probability distribution for each word in the output sequence. Like before, our sequence of decoder hidden states $h_1...h_{t_2}$ is used with a linear projection and *softmax* function to produce a sequence of probability distributions $P(w_2|h_1, w_1)...P(w_{t_2}|h_{t_2}, w_1...w_{t_2-1})$ where $h_1 = h_{t_1}$. Since we our using these probability distributions to generate our output sequence, we can train our encoder-decoder with *cross entropy loss*. Figure 2 gives an overview of our encoder, decoder, and their relationship.

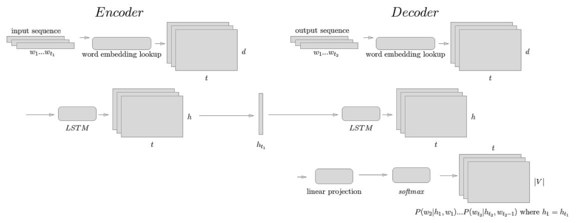

Figure 2: An overall look at our encoder decoder. The encoder uses an LSTM to produce a fixed length vector encoding the input sequence (title and ingredients). The decoder uses this encoding and the output sequence to produce a sequence of distributions for next words over the vocabulary. These distributions can be penalized with cross-entropy loss for training and can be sampled from in order to produce output generations.

3.2.3 Training and Sampling

Because our decoder produces a sequence of probability distributions, we can train the model with *cross entropy loss*, as we did for the language model and use the same sampling techniques to produce our output generation of the recipe's instructions. Doing so increases the likelihood of sampling the target sequence.

In the language modeling context, cross-entropy loss makes good sense as a training objective, as the purpose in language modeling is to directly model the distribution which we are training with cross-entropy. However, in a generative context such as recipe generation, our goal is to produce a semantically correct output sequence, in our case a set of natural language instructions a human being could reasonably interpret and use to produce a meal given the input ingredients. This is problematic because equivalently useful instructions may have wildly different cross-entropy loss. For example, if a single word which does not affect semantic meaning is inserted early in an otherwise perfect matching generation, every word following the inserted word is penalized as if the generation were incorrect. This is a major drawback for using *cross-entropy loss* in this generative context, but it can still produce reasonably effective models. Given its drawbacks, alternative loss functions and means of training natural language generation models are an active area of research, which we later explore in applying reinforcement learning objectives to training encoder-decoder models.

3.3 Results and Adjustments

3.3.1 First encoder-decoder model

With the first sequence-to-sequence model we made no architectural variations to our general model definition. We used training examples where the title and ingredients are presented in a sequence beginning with the title, followed by ingredients delimited with ing tokens. The output for each is a complete recipe sequence of no more than three hundred words.

We trained our model for sixty epochs over ten-thousand examples. On completion, we were able to achieve a test perplexity of 27.7 as seen in Table 6.

Table 5: Hyper-parameters for our first model

HYPER-PARAMETER	**VALUE**
Word embedding size (d)	200
Hidden state size (h)	512
Learning Rate	10^-5
Training algorithm	*Adam*
Input	title and ingredients sequence
Output	full recipe

Table 6: Average perplexity across our data partitions

Train Perplexity	7.07
Valid Perplexity	29.79
Test Perplexity	27.76

Right away, we can see that we've dramatically over-fit as our test and validation perplexities are significantly higher than our training perplexity. This is further shown in comparing our cross-entropy loss for training and validation sets across each epoch, shown in Figure 3.

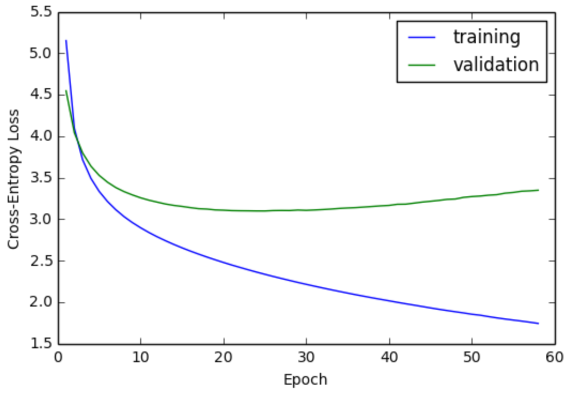

Figure 3: *Cross-entropy loss* on the training and validation sets across each epoch. After twenty epochs, we notice an increase in validation loss and a decrease in training loss, indicating over-fitting.

