5-2022

Investigating Vector Space Embeddings for Database Schema Management

Goldy Malhotra
gxm6116@rit.edu

Investigating Vector Space Embeddings for Database Schema Management

by

Goldy Malhotra

THESIS

Presented to the Faculty of the Department of Computer Science

Golisano College of Computer and Information Sciences

Rochester Institute of Technology

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Science

Rochester Institute of Technology

May 2022

Abstract

Investigating Vector Space Embeddings for Database Schema Management

Goldy Malhotra, M.S.
Rochester Institute of Technology, 2022

Dr. Michael Mior, Supervisor
Dr. Alexander G. Ororbia II, Reader
Dr. George Zion, Observer

Text generation in the area of natural language processing as part of the artificial intelligence field has been greatly improving over the last several years. Here we examine the application of vector space word embeddings to provide additional information and context during the text generation process as a way to improve the resultant output through the lens of database normalization. It is known that words encoded into vector space that are closer together in distance generally share meaning or have some semantic or symbolic relationship. This knowledge, paired with the known ability of recurrent neural networks in learning sequences, will be used to examine how vectorizing words can benefit text generation. While the majority of database normalization has been automated, the naming of the generated normalized tables has not. This work seeks to use word embeddings, generated from the data columns of a database table, to give context to a recurrent neural network model while it learns to generate database table names. Using real world data, a recurrent neural network based artificial intelligence model will be paired with a context vector made of word embeddings to observe how effective word embeddings are at providing additional context information during the learning and generation processes. Several methods for generating the context vector will be examined, such as how the word embeddings are generated and how they are combined. The exploration of these methods yielded very promising results in line with the overall goals of the performed work. The benefit of incorporating word embeddings to supply additional information during the text generation process allows for better learning with the goal of generating more human-useful names for newly normalized database tables from their data column titles.

Table of Contents

List of Figures

Chapter 1

Introduction

A frequent process when formulating a database schema or improving an existing one is database normalization. When database normalization is performed, existing tables are reorganized into a set of new relational tables. These newly formed tables are assigned distinctive names, based on the data attributes being stored there. The data attributes are known as columns in databases and have names that do not change when performing normalization. The step of assigning new table names currently is done manually during the normalization process by the developer of the database schema. While much of the normalization process can be automated programmatically, the creation of these relevant and human useful names is generally by the database developer after normalization. Instead of relying on the database developer to assign names to the newly normalized tables, that step can be automated through the use of a neural network model trained for text generation. This thesis seeks to explore and evaluate the use of vector space embeddings with a recurrent neural network to create meaningful and coherent title names.

Words encoded into a high-dimensional vector space, called word embeddings, offer additional information about the relationship between some words. This additional information can be learned by a neural network model to improve its understanding of how words relate, which is especially helpful during natural language processing tasks such as text generation. This thesis examines the use of word embeddings to offer additional contextual information for words during the neural network learning process towards the generation of text.

The following section will briefly cover the prerequisite knowledge for this thesis work, namely the two primary concepts being tackled, the use of word embeddings and recurrent neural networks. Section 3 will cover the work in more detail, including the overall architecture of the built system. That section will also cover the training data preprocessing and cleaning, as well as the design and training processes for the recurrent neural networks being used. Section 4 will cover the evaluation system as well as the custom evaluation metric being used to properly evaluate the model for both quantitative and qualitative performance. Section 5 will explore and discuss the results of this thesis work. The penultimate section, Section 6, will cover related work already being explored as well as possible future work that can be drawn from the work described in this paper. Finally, Section 7, will cover a conclusive overview of the work performed as well as the results obtained.

The contributions made in this work primarily focus on extending RNN-based architectures to leverage context vectors made by combining word embeddings. This includes the testing of the effect of leveraging a context vector for text generation, the word embedding generation methods, and methods for combining word embeddings into context vectors. Additionally, the contributions include the creation of a Python language toolkit for dataset preprocessing and cleaning, model object creation, training, and evaluation, and for hyper-parameter optimization.

Chapter 2

Background

The thesis work is multidisciplinary, combining together the fields of artificial intelligence and data science. The neural network and word embedding techniques being used are from the natural language processing area of artificial intelligence. The training process, handling of the large training dataset, and possible implementations of data analysis pulls techniques from the big data aspect of data science. The two major technologies being explored through this thesis work are the use of vector space encodings for words, known as word embeddings, and the use of recurrent neural networks to learn natural language sequences for text generation. These two technologies are further described below.

2.1 Word Embeddings

Humans have a well learned understanding of words and their use, mostly through their immersion in language and its use. We understand the way words are put together in sentences and the way the meaning and context changes depending on the words used. The use of distributed representations of words, or word embeddings, allows for this understanding of relationships between words to be applied to neural network learning models. Instead of limiting the learning of the neural network through the use of other techniques, most of which offer “no notion of similarity between words” as stated by Mikolov et al. [12], word embeddings provide more information about each word by accounting for the connection between words. Instead of using words “in terms of discrete units that have no inherent relationship to one another,” words encoded into vector space can benefit from their similarity or relation to other words as “similar words are likely to have similar vectors” [14]. More concretely, these relationships between words in a continuous vector space is measured through metrics such as Euclidean distance or cosine similarity. Additionally, word embeddings are often encoded into higher dimensional vector space as more information can be attached to a single word, generally through the ability to provide multiple relations and additional semantic context. The increase in word embedding dimensionality also leads to increased computational complexity.

The process for forming distributed representations of words today is generally handled through the use of a trained neural network whose input layer weights for a given word input make up the word embedding itself. Some of the more popular models for embedding formation are Google’s Word2vec [12], Stanford’s GloVe [16], and Facebook AI Research’s fastText [3]. Mikolov et al. [12] proposed two major architecture’s for Word2vec, a “continuous bag-of-words model” (CBOW) and a “continuous skip-gram model” (Skip-gram). These proposed architectures, shown in Figure 2.1, take opposing paths on how they predict the output result. The CBOW model attempts to predict the output word given context words, while the Skip-gram model attempts to predict context words given a single word. The GloVe model draws from the Word2vec models but also incorporates statistical data of the training corpus, similar to global matrix factorization methods which "effectively leverage statistical information" but tend to "do poorly on the word analogy task" [16]. The fastText model is derived from the Word2vec model but learns on n-grams and instead seeks to "represent words as the sum of the n-gram vectors" [3]. Due to learning on n-grams, fastText also can attempt to formulate word embeddings for words it has never seen before, which is primarily why it is the proposed process for the word to embedding conversion for this thesis work. A different approach that is examined in this work is the idirectional ncoder epresentations from ransformers (BERT) model [5], also developed at Google. The BERT model proposed an entirely different architecture from the previously mentioned vector space encoding models, partially leveraging the Transformer encoder architecture. A Transformer is a deep learning model that functions by "relying entirely on an attention mechanism to draw global dependencies between input and output" [23]. The BERT model borrows the encoder portion of the Transformer architecture, featuring several layers of these bidirectional encoders in a single BERT model. BERT uses a masked language model (MLM) pre-training objective, where some words in the model input are masked and the model must predict them using only the context words, to train the bidirectional Transformer layers. The model is also trained at the same time using the next sentence prediction (NSP) task, where a pair of sentences are input and the model must predict if they are sequential sentences in the original source corpus. Because of these mechanisms, the BERT model creates sub-word representations of its training vocabulary, allowing it to generate word embeddings for out of vocabulary words, similar to the fastText model.