We can also examine our model's generative capabilities. We use *argmax* sampling to produce a recipe for a given input sequence of recipe title and ingredients. We generate a fixed number of words, which can include an end token, indicating that the recipe is concluded. Below is one generation from each data fold:

Input: korean beef and noodles ... (train)
Generated Output: in a large skillet , heat oil over medium - high . add onion and cook , stirring , until softened , about 5 minutes . add garlic and ginger and cook , stirring ...
True: to prepare beef , sprinkle cornstarch over beef ; toss to combine . add 1 tablespoon soy sauce and next 4 ingredients ...
Input: huevos rancheros ...(valid)
Generated Output: preheat the oven to 350 degrees . cut the bread into 1 / 2 inch slices . in a large bowl , mix together the ground beef ...
True: heat a tablespoon or two of oil in a large skillet over medium heat . add tortilla and cook on both sides until just lightly browned and crispy . remove to plate. ...

Input: slow cooker charro beans ... (test)
Generated Output: 1 . in a large saucepan , cook the rice noodles according to the package directions . drain and set aside. ...
True: place beans in a colander , rinse well , and remove any stones or shriveled beans . cook bacon until just crispy. ...

There a few things worth considering with the generations from this model. The first is that generations generally use correct grammar and syntax, indicating a reasonable ability to model language. Another is that the generated recipes seem to be internally consistent. If we heat oil in a skillet, we soon add ingredients that are reasonable to fry in a pan, and after cooking noodles, we drain them. The generations also show good diversity, a problem we will face in our later models. For weaknesses, the generations tend to focus only on very common ingredients. More importantly, rarely correlate with input, especially in the validation and test sets. We can see this in the example from the validation set in which the model completely misunderstands how to make huevos rancheros, and instead generates a recipe for a baked dish.

These generations are coherent but reflect significant over-fitting. As such, we begin making adjustments to our model to correct for some of the problems we discover here. The first problem we address is our significant over-fitting, which we address using Dropout for regularization in the model.

3.3.2 Addressing over-fitting with dropout

To handle over-fitting, we employ dropout as a regularization technique. During training, a dropout module sets each element in the input tensor to zero with probability p where p is a hyper-parameter. Dropout's success has been shown empirically in previous work, and one can intuitively see how it forces a model to make effective use of more of it's parameters and reduces its ability to memorize training examples (Srivastava et. al. 2014). We chose to place a dropout module before and after each LSTM in both the encoder and decoder with a drop probability of 0.4 or 40%. We also show an increase in hidden size (H) and embedding size (D) which provide more parameters for the model to use to compensate in model capacity for the dropout regularization.

Table 7: Hyper-parameters for our model with added dropout

HYPER-PARAMETER	**VALUE**
Word embedding size (d)	300
Hidden state size (h)	800
Learning Rate	10^-5
Training algorithm	*Adam*
Input	title and ingredients sequence
Output	full recipe
Drop probability	0.4
Dropout placement	Before and after LSTM modules

Using this approach, we achieve a more balanced train to test perplexity ratio as well as a reduction in test perplexity, as can be seen in Table 8. As we can see in Figure 4, we additionally see a more consistent decrease in validation loss over time.

Table 8: Average perplexity across our data partitions

Train Perplexity	18.66
Valid Perplexity	19.21
Test Perplexity	19.31

When examining our generations using *argmax* sampling, an unfortunate and clear problem becomes apparent:

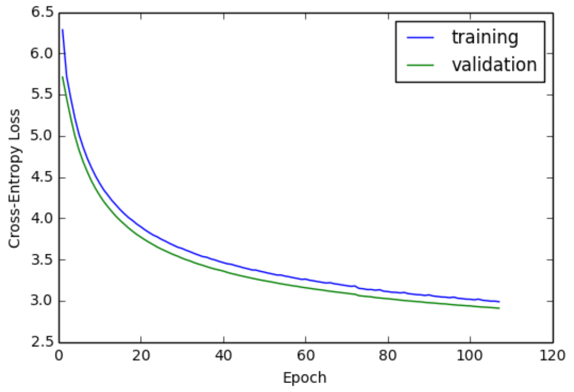

Figure 4: *Cross-entropy loss* on the training and validation sets across each epoch. With dropout, we no longer see the increase in validation loss that was indicative of over-fitting.

Input: Green Smoothie ... (train)
Generated Output: 1 . Preheat oven to 350 degrees . In a large bowl , combine the flour , baking powder , salt , and salt . In a large bowl , whisk together the flour , baking powder ...
True: Put all ingredients in a blender , and blend until smooth.
Input: Egg White Omelette With Spinach ...(valid)
Generated Output: 1 . Preheat oven to 350 degrees. 2 . In a large bowl , combine the flour , baking powder , salt , and pepper . In a large bowl , whisk together the flour , baking powder ...
True: Wash the spinach well and cook very briefly in half a cup of water until wilted . Drain well , squeeze dry and roughly chop . Lightly beat the 2 whole eggs and 4 egg whites together with chopped chervil or parsley ...
Input: South Indian Style Lemon Pickle Powder ... (test)
Generated Output: 1 . Preheat oven to 350 degrees . In a large bowl , combine the flour , salt , pepper , and salt . In a large bowl , combine the flour , baking powder ...
True: Here we have 4 lemons so , cut 4 lemons into small bite size pieces . And Squeeze 2 lemon . Pour the squeezed lemon juice over the lemon pieces . Using wooden ladle ...

Our generations from this model show almost no diversity when using *argmax* sampling, and have nothing to do with the input sequence, despite improvements in test perplexity. This motivates us to consider a few changes, the first being use of stochastic sampling in generating our output sequence.

Recall that the final step in producing the sequence of valid probability distributions we sample from is a *softmax* calculation. A variant of the *softmax* calculation includes a temperature parameter τ :

$$p(x_k) = \frac{exp(\frac{x_k}{\tau}))}{\sum_{j=1}^{|V|} exp(\frac{x_j}{\tau}))} \tag{14}$$

As $(\tau \rightarrow \infty)$, all words in the vocabulary become equally probable and generations become uniformly random. As $(\tau \rightarrow 0^+)$, the most likely word dominates the rest, so that sampling from the resulting distribution is equivalent to *argmax* sampling. We explore stochastic sampling for temperature values between zero and one, and see increased diversity but no improvement in our generations:

Input: Green Smoothie ... (train)
τ = 1: Preheat an oven to 350 degrees F F F. Line slow plastic pan with a little water ; set aside for 10-15 minutes. ...
τ = 0.2: Mix together and sesame seeds. Set aside. Wipe mix. Heat a few olive oil over medium heat , and season to coat both sides of oil and sprinkle with garlic powder. ...
True: Put all ingredients in a blender , and blend until smooth.
Input: Egg White Omelette With Spinach ...(valid)

τ = 1: Boil potatoes and cook at medium . Drain well in first 5 to high high . Stir briskly with juice , cumin , salt , and pepper. ...

τ = 0.2 Preheat oven to 375 degrees . Add the hands to the lowest bacon and let stand for about 30 minutes . Lightly peanut the coconut oil and saffron. ...

True: Wash the spinach well and cook very briefly in half a cup of water until wilted . Drain well , squeeze dry and roughly chop . Lightly beat the 2 whole eggs and 4 egg whites together with chopped chervil or parsley ...

Input: South Indian Style Lemon Pickle Powder ... (test)

τ = 1: Preheat oven once to make an UNK spray . Place the salmon (about a skillet in UNK) and bake for 4-5 minutes or until soft (5 to 10 "). ...

τ = 0.2: 1 . Heat oil in a skillet over medium heat , and cook 8 minutes or until tomatoes is foamy but they are brown on the bottom of kitchen . ...