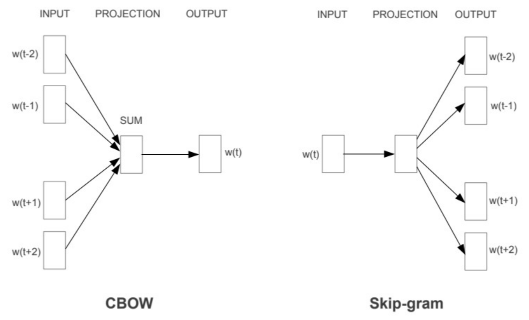


Figure 2.1: Mikolov et al. [12] Proposed Word2vec Architectures

2.2 Recurrent Neural Networks

Given the natural language processing task at hand, the most clear choice for neural network type was a recurrent neural network, or RNN, and its derivatives. Since most NLP tasks involve variable-length input and output sequences, the recurrent neural network with its ability to store and learn from past information is the most logical choice. Having the ability to learn the temporal dynamic behavior of language and communication is critical for text generation tasks. Language learning can be thought of as a time-series, where the connections between words or even characters is a reoccurring observation, which leads to the idea that once those connections are learned, new words and sentences can be formed. RNNs can take an input sequence, learn the connection between the pieces of the input sequence, and then use that learning to predict future possible values during word or sentence generation. RNN nodes contain a connection to themselves, effectively giving the architecture a form of “memory” [24] of past sequences it has seen before. This is the main feature that allows the model to learn sequential data, as the hidden state can be updated over individual time steps per sample. The hidden state and output equations at each time step for a RNN can be seen in Figure 2.3.

The major differences between other popular neural network architectures, such as convolutional neural networks, and RNNs, are the RNNs’ ability to learn through time using the backpropagation through time (BPPT) learning algorithm. Backpropagation is a very popular learning algorithm used during the neural network training process with another algorithm known as gradient descent, described further in Section 2.3, to update “the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector... and the desired output vector” [18]. Backpropagation is used to calculate the derivatives of the loss or error during a defined training period. These are then used to update the parameters of the model. BPTT is derived from the Backpropagation learning algorithm as an effort to apply it throughout the time-steps of a recurrent neural network, capturing the temporal qualities of the RNN architecture.

There do exist several possible problems with the use of recurrent neural networks. A major problem that can be faced is model instability, usually caused by when the loss gradient either explodes or vanishes. An exploding gradient is where the gradient term continually increases towards infinity, while a vanishing gradient is where it decreases rapidly to zero. Both of these result in instability for the model, resulting in the model only producing erroneous values, due to floating point arithmetic overflow or underflow, or not learning at all in some cases. Thankfully there exist variations of the RNN, primarily the LSTM (long-short term memory) and the GRU (gated recurrent unit), which improve upon the design of a simple RNN and attempt to alleviate the majority of the instability issues using gate-based architectures. The hidden state, gate, and output equations at each time step can be seen for these two architecture in Figures 2.4 and 2.5.

As explained previously, below are the forward propagation equations for the three architectures explored in this work. To fully grasp the function and use of these equation, a base knowledge of linear algebra and statistics is necessary, as the majority of variables involved are matrices and techniques often rely on probabilities. One primary operation is the matrix multiplication, represented using a $*$ in the equations below. Matrix multiplication involves two matrices of sizes m x n and n x p, respectively. The size of the columns of the first operand must match the size of the rows of the second. The elements of the rows of the first operand are multiplied by the element in the columns of the second operand, resulting in a matrix of size m x p. This operation is commonly used to handle learnable weights for inputs, but also to linearly map operands in cases of size mismatching for future operations like element-wise addition, where the two matrix operands must be the same size. Another common operation is element-wise multiplication, represented below by a $\cdot$, where the operands must be the same size as well. Additionally, an important operation when dealing with neural networks is activation functions. These functions are used to introduce non-linearity to the calculations in an effort to better fit the model to the input data set. Some of the activation functions used in the equations below are the sigmoid (σ) function and the hyperbolic tangent ($tanh$) functions. ϕ is also shown below, representing a developer selected activation function, generally $tanh$ or the rectified linear unit (ReLU) activation functions. A major function used in these architectures is the $softmax$ function, which is used to achieve the output result x_{t+1}. The calculations before the $softmax$ is applied results in unnormalized log probabilities, known as logits. This probability distribution is a categorical multinomial distribution over a set number of classes per time step. This categorical distribution represents the discrete probability distribution over k-classes for the next character in the sequence being predicted or generated. The $softmax$ function is applied to the logits to normalize them, resulting in probabilities over k-classes that sum up to 1. The $softmax$ function, shown for a single element x_i in Figure 2.2, normalizes by applying the exponential function to each element in the K length input vector and then dividing each element by the sum of all the exponentials for the input vector. A parameter β is used to influence the balance between probability classes and the uniformity of the distribution.

$$softmax(x)_i = \frac{e^{\beta x_i}}{\sum_{j=1}^{K} e^{\beta x_j}}$$

Figure 2.2: Softmax Function Equation

$$\begin{gathered} h_t = \phi(W * x_t + V * h_{t-1} + b_h) \\ \hat{x}_{t+1} = softmax(h_t * U + b_q) \end{gathered}$$

Figure 2.3: RNN Hidden State & Output Layer Equations

$$\begin{gathered} g_t = tanh((x_t, h_{t-1}) * W_g + b_g) \\ i_t = \sigma((x_t, h_{t-1}) * W_i + b_i) \\ f_t = \sigma((x_t, h_{t-1}) * W_f + b_f) \\ o_t = \sigma((x_t, h_{t-1}) * W_o + b_o) \\ state_t = g_t * i_t + state_{t-1} * f_t \\ h_t = o_t * \phi(state_t) \\ \hat{x}_{t+1} = softmax(h_t * U + b_q) \end{gathered}$$

Figure 2.4: LSTM Hidden State & Output Layer Equations

$$\begin{gathered} u_t = \sigma(x_t * U_x + h_{t-1} * U_h + U_b) \\ r_t = \sigma(x_t * R_x + h_{t-1} * R_h + R_b) \\ state_t = \phi(x_t * S_x + r_t \cdot (h_{t-1} * S_h)) \\ h_t = u_t \cdot h_{t-1} + (1 - u_t) \cdot state_t \\ \hat{x}_{t+1} = softmax(h_t * U + b_q) \end{gathered}$$

Figure 2.5: GRU Hidden State & Output Layer Equations

The primary goal of the LSTM architecture, as stated by Hochreiter and Schmidhuber, was to “construct an architecture that allows for constant error flow through special, self-connected units without the disadvantages" [7] during BPTT. Figures 2.6 and 2.7 show the flow of the gradient during BPTT. They also highlight how the error flow is broken up in the RNN node by the multiplication of the weight and the input stack, juxtaposed to the uninterrupted flow of the error in the LSTM node. The LSTM node features gate units to better control the plasticity and stability of the learning process. The first gate is known as the forget gate, as it controls how the cell state of the previous node affects the current node. The next gate unit, the input gate, is made up of two operations to incorporate the inputs to the current node into the cell state of said node. The final gate, the output gate, then controls how the cell state and inputs affect the output hidden state of that node.

The GRU, shown alongside the RNN and LSTM in Figure 2.8, uses a gated architecture similar to the LSTM. Instead of using three gates like the LSTM, the GRU features only two gates, the update and reset gates. Unlike the LSTM, the GRU does not feature a cell state, instead maintaining only a hidden state. The update gate primarily influences how much the previous state and input will affect the hidden state going forward. The reset gate influences how much of the previous information to remove or forget from the hidden state going forward. These gates together allow the GRU some of the benefits of the LSTM while being slightly more computationally efficient due to the lower number of overall computations.

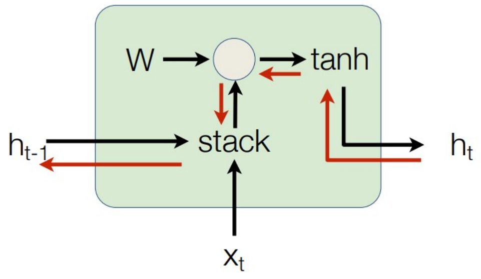


Figure 2.6: RNN gradient flow during BPTT [20]

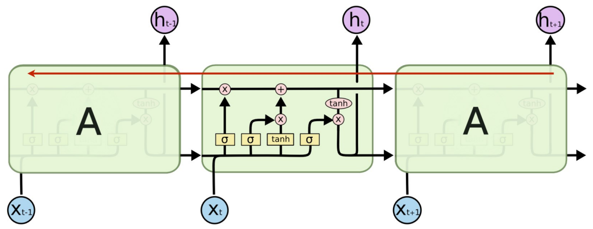


Figure 2.7: LSTM gradient flow during BPTT [20]

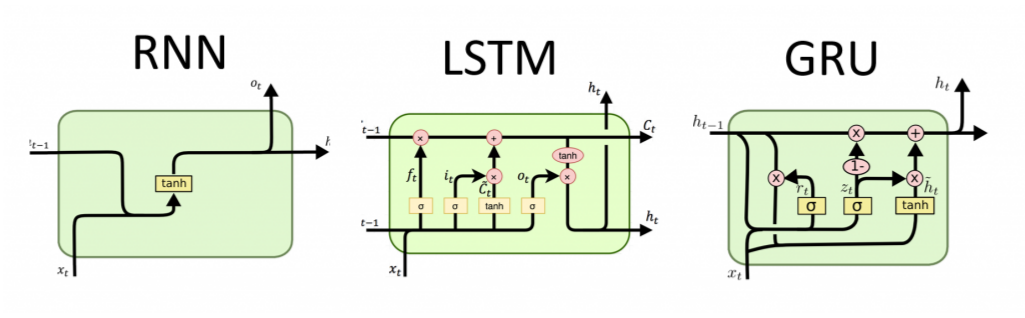


Figure 2.8: RNN-Based Architecture Node Diagrams, from dProgrammer Lopez [6]

2.3 Gradient Descent

Equally as important as the individual model architecture is the way by which the model learns. Gradient descent, combined with the backpropagation algorithm, is “by far the most common way to optimize neural networks” [17]. Once the backpropagation step is completed during a training period, the resultant derivatives can be used with gradient descent or a derived optimization technique to update the learnable parameters of the model. Effectively, backpropagation is used to find the gradient of the loss and gradient descent is then used to follow that gradient downward to minimize the loss. There are a number of optimization techniques that employ gradient descent to yield differing learning results. Updating the model parameters after every sample is known as stochastic gradient descent and generally yields very messy and more random changes to the loss curve. Batch gradient descent is updating the model parameters only once, after collecting the gradients of every training sample. This yields less random results but can be either too weak or too strong in updating the parameters. The middle ground between these approaches is mini-batch gradient descent, which updates the model parameters after a defined number of samples, which is considered a hyper-parameter during the training phase.

Chapter 3

Architecture

The majority of the work described here centers on the creation, training, and evaluation of a recurrent neural network model. The primary contributions here are the implementations of the modified RNN-based architectures which take advantage of context vectors made from word embeddings. There are four major portions of the implemented high-level architecture: the collection of valid real-world data, the creation of the model, the training of the model, and the evaluation of the trained model. Figure 3.1 gives a brief high-level view on the data flow for a single data sample during the text generation process. Since the model is focused on generating table names at the character level given context for data column names, the training data used is made of real-world examples of database table names along with their associated data columns. These examples were obtained from crawling open source repositories on GitHub. The collected data does include less desirable, messy data which required cleaning and preprocessing before being used to train the model; that step is detailed in Section 3.1. The basis for the model is a recurrent neural network being trained to generate words at the character-level based on seed text input including a context word embedding. The details of the neural network model's design are explained further in Section 3.2. After both data collection and model creation, the next major portion of the architecture is the training of the model to perform its intended task of character-level text generation; that process is described in Section 3.3. After training the model on the collected data, the model's output results were evaluated for both quantitative and qualitative performance, described in Section 4. These steps together formed the larger process for creating a recurrent neural network based model to learn sequential data through the use of word embeddings and to perform character-level text generation.

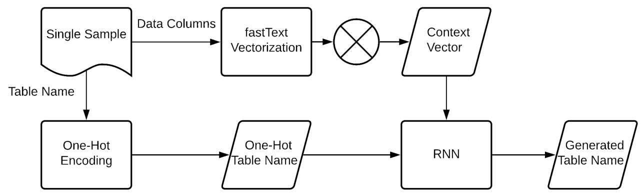


Figure 3.1: Single Sample Data Flow

3.1 Data Preprocessing & Cleaning

Due to the inclusion of real-world collected data for neural network training, the collected data needs to be properly cleaning and preprocessed before being used. Real-world data allows for the model to learn from the habits and styles of real users and developers in the world, but also contains significant amounts of dirty or useless data from leftover work or testing. The dataset, pulled from the Google BigQuery [8] Project, consists of any SQL table creation statements from scraped open-source GitHub repositories, which were then saved as table name followed by data column names with one line per table creation statement found. This data was selected as Structured Query Language (SQL) is the primary language used to create and modify relational databases. A SQL create statement, such as `CREATE TABLE Person (SSN int, LastName varchar(255), FirstName varchar(255))`, contains the database table name and the data column names, highlighted in blue, that are necessary for the training of the neural network model. The first step employed in the data pipeline for this work was to properly filter and clean this collected data set. The data was filtered on five conditions which were set to remove the majority of the data which would not help or even detract from the intended learning of the model without also removing too much of the valid data samples. This results in still having a significant amount of usable data, but does preserve some inherent dirtiness associated with human generated data. The filtering first checks for any non-alphanumeric characters and if the table name contains any numbers. Those that contain non-alphanumeric characters or numbers are removed from the data set. Then, the filtering process compares the first two characters of the table name and checks the length of the table name. If the first two characters are the same or if the table name is smaller than two characters, the sample is then removed from the data set. The final filtering condition is based on a probabilistic splitting technique from the Word Ninja module. The module is able to split a single non-spaced string into individual words based on the probability distribution as learned through the "English Wikipedia unigram frequencies" [2]. The table name is split using this module and the resultant list of words is counted for proper words in the English language using the NLTK module's English Corpus as reference. If 50% of the words are part of the English language, then the sample is kept, otherwise it is removed from the data set. This allows for some samples, such as "useraccounts", to be kept while still removing others, like "ac_ak_profiles". This filtering pipeline greatly reduces the amount of dirty or messy samples in the dataset while still allowing for plenty of real-world data to be included for the neural network model to learn from.