True: Here we have 4 lemons so , cut 4 lemons into small bite size pieces . And Squeeze 2 lemon . Pour the squeezed lemon juice over the lemon pieces . Using wooden ladle ...

Within these generations we can notice the effect of changing temperature on grammatical coherency, but neither temperature provides useful generations. This trade-off between over-fitting and generation quality proves to be a recurring significant challenge for our model, which leads us to hypothesize that the encoder's signal from the input sequence is not being used properly in the decoder.

We attempt several adjustments to our model and training regiment that are worth mentioning but saw limited success. (1) We use a stochastic beam search, which uses beam-search with our stochastic sampling method, maintaining a beam of the k most likely sequences. This maintains generation diversity while improving grammatical coherence, but does not improve relationship of generation to input sequence. (2) We explore a variety of hyper-parameters and modeling tweaks, such as lowering dropout, increasing or decreasing the hidden or embedding size, and using dropout in different locations. Each of these affects generation quality, but fails to improve the relationship between input sequence and generation. (3) We also explore word dropout for decoder inputs, which have been used successfully in other RNN decoders which make use of latent variables (Bowman et. al 2016). In the instruction sequence passed to the decoder $w_1...w_{t_2}$, we replace a word w_i with the UNK token with probability $p = 0.2$. Our motivation for this was to encourage the model to make better use of the encoder input, as we've reduced its capacity to minimize loss by simply using the decoder as a language model. This proves useful in later models, but not in our encoder-decoder as it has been introduced so far. (4) Finally, we use a bag-of-words encoder, which instead of passing a sequence of word embeddings through an LSTM, adds them together to form a single vector in $\mathbb{R}^D$, which is used as the first hidden state in the decoder LSTM. Likewise, this does not make a significant improvement in generation quality.

Having tried these adjustments and not finding significant improvements, we decided to build an attention-based encoder-decoder in hopes that attentive access to the ingredients might improve the model's ability to make effective use of of the input sequence.

3.4 Introducing attention over ingredients to the encoder-decoder

To improve our model's use of the title and ingredients in producing a generation, we introduce an attention mechanism over the ingredients. For this to work best, we begin with some transformations to our data set.

3.4.1 Preparing the data

Instead of combining our title and ingredients into a single input sequence with delimiting tokens, we instead consider a sequence for the title and a sequence for each ingredient. For each ingredient sequence, we remove quantities and stop-words, retaining only a sequence of at most six words. We choose to consider only the first thirteen ingredients in a recipe to reduce necessary computation, adding padding sequences where necessary.

3.4.2 Attention over ingredients

In our previous model, the decoding LSTM is given its previous cell state $C_{t-1} \in \mathbb{R}^H$, its previous hidden state $h_{t-1} \in \mathbb{R}^H$, and a single word embedding $W_t \in \mathbb{R}^D$, where it outputs h_t, which is used to compute the distribution for $P(w_t|ht-1, w_{t-1})$. In this model, we additionally give the decoding LSTM an encoding of the title $T \in \mathbb{R}^H$ and an attention over the ingredients $a_t \in \mathbb{R}^H$, by concatenating them to the input word embedding W_t. We will describe how T, a_t are computed and used below:

Using the same encoder architecture described in the previous section, we compute an encoding for the title $T \in \mathbb{R}^H$ by using the words in the title $w_1...w_t$ in the encoder and using the final hidden state h_t ans our value for T. Unlike our title encoding, a_t is computed at every decoding time-step. We begin by embedding the words in our ingredients in an embedding space $\mathbb{R}^{D'}$ distinct from the embedding space for title and output sequence words. We use a separate embedding space to give our model the capacity to differentiate a word's context in an ingredient list and in a generation. For our computations, $D' = H$, but this can easily be generalized with the addition of a linear projection to models where this is not the case. We then add these ingredient embeddings together so that each ingredient is represented by a single vector in $\mathbb{R}^{D'}$. This gives us an $N * D'$ ingredient matrix ING where N is the number of ingredients. At each time step, we compute our attention over ingredients a_t as follows:

$$a_t = \textit{softmax}(ING * h_{t-1}) * ING \tag{15}$$

Essentially, we use the previous hidden state h_{t-1} to decide what information to extract from our representation of the ingredients by using a soft adaptive selection.