After filtering, the next major step in preprocessing the data for training is tokenization. Tokenization is the process by which generally a string is reduced to a list of tokens. Since the RNN model is targeting character-level generation, the tokens are the individual characters. Using the Keras tokenizer module, a tokenizer object is created and is fit to the post-filtered data set. The data set contains 42 unique characters, each of which also has a unique integer assigned to it in the tokenizer. The tokenizer will be used to convert strings into arrays of integers based on the tokens of that string. This tokenization process allows for the transformation of string input data into a list of numerical values that can be used as input during the training of the RNN model as the model itself can only take in numerical values to be incorporated into the forward propagation phase of training.

3.2 Neural Network Model Design

The neural network models being applied to generate text in conjunction with word embeddings are fairly simple architectures, shown in Figure 3.2.

The three RNN-based model types implemented feature the same architecture with the only major difference being the forward propagation step during training. The model designs feature an initial input layer which is equivalent in size to the number of unique characters in the training data as the input data must be sequences of one-hot encoded characters. For this work, the number of unique characters found in the dataset is 42 characters. The input layer then connects to a hidden layer with a variable number of nodes. There is no concrete solution for finding the optimal value for this hyper-parameter other than trial and error, though values between 128 and 256 have been found to give the best results for this particular work. Another hyper-parameter for this hidden layer is the activation function used to add non-linearity to the data. The two activation functions that proved most functional for this application are the hyperbolic tangent (tanh) and the Rectified Linear Unit (ReLU). The hidden layer then connects to the final output layer, which takes the un-normalized log probability (logits) produced by the hidden layer and normalizes them into predictions using the softmax function. The output layer is also equivalent in size to the number of unique characters in the data set. The final output of the model is then a set of softmax predictions for each of the possible characters to be generated next for the given input sequence.

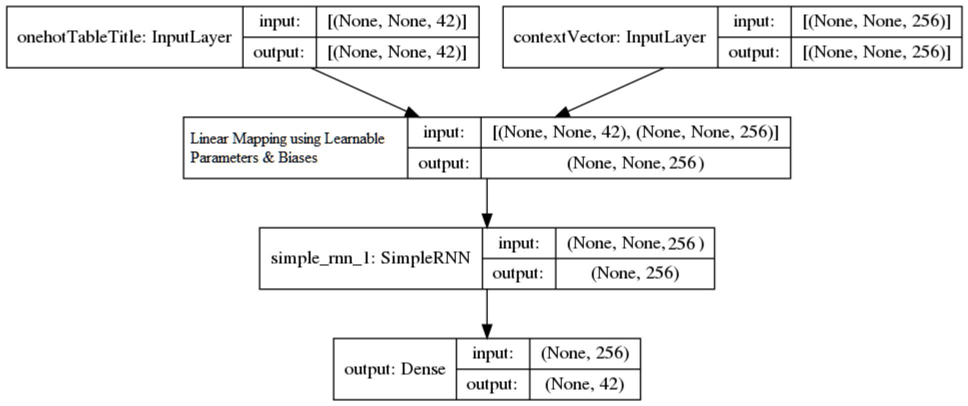

onehotTableTitle: InputLayer
input: [(None, None, 42)]
output: [(None, None, 42)]
contextVector: InputLayer
input: [(None, None, 256)]
output: [(None, None, 256)]
Linear Mapping using Learnable Parameters & Biases
input: [(None, None, 42), (None, None, 256)]
output: (None, None, 256)
simple_rnn_1: SimpleRNN
input: (None, None, 256)
output: (None, 256)
output: Dense
input: (None, 256)
output: (None, 42)

Figure 3.2: High-level architecture for RNN model with context vector

3.3 Neural Network Model Training

The training of a neural network model is a very important step which focuses on fitting the model's internal parameters to some task using some training data for it to learn from. Training usually consists of two major steps. First is the forward pass, known as forward propagation, where the input data is propagated forward through the network. Second is the back propagation, where the results of the forward pass are compared to the actual true answer for that input data, and that result is then used to update the weights or internal parameters of the model. This allows it to get closer to producing the correct answer with each iteration. The training process is generally done in terms of epochs, where each epoch is a single pass through the entire dataset. The training process used in this work consists of two datasets, one for training and one for validation. The model is first trained on the training dataset, which does include updating the weights through backpropagation, and then validated using the validation set, which does not update the weights. Validation is done to check the progress of the model as training progresses.

The training implementation consists of, for each epoch, converting every single line from both the training and validation datasets into batches to employ mini-batch gradient descent, which updates the internal model parameters after each batch, instead of using stochastic gradient descent, which updates the parameters after every single training sample. Each sample in the batch is then converted into a list of one-hot encoded vectors for the table name portion and a context vector, made up by summing the word embeddings for the data columns. The word embeddings used during the training process are generated using the fastText model from Facebook Research [3], primarily for its ability to generate embeddings for unknown words. These two pieces will serve as the training inputs during the forward propagation stage of the learning process. Because the model is based on a RNN, the list of one-hot encoded characters is treated as a sequence, where each one-hot encoded character is the input at a single time-step and the next character in the table name is the label. The context vector is directly included during the hidden state calculations, shown in Figure 2.3 for the RNN model, Figure 2.4 for the LSTM model, and Figure 3.5 for the GRU model. These equations are the result of extending the standardized base architectures described in Section 2.2 to allow for the inclusion of the context vector. After forward propagation is completed, the TensorFlow [1] gradient tape API is used to handle backpropagation through time. This allows for the calculation of gradients per sample, which are then averaged for the entire batch and the resulting gradient is then applied with the ADAM optimizer [9] to update the internal parameters of the model. With the training step concluded, the validation step then runs and evaluated the performance of the model on the much smaller validation dataset. The implemented training process employs a technique known as early stopping, which stops the training of the model if the model does not improve over some threshold of epochs by a defined metric. In this work specifically, if the model's validation loss does not improve over a set number of epochs, the model stops training.

$$h_t = \phi(W * x_t + V * h_{t-1} + M * c + b_h)$$
$$\hat{x}_{t+1} = softmax(h_t * U + b_q)$$

Figure 3.3: Modified RNN Hidden State & Output Layer Equations

$$g_t = tanh((x_t, h_{t-1}) * W_g + b_g + M * c)$$
$$i_t = \sigma((x_t, h_{t-1}) * W_i + b_i + M * c)$$
$$f_t = \sigma((x_t, h_{t-1}) * W_f + b_f + M * c)$$
$$o_t = \sigma((x_t, h_{t-1}) * W_o + b_o + M * c)$$
$$state_t = g_t * i_t + state_{t-1} * f_t$$
$$h_t = o_t * \phi(state_t)$$
$$\hat{x}_{t+1} = softmax(h_t * U + b_q)$$

Figure 3.4: Modified LSTM Hidden State & Output Layer Equations

$$
\begin{gathered}
u_t = \sigma(x_t * U_x + h_{t-1} * U_h + U_b + M * c) \\
r_t = \sigma(x_t * R_x + h_{t-1} * R_h + R_b + M * c) \\
state_t = \phi(x_t * S_x + r_t \cdot (h_{t-1} * S_h)) \\
h_t = u_t \cdot h_{t-1} + (1 - u_t) \cdot state_t \\
\hat{x}_{t+1} = softmax(h_t * U + b_q)
\end{gathered}
$$

Figure 3.5: Modified GRU Hidden State & Output Layer Equations

3.4 Neural Network Model Output

After the model has been trained to fit the input data, the model can then be applied to its original task of character-level text generation. Generally, a model should be tested and evaluated for its performance on data it has not seen before to observe how well the model generalizes on the data and ensuring the model does not suffer from issues such as over-fitting on the input data. These steps are covered in greater detail in Section 4. The generative process using the trained model still involves running input data through the forward pass step but does not involve the backpropagation step, as learning is no longer the goal for this new input data. The forward pass step results in both logits (unnormalized log probabilities for each of the possible output classes) and a softmax normalized probability distribution over each of the possible output classes. We explored several algorithms to use these results in generating qualitatively excellent database table names, based on manually observing the output and comparing to the truth table name and the data column names used. The first algorithm explore was straight-forward greedy search, which consists of simply choosing the character with the greatest probability in the softmax distribution at every generative step. Depending on the training of the model, this approach resulted in coherent but qualitatively awful results. While it was impressive that the model was able to generate complete words, the words generally were unrelated or far from the meaning of the true table names when tested on samples from the training data. The next approach was beam search, which selects k best candidates from all possible options at every generation step for every candidate already selected to some max depth or end goal. Beam search uses the cumulative log probabilities for each generated character as a way of scoring each candidate, returning the k best candidates when the algorithm reaches the end goal or max search depth. This algorithm resulted in fairly incoherent results without any qualitative benefit. Thankfully, the third algorithm tried, sampling, would have great results. The multinomial categorical distribution represented by the logits are randomly sampled, with the resulting class being the next character in the sequence being generated. Random sampling works because classes with larger log probabilities are more likely to be randomly selected during the sampling process. Sampling often also results in more qualitatively interesting results, as compared to the two other approach tested here. The sampling algorithm resulted in coherent and qualitatively pleasing results where the generated table name was much closer to the intended table name for samples of good quality data. The method used for sampling also contained an optional temperature parameter, used to influence the sampling process. A higher temperature value leads to more variety, while a lower temperature value would lead to the higher probability classes being selected more often. This parameter was not applied in the sampling process used in this thesis work, primarily since the quality and coherency of the results was sufficient for the intended goals being examined.

3.5 Model Hyper-parameter Optimization

An important part of every language model is the selection of hyper-parameters used. These hyper-parameters are essentially the settings of the neural network model outside of its design, such as number of nodes in hidden layers or the type of activation function used during forward propagation. They are selected by the developer prior to training the model in the aim of achieving a model that better fits the non-linear function or relationship targeted by the intended task for the model. The hyper-parameter selection should generally yield better results, generally measured quantitatively through some metric, such as loss, during training or evaluation.

A hyper-parameter optimization process was implemented during this work to efficiently obtain the best hyper-parameters to minimize the final evaluation loss of the model during training. The implemented process makes use of Bayesian optimization [21], which has been shown to work well to optimize neural networks. This algorithm works by "assuming the unknown function was sampled from a Gaussian process and maintains a posterior distribution for this function" [21]. The optimization algorithm contains a Bayesian statistical model, which is based on a Gaussian process. This statistical model contains the posterior probability distribution, a probability distribution that is updated when new information is acquired, that is used to fit on the results of the function being optimized and used to select future inputs to the function. Per iteration, the results of the function being optimized are used to update the posterior probability distribution in the Bayesian statistical model, which is then used to determine the inputs from the search-space used for the next iteration following the Gaussian process.

The unknown function used with the Bayesian optimization algorithm is a black box function from the perspective of the optimization algorithm where it inputs the next set of hyper-parameters from a given search space and simply receives a loss value after that unknown function's processes have completed. In this work specifically, the unknown function creates and trains a RNN-based neural network on the same dataset for each, using the hyper-parameters supplied by the optimization algorithm at that optimization step. The algorithm effectively maps the input set of hyper-parameters to their resultant loss value for each successful optimization iteration and uses this data, assumed to be based on the Gaussian distribution, to determine the set of input hyper-parameters for the next optimization iteration until the loss is successfully reduced to it optimized value or the algorithm hits the maximum number of iterations. The search space supplied to the optimization algorithm covered the learning rate of the model from 0.0001 to 0.005 in a log-uniform fashion, the batch size of each training batch from 16 samples to 128 samples, the choice of activation function between tanh and ReLU, and the dimensionality of the model and context vector from 64 up to 256 dimensions.

Chapter 4

Evaluation

4.1 Evaluation Metrics

Evaluation of a neural network is a significant step in confirming the performance of a created model. The evaluation process for the models created here include the use of collected and cleaned real-world data for proper validation and testing. Currently, the two metrics used to quantitatively evaluate the performance of the models were loss and accuracy, but these do not account for qualitative results. The loss is a measure of how far the model's predictions were to the actual truth value for that input data. There exists different loss types which are applicable for different prediction tasks, with the loss for this task being categorical cross-entropy. This loss was chosen due to the task of text generation being effectively predicting the class of the next character in a sequence which is done as a multi-class classification problem. Cross-entropy is used to measure the difference between two probability distributions, with categorical cross-entropy used to apply that concept to multiple possible output classes. The loss equation for categorical cross-entropy is shown in Figure 4.1, where i signifies an individual sample and j represents a particular class for that sample. In that same figure, y is the true values and $\hat{y}$ is the predicted values. This allows for the probability distribution of the truth values to be compared to the probability distribution of the predicted values over the defined number of classes. In addition to the loss and accuracy metrics, the evaluation process will include other forms of measuring the performance of the model, which also take into account the qualitative performance of the model. As the results of this model are for the benefit of human usefulness and readability, the qualitative performance of the model is a high priority.

$$loss_i = -\sum_{j}^{J} (y_{i,j} * log(\hat{y}_{i,j}))$$

Figure 4.1: Categorical Cross-Entropy Loss Equation

4.2 Evaluation Process

One evaluation method includes using the fuzzy Jaccard similarity to compare sub-words made from the generated table name and the true table name as a way of accounting for syntactic and semantic variations. The Jaccard similarity is a statistical measure of how similar two sets are. For sets S and T, the Jaccard similarity is achieved through "the ratio of the size of the intersection of S and T to the size of their union" [10]. Both the Jaccard equation and the adapted fuzzy Jaccard equation are shown in Figure 4.2. Given the generated table name and the true table name, the evaluation script would first split those inputs into sub-words which are then processed into synsets from the NLTK Wordnet Python library [4]. A synset is a method of grouping words that have the same meaning as well as providing methods to getting from a single word to its group of hypernyms and hyponyms. The two sets of synsets are then used to calculate a syntactic and semantic score for each set, represented by P and R in Figure 4.2, which are then used in the fuzzy Jaccard similarity equation to achieve an evaluation metric for how similar those two sets were. A higher score indicates that the two sets were more similar, syntactically and semantically, while a lower score shows that they were further apart.

$$Jaccard = (S \cap T)/(S \cup T)$$
$$fuzzyJaccard = (2 * P * R)/(P + R)$$

Figure 4.2: Fuzzy Jaccard Similarity

Chapter 5

Results and Discussion

The results attained throughout this work in exploring the use of word embeddings to provide context to a recurrent neural network in the text generation process are examined here. First, the main conceit of word embeddings are explored and shown through plotting after processing with dimensionality reduction techniques. Words with similar meanings are meant to be mapped closer in vector space, which can be confirmed through the plotting process described in Section 5.1. Similarly, context vectors for the same table name but with differing data column names should also be closer together when plotted on a 2D axis, as explored in Section 5.2. Section 5.3 covers the results of several trials testing the use, generation, and combination of word embeddings to provide context as models as trained, validated, and evaluated for the task of text generation. The results attained through that testing and this work overall are combined into a single model, which is described in Section 5.4.