Finally, we concatenate T and a_t with the decoder input word embedding to form a single input vector $x_t \in \mathbb{R}^{D+H+H}$. Figure 5 provides an overview of the encoder-decoder model with attention over ingredients. In our old approach, the input sequence, a concatenation of the title and all ingredients, was encoded using an LSTM-based encoder, which gave us a single encoding of the entire input sequence $h_{t_1} \in \mathbb{R}^H$. Furthermore, this signal from the input sequence was only given to the decoder at the beginning of generation, making it difficult for it to make any reference to the input across long spans in the output sequence. In our new approach, the decoder is given the encoding of the title at every time-step, allowing it to refer back as necessary. Similarly, our attention mechanism over ingredients allows the decoder to learn what portions of the input sequence of ingredients are relevant at different time-steps.

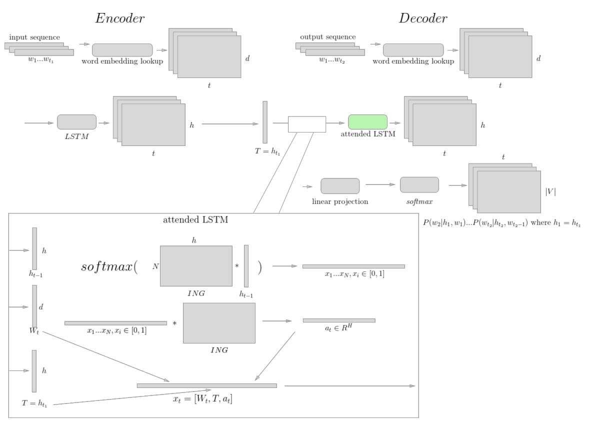

Figure 5: An overview of the encoder-decoder model with attention over ingredients and available title encoding. Within the LSTM, each time-step computes an attention feature over an ingredient matrix for the recipe.

3.4.3 Results using attention

We train our attention model until it reaches our stoppage criteria of two consecutive epochs with increasing validation loss, running for over 350 epochs, requiring more than 120 hours of training. Our hyper-parameters can be seen in Table 9.

Table 9: Hyper-parameters for our attention model

HYPER-PARAMETER	**VALUE**
Word embedding size (d)	100
Hidden state size (h)	512
Learning Rate	$10^{-}5$
Training algorithm	*Adam*
Input	title sequence and first thirteen ingredient sequences
Output	full recipe
Drop probability	0.35
Dropout placement	Before and after LSTM modules
Ingredient embedding size	512
Word Dropout (decoder input)	0.2

In terms of perplexity across each data set, our model performs slightly better than our best attention-less encoder-decoder, as can be seen in Table 10. Again, we do not see the signs of over-fitting we saw in our earliest models.

Table 10: Average perplexity across our data partitions

Train Perplexity	17.38
Valid Perplexity	18.22
Test Perplexity	18.28

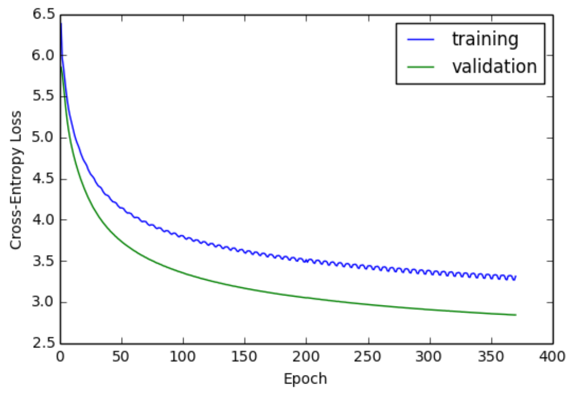

Figure 6: *Cross-entropy loss* on the training and validation sets across each epoch. We see a steady decline in both training and validation loss across 350 epochs.

Finally, we can look at this model's generative ability. We return to using *argmax* sampling and see more success and diversity than in our previous models.