5.1 Word Embedding Verification

As a major component to this work, the actual generation and resourcefulness of the word embeddings used were evaluated. This process involved generating and using a dimensionality reduction technique to plot word embeddings to ensure that they encapsulated meaningful relational data to each other at minimum in a 2D space. Using a fastText model trained specifically on the collected dataset, each table name was converted into a 256 dimensional word embeddings. Since it would be very computationally intensive and quiet impractical to generate a 256 dimensional visualization for these embeddings, a dimensionality reduction technique, in this case using T-distributed stochastic neighbor embedding [22], or t-SNE, was applied to the word embeddings to map each embedding to a 2D point for better visualization. t-SNE performs a non-linear mapping of the high-dimensional data to a lower dimension, adapted from the work on the Stoachastic neighbor embedding by Hinton and Roweis, with improvements to specifically reduce "the tendency to crowd points together in the center"[22] that other previous techniques suffered from. Since this technique's memory and computational complex increase exponentially with the number of points, the plots described in Figure 5.1 only show 1000 samples instead of the entire dataset. The first plot in Figure 5.1 shows a overview of the spread of the plotted embeddings, while the second plot is a closer look at a small cluster from the first plot. Each point is labeled with the table name used to generate the original embedding and, as visible in the second plot in Figure 5.1, words with similar meanings or semantic connections are indeed closer together in vector space than more dissimilar words. This shows that the vectorization of the table names through the use of the fastText model was able to well capture the relationship between the words in the dataset.

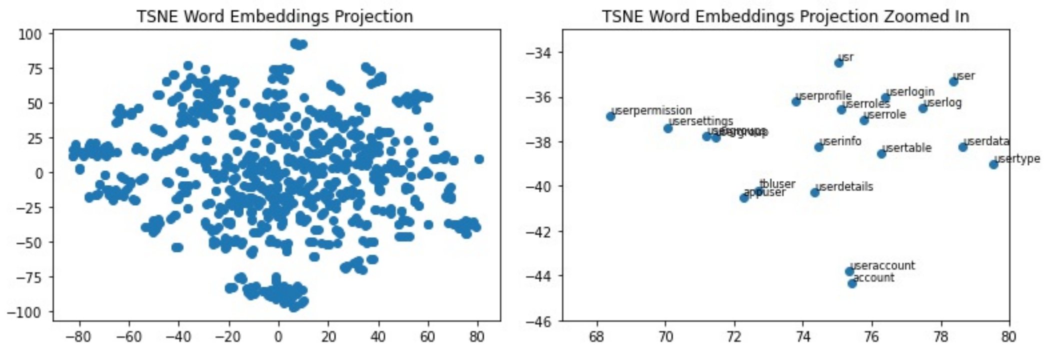


Figure 5.1: 2D Plots of Word Embeddings Using t-SNE[22]

5.2 Context Vector Visualizations

Using t-SNE, Figure 5.2 shows the context vectors for every occurrence of the top 10 most frequent tables used in the training phase of this work. While some of the more generic table names, such as ‘test’ in purple, have a large spread, the more well-defined tables are naturally more clustered together, such as ‘user’ in coral. As the associated data columns for each occurrence of these tables generally refer to the same object or meaning, the associated context vectors would be closer in vector space as well. Just as with the plotted embeddings in Section 5.1, strongly related vectors are closer in distance than other, less-related vectors. This quality formed the basis for how this inclusion benefits the neural network training process for text generation. The model is able to better learn the relation between the words used to create the context vector and able to better differentiate between what table names to generate based on the context vector values.

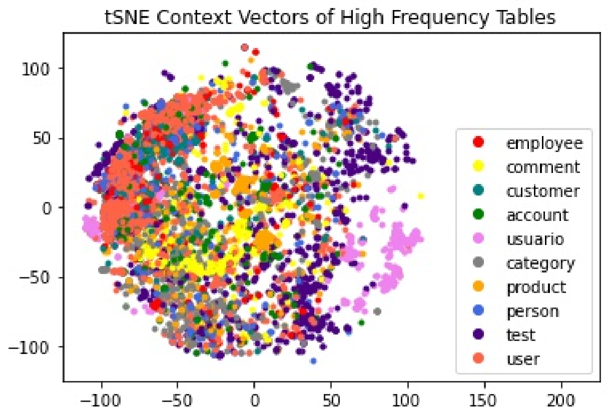


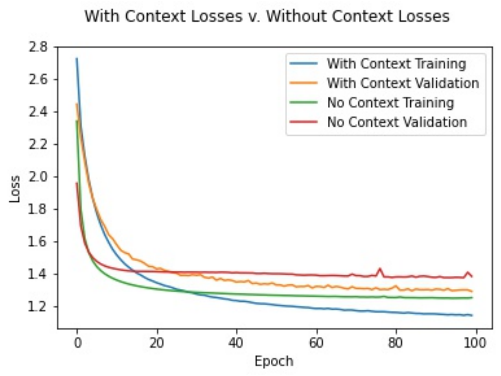
Figure 5.2: 2D plot of context vectors of the highest frequency tables using t-SNE [22]

5.3 Model Results

Through the use of vector space embeddings, a recurrent neural network model capable of character-level text generation has been successfully created. Each of the models used throughout the trials were created newly at the start of each trial and then trained and evaluated using the same process, as described in Section 3.3. Each of the models were trained on a Ubuntu Linux-based system, leveraging NVIDIA Tesla P4 GPUs with Google's TensorFlow platform [1] to speed up and optimize the training process. The models examined here also used the same set of hyper-parameters: 100 epochs of training and validation, a learning rate of 0.001, a batch size of 64, and the *tanh* activation function. The inclusion of vector space embeddings as a context vector has allowed the model to better learn the difference between input data samples and the relationship between individual words as characterized through the vectorization process used. If there was no context vector included, since each table generation begins with the same start token, there is no information available for the model to generate text closer to what the desired results are. Instead of relying on the model to decide what should be generated, the context vector gives the model significantly more information to rely on for what should be generated compared to having none for the context-less generation process.

Shown below are sample results which offer a look at a few of the various model examined throughout this work in the process of creating a model well suited towards the specific target task of generating database table names from data column names. Figures 5.3 - 5.5 show the training curves and output results of several models examined throughout this work. Each set focuses on comparing some of the different ways to incorporate the context vector into the model training and text generation process. Each of the trial results were averaged over three trials for each of the models being tested. Along with the trials described below, additional model improvements were separately attempted, such as jointly tuning the embeddings during model training, but did not yield useful results.

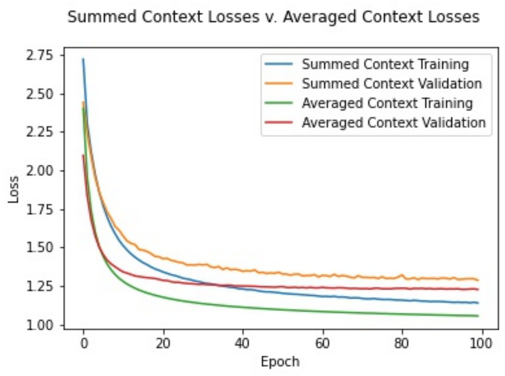
Figure 5.3 acts as a baseline by showing the effect of adding the context vector to the model alongside a model without a context vector at all. The figure shows how the context-less trials achieved lower loss values quicker than the trials using context vectors but also how those trials without context vectors plateaued. The trials with context vectors did eventually achieve lower loss in both training and validation. The trials with context vectors are able to more often generate the true table name compared to the context-less trials, which while able to generate coherent words often, generate words pseudo-randomly without any connection to the intended meaning or purpose of the database columns. As shown in the table next to the plot, the trials with the context vector produces coherent results that are generally in the same vein as the true table. The trials without the context vector often produces near coherent words, but the resulting table name is not semantically close to the true table name. Using the evaluation method described in Section 4.2, both sets of trials were evaluated with the trials using the context vector averaging an evaluation score of 14.4% and the context-less trials averaging an evaluation score of 11.4%. While the titles generated by the context-less model are not as consistently coherent as the other model, the output it produces in being a standard RNN are still of decent quality.



Model Outputs		
Truth	Context	No Context
user	user	mue
project	asset	customer
account	user	event
trophy	meta	torg
cart	map	persones
lane	file	ball

Figure 5.3: Plot comparing the training and validation losses of a model with the context vector and a model without the context vector

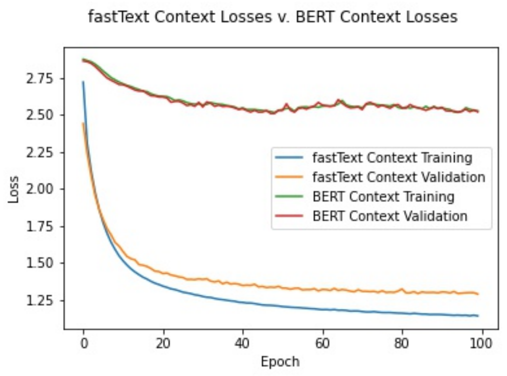
Figure 5.4 compares two models that both take advantage of context vectors, but generate those context vectors in differing ways. The first set of trials, shown as the Summed Model, incorporates a context vector made by summing up the word embeddings of the data columns for each training sample. The second set of trials, shown as the Averaged Model, uses a context vector that is generated by averaging the word embeddings for the data columns for each training sample. The averaged model does have consistently lower loss values per epoch, but both models perform about the same qualitatively. The table of results show that both models generate table names that are both coherent and semantically close to the intended meaning of the true table name, usually. As in the second to bottom result, both models were close to the true table name but in differing ways. The Summed Model produced map, presumably linking the associated context vector to cartography, while the Averaged Model produced a word that was only one letter away from matching the true table name. The Summed Model trials achieved an averaged score of 14.4% during the evaluation process, while the Averaged Model trials achieved an averaged score of 15.3%. Both models produced qualitatively fine results and given more training time, could produce even better results.



Model Outputs		
Truth	Summed	Averaged
user	user	user
project	asset	project
account	user	user
trophy	meta	tag
cart	map	card
lane	file	media

Figure 5.4: Training and validation losses of models with summed and averaged word embeddings as a context vector

Figure 5.5 compares two sets of trials that both use the summing method for generating context vectors but use differing word vectorization sources. One set uses fastText [3] to generate word embeddings for the data columns, while the other set uses Google's BERT [5]. Both word embedding generation models support out-of-vocabulary words as they both use sub-word representations to better map words into high-dimensional vector space. While fastText primarily uses n-grams as it's sub-word representation during learning, BERT uses a transformer encoder system to encode and then learn the sub-words for the input training data. The fastText model used in the fastText trials was trained using the pre-processed and cleaned dataset, while a pre-trained BERT model with 8 Transformer layers was used for the BERT trials. Each data column was individually converted into a word embedding using each of the models to synchronize the context vector combination method between the trials. In the BERT model, the generated word embeddings were context-free embeddings grabbed from the model's final hidden layer, since the input sequence to the model was just the single word instead of an entire sentence. The BERT model also featured positional encodings, which are primarily used for the model to understand word order in an input sentence, added to the generated word embeddings. As the input to the model were individual words, this positional encoding was not removed. Both models trained in this trial used context vectors created by summing up the generated word embeddings for each of the data columns in a sample. As shown in the results, the trials using BERT word embeddings, surprisingly, performed significantly worse than the fastText approach. The BERT trials had consistently higher loss values per epoch during training and validation and produced very incoherent results during generation and evaluation. The primary theory behind this disparity is likely the difference in how the two vector encoding models were trained. fastText was likely able to generate better word embeddings due to its more limited and focused corpus, whereas the BERT model was pre-trained using significantly larger amounts of training data across several languages.



Model Outputs		
Truth	fastText	BERT
user	user	s
project	asset	os
account	user	ouh
trophy	meta	bstebn
cart	map	crot
lane	file	gagtcrued

Figure 5.5: Training and validation losses of models using fastText and BERT word embeddings

5.4 Final Model

After combining the results of the hyper-parameter optimization process and the testing examined for Section 5.3, a final model was produced as the exemplification of the work performed here. This model used the base RNN architecture which consists of a dimensionality of 256, 64 samples per batch, a learning rate of 0.001, used the ReLU activation function, and leveraged context vectors made by averaging the data columns associated with the sample table name. This model was trained over 300 epochs, instead of just the 100 epochs used through the other model testing. As shown in Figure 5.6, the losses are similar to the ones seen in Figure 5.4 for the Averaged Model trials. Even past the 100 epoch point, the training loss continues to decrease, but the validation loss remains fairly constant. The model performs slightly better than both of those models, achieving an evaluation score of 16% on the test dataset using the evaluation process discussed in Section 4.2. Figure 5.7

serves to highlight that the majority of the generated tables are coherent with the quality of the title generated depending on the quality of the input sample. The quality of the generated title depends significantly on the quality of the data columns. Samples with messier or overly generic columns, such as the ‘cache’ or ‘ban’ samples, will still produce near coherent results associated with the provided data columns. These generations may even be more useful than the original table title associated with the used context vector, such as the ‘mytable’ sample in Figure 5.7. Samples with clear and well-defined columns will generally produce qualitatively good results.

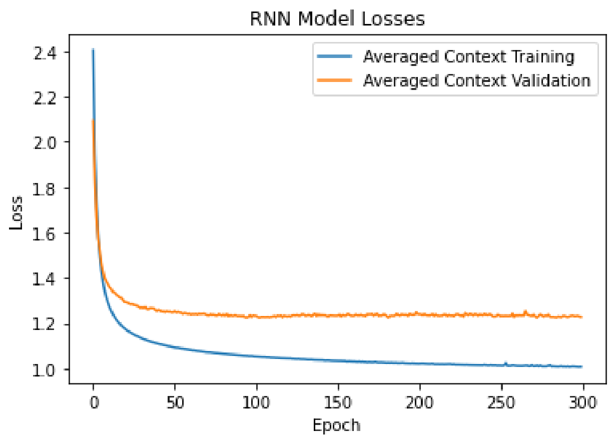


Figure 5.6: Training and validation losses of the RNN model

Model Outputs		
Truth	Generated	Columns
user	user	id_user, login, pwd, name, comments
ban	post	id, ip, time
catalogue	series	catalogueid, loguserid, source, isbn, bookname, ...
edge	branch	edge_id, child_node_id, parent_node_id
mytable	employee	id, lname, fname, age, gender
cache	pasentest	f1, f2, t, url, data

Figure 5.7: Table Results with data columns of RNN model

Chapter 6

Related & Future Work

As the need for better Natural Language Processing (NLP) solutions is needed in an exponentially expanding field, the need for better and more useful word vectorization also increases. Vector space embeddings are already a major possible avenue for developers to leverage in allowing machine learning and neural network models to closer understand words or at bare minimum, the relationship between words. While the work on already done on and with these techniques are great, there is still plenty that can be improved. Section 2.1 examined several different word vectorization methods and there still exists plenty of alternative options targeted towards other NLP tasks.