Input: California Chicken Wrap ... (train)
Generated Output: 1 . Combine first 4 ingredients in a large bowl . 2 . Place tortillas on a baking sheet . Bake at 350 degrees for about 30 minutes . Remove from oven and let stand for 10 minutes . Serve with tortilla chips .

True: Spread Mayonnaise Dressing with Olive Oil on tortillas . Layer chicken , avocado , red pepper , red onion and salad greens down center of each tortilla . Roll and fold the filled tortillas .

Input: Mexican Punch ... (train)

Generated Output: 1 . In a large bowl , combine the orange juice , sugar , sugar , and sugar . Add the UNK and stir to combine . Add the remaining ingredients and mix well . Pour the mixture into the prepared pan and bake ...

True: In a large pitcher , UNK the limes , lemons and orange . Add the tequila , simple syrup , lime juice , UNK and UNK . Stir well and add ice .

Input: Beef and Ravioli Soup ...(valid)

Generated Output: 1 . Preheat oven to 350 degrees F. In a large skillet , melt butter over medium heat . Add beef and cook until browned , about 5 minutes . Add beef and cook until browned , about 5 minutes ...

True: Toss sirloin , and 2 tablespoons flour in bowl . Melt 2 tablespoons butter in large Dutch oven over medium-high heat . Cook until beef is browned ...

Input: Honey Ginger Shrimp ...(valid)

Generated Output: 1 . Heat oil in a large skillet over medium heat . Add onion and cook until fragrant , about 5 minutes . Add garlic and cook until fragrant , about 5 minutes . Add the shrimp and cook for about 5 minutes ...

True: Heat the olive oil and red pepper flakes in a large skillet over medium heat . Add the onions , garlic , ginger and honey ; cook and stir until fragrant . Add the shrimp , and cook for 5 minutes ...

Input: Dark Chocolate Mousse with UNK Orange ... (test)

Generated Output: 1 . In a large bowl , whisk together the egg yolks , sugar , and sugar . Add the egg yolks and beat until smooth . Add the egg yolks and beat until smooth. ...

True: PREPARE THE UNK , MAKE THE UNK : Whisk the orange juice , sugar , and yolks in a medium mixing bowl set over a small pot of simmering water for 10 minutes , scraping the sides of the bowl often , until the liquid has become thick and tripled in size ...

Input: Zucchini Bread ... (test)

Generated Output: Preheat oven to 350 degrees . In a large bowl , combine the flour , baking powder , baking powder , salt , and salt . In a large bowl , whisk together the flour ...

True: Preheat oven to 350 degrees F. Spray a 9 X UNK loaf pan with nonstick spray . In a large bowl , whisk together flour , salt , baking powder , cinnamon and brown sugar ....

Looking at these generations, we can see a strong qualitative improvement in the relationship between the inputted title and ingredients and the resulting generation. Recipes which involve baking typically start with preheating the oven, and instructions that begin with heating oil in pan are often followed by the frying of relevant ingredients. Still, our generations are far from perfect. We can see our model frequently repeats whole sentences as it gets further into the generation. Furthermore, while it tends to use common ingredients when relevant, it doesn't often use key uncommon ingredients, such as zucchini in zucchini bread. While our model seems aware of cooking context, it struggles with long term dependencies in the procedural agenda, such as failing to put the filling inside the tortillas when making chicken wraps. Other approaches have been proposed for handling these long term dependencies such as the neural checklist model, which uses attention mechanisms to track both what ingredients need to be mentioned and those which have already been used in the generation (Kiddon et. al. 2016). We hope to improve our generations and their management of procedural dependencies through reinforcement learning.

3.4.4 Reinforcement training objective

Reinforcement learning is an active area of research in both deep learning and artificial intelligence (Sutton and Barto 2012). Here the problem is re-framed as a Markov decision process (MDP), with a state space S a set of actions A available at each state $s \in S$, a transition function $T(s, a, s')$, and a reward function $R(s, a, s')$. The transition function provides a distribution over possible resulting states s' given that the model performed action a in current state s. The reward function gives a value to to each transition. In reinforcement learning, neither T or R are known ahead of time.

We can consider recipe generation as an MDP, where the state space S is all strings composed of words in the vocabulary V, the action space is the generation of a of the next word w_i given a prefix sequence $w_1 ... w_{i-1}$ and any other contextual information like an encoder hidden state, etc. The transition depends on our model and sampling technique, but for our encoder-decoder models with *argmax* sampling is deterministic. The reward function is our design choice, which we use to alter the loss signal in back propagation to encourage generations with better rewards. Here we propose one possible reward function, with plenty of room for future work.

For each training example, we generate a complete sequence up to a fixed length L, and then reward it according to the length of the *greatest common sub-sequence (GCS)* between our generation and the target output:

$$R(s, a, s') = \begin{cases} 0 & |s'| < L \\ GCS(s', s_{true}) & |s'| \geq L \end{cases} \tag{16}$$

In order to use this reward to train our learn-able parameters, we use a strategy in which the model produces two competing generations s_1, s_2 which are each given a reward r_1, r_2. We continue to generate sequence pairs until the difference between r_1, r_2 exceeds some threshold (a hyper-parameter). Finally we update the learning rate lr for this example to reflect the relative difference between r_1, r_2 and the average reward difference avg from previous training examples:

$$lr = lr * (\frac{|r_1 - r_2|}{avg}) \tag{17}$$

The motivation for using *GCS* with this competing generation strategy is that a training example for which there is a significant difference in reward represents an example where we have tractable progress to make in learning. By contrast, an example in which the reward difference is always low could be an example where we have already converged to a good generation, or it could be an outlier or poor quality example from which we can not expect a quality generation. The sampling technique in which we create multiple generation pairs significantly increases training time, and so due to timing constraints, the effectiveness of our reward choice and training strategy is not fully explored.

3.5 Evaluating our generative models

So far, we have completed a walk-through of the process we embarked on to build models for generating recipes, and have seen many of the challenges and successes along the way. Here we provide a more careful comparison of the models we built and their success in addressing our problem. We evaluate each model using BLEU, which while more apt for machine translation, is still an effective and available metric for evaluating our generations. We consider each generation from our model as a candidate sequence and the true instructions from the example to be a trusted reference sequence. We create seventy-five generations of at most two hundred words each, and compute multi-BLEU for each relative to the first two hundred words in the true output. Our results can be seen in Table 11.

Table 11: BLEU scores for each encoder-decoder model for recipe generations. Models are named with following the pattern [RNN Type]-[Word size]-[Hidden size] and are in order of appearance.

LSTM-200-512	2.41
LSTM-300-800 (dropout)	1.89
attn-LSTM-100-512	3.54

As we can see, overall BLEU scores are low, which is common for long sequences. Our model with attention over ingredients outperformed others, matching our intuition when qualitatively analyzing its generations.

4 Discussion

As we can see from the models we have produced, successful recipe generation is a challenging problem. In ours and related work, one can see a lot of promise in RNN-based deep learning models using a sequence to sequence approach for this task, especially those which include attention over part or all of the input sequence. We propose further experiments with reinforcement learning methods as a significant avenue for future work, with its own set of challenges to face, including the sparsity of a reward signal, and the increased training time required to calculate some rewards.

5 References

Bowman, S.R., Vilnis, L., Vinyals, O., Dai, A.M., Jozefowicz, R., and Bengio, S. (2015). Generating Sentences from a Continuous Space. arXiv:1511.06349 [Cs].

Hochreiter, S., and Schmidhuber, J. (1997). Long Short-Term Memory. Neural Comput. 9, 17351780.

Kiddon, C., Zettlemoyer, L., and Choi, Y. (2016). Globally Coherent Text Generation with Neural Checklist Models. Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing 329339.

Kingma, D.P., and Ba, J. (2014). Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [Cs].

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research 15, 19291958.

Sutskever, I., Vinyals, O., and Le, Q.V. (2014). Sequence to Sequence Learning with Neural Networks. arXiv:1409.3215 [Cs].

Sutton, R., and Barto, A. (2012). Reinforcement Learning: An Introduction (The MIT Press).