A significant related work that inspired the work performed here is the Column2vec project [15]. Column2vec described two possible models for generating table names based on database column names. The project proposed two separate model architectures which also leveraged the use of word embeddings. One possible avenue suggested in that work for improving the generative abilities of the model would be to also incorporate statistical data from the database columns into the training process, such as calculating and applying the mean or standard deviation of numerical columns in addition to the context vector. This, however, would require a larger training corpus in terms of size, as accurate additional metadata would be required in addition to the existing table name and column names. The additional metadata would need to be calculated already and included in the dataset, or the database data would need to be included to calculate the relevant metadata to be used in the training phase.

Other attempts at incorporating context into the text generation process have been done. Santhanam explores applying context in text generation, highlighting the issue that “without any such context, there is no semantic consistency among the generated sentences” [19]. Santhanam’s work explores two processes for the application of context. The two primary methods explored in that work are word importance and word clustering. Word importance picks the word in a training sample with the highest overall frequency and simply one-hot encodes it. Simply one-hot encoding a word doesn’t too much to preserve the semantic relationship between the words in the sample, but does give some additional context across several samples. The word clustering approach is similar to the summed context vector explored in this work, with their vectorization source being the Word2vec. The major difference is that instead of using the summed context vector as the context source during training, the distributed representations in vector space are clustered and the vectors in the cluster center of the summed vector are combined into a context vector. This allows similar sentences to have the same context vector, which increases the separation between sentences belonging to other clusters, but removes all differing context for sentences in the same vector space cluster.

Additional future work that can be explored using this work as a basis include research and exploration into alternative neural network models. A different sequential model, such as a encoder-decoder Seq2Seq style model could be used. In the Seq2Seq architecture, a separate model layer could be devoted to learning the context vectors, with the result of that layer combining with the input table name learning layer to produce the desired output. This approach would also greatly increase the computational complexity of the model and lead to increased training times. There has also been some promise discovered recently in the use of Generative Adversarial Network (GAN) models for text generation tasks apart from their already significant promise in image generation tasks. Additionally, the use of the BERT vector space encoding model could be re-examined to change the generation of the word embeddings, such as using context-based embeddings instead of the context-free ones used.

Chapter 7

Conclusion

Through the lens of database normalization, this work has explored the use of word embeddings to offer additional contextual and relational information during the text generation process for creating a new table name after a database has been normalized. Currently, normalization of database tables results in new tables with data column names carried over from the source table, but no table name. Meaningful table names are vitally important for ensuring the overall schema is human readable and useful towards other tasks. This thesis examines the use of word embeddings, generated from the data column names, to offer additional semantic information during the neural network learning process to improve the text generation performance of the model.

The primary technologies used throughout this work consist of material from the artificial intelligence field, the data analysis field, and natural language processing field. A large portion of the work stemmed from the creation and training of a recurrent neural network based model to achieve character-level text generation. The most straightforward choice for model architecture was a recurrent neural network-based model, either a simple RNN or a long short term memory (LSTM) model. These were chosen primarily for their ability to learn over sequences, exhibiting significant dynamic temporal behavior, which is essential for natural language processing tasks like text generation. The training phase included the collection, cleaning, processing of real-world table creation data from public open source GitHub repositories. This data is filtered and tokenized to remove the majority of the dirty data that would otherwise detract from the model's learning. The training phase also included the processing of the training data through the forward propagation, backpropagation, and gradient descent steps to allow for model to fit on the training data. This portion of the training process sought to incorporate the use of word embeddings as a context vector for each training sample to allow for the relationships and semantics between words to be learned by the model.

After the training of the model was completed, the equally important process of evaluating the model and its results was performed. In addition to using the loss and accuracy of the model over a test set of samples, a more qualitative approach was also done. A script applying the fuzzy Jaccard similarity between the synsets of the true table name and the generated table name was used to give a score to the performance of the model, taking into account the readability and human meaningfulness of the results, in addition to the numeric results from the loss and accuracy. A higher score showed that the two table names were close, both semantically and syntactically, while a lower score showed that they were not as similar or related. This evaluation process was applied to the models discussed in Section 5 to offer a better comparison metric between the models that accounted for the human usefulness of the generated labels instead of simply relying on the quantitative metrics calculated during training and validation. There, it was shown that models that did take advantage of context vectors lead to significantly better text generation results.

Through this work, a neural network model was successfully created that was capable of character-level text generation and incorporated vector space representations of the data column titles into the learning process. Compared to language generation models that did not include some form of context, a model that did produced qualitatively better results during database table name generation. By integrating the database column titles into the learning process, the created language models were able to generate higher quality and more human useful database table titles.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[2] Derek Anderson and Scott Randal. Word ninja. `https://github.com/keredson/wordninja`.

[3] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information, 2017.

[4] NLTK developers. Source code for nltk.corpus.reader.wordnet. `https://www.nltk.org/\_modules/nltk/corpus/reader/wordnet.html`.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[6] dProgrammer Lopez. Rnn, lstm gru. `http://dprogrammer.org/rnn-lstm-gru`, 2019.

[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[8] Felipe Hoffa. Github on bigquery: Analyze all the open source code, Jun 2016.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[10] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, USA, 2nd edition, 2014.

[11] Leland McInnes. Uniform manifold approximation and projection. `https://github.com/lmcinnes/umap`.

[12] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[14] Tomas Mikolov, Wen tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In Lucy Vanderwende, Hal Daumé III, and Katrin Kirchhoff, editors, *HLT-NAACL*, pages 746–751. The Association for Computational Linguistics, 2013.

[15] Michael J. Mior and Alexander G. Ororbia II. Column2vec: Structural understanding via distributed representations of database schemas, 2019.

[16] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[17] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016.

[18] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986.

[19] Sivasurya Santhanam. Context based text-generation using lstm networks, 2020.

[20] Leonid Sigal. Topics in ai: Rnns (part 2). University Lecture, 2020.

[21] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing*

Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States, pages 2960–2968, 2012.

[22] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008.

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[24] P. Werbos. Backpropagation through time: what does it do and how to do it. In *Proceedings of the IEEE*, volume 78, pages 1550–1560, 1990.

Vita

Goldy Malhotra was born in New Delhi, India on January 23, 1998, the son of Ramanjeet Singh Malhotra and Ramit Kaur Malhotra. He received the Bachelor of Science degree in Computer Engineering Technology from Rochester Institute of Technology, Rochester, New York in 2021. He is currently pursuing his Master of Science degree in Computer Science from Rochester Institute of Technology, Rochester, New York. His research interest includes Machine Learning, Artificial Intelligence, and Data Science and Analytics. His current research includes combining word embedding context vectors with recurrent neural networks to improve text generation for database schema management.

Permanent address: 455 Richard Way
North Plainfield, New Jersey 07062

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.