Language Modeling on Penn Treebank

Shashwat Pandey
University of California, Santa Cruz / Santa Cruz, CA
spandey7@ucsc.edu

1 Introduction

The purpose of this assignment is to train a neural network model over Penn Treebank corpus and come up with a language model. The language model is a word sequence prediction model that has been trained to anticipate the following word in light of the previous input sentence. Because we are not establishing distinct output labels, this type of work is a self-supervised learning task. Instead, we're using the texts as inputs and outputs in a precise method that will aid the model in comprehending the foundations and grammatical structure of the dataset's language (or the language used in the dataset). Such a model can be used to carry out sequence generation as well. In other words, it can forecast which token, out of all those that could happen, is most likely to follow, given any initial collection of tokens. This means that by having the model repeatedly produce the next word given the previous words so far, we can give the model one word and have it generate a whole series. Applications of these tools include auto-completion. They can learn to help us write emails, code, Shakespeare books, just whatever we train them on.

To begin with, we'll be using the CRoss Industry Standard Process for Data Mining (CRISP-DM) pipeline for our analysis and modelling, which is the defacto standard and an industry-independent process model for applying machine learning projects.(1). Figure 1 provides an exact idea of the flow supposed to be followed before tackling any data-science/machine learning based problem and its entire lifecycle. The following section talks about the analysis on the dataset.

1.1 Dataset

The dataset which we are using for the task is the Penn Treebank Project: Release 2 CDROM, featuring a million words of 1989 Wall Street Journal

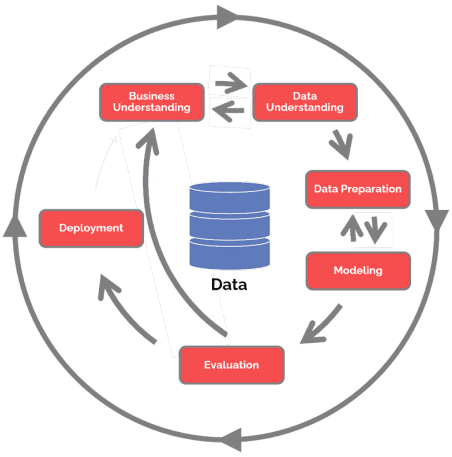

Figure 1: CRISP-DM Diagram: Inspired by Wikipedia

material. This corpus has been annotated for part-of-speech (POS) information. In addition, over half of it has been annotated for skeletal syntactic structure.(2) The data has been already split into train, test and validation sets having 42068, 3761, and 3370 sentences respectively. To get a better understanding of the data, we visualised the word count of all the sequences in the training dataset in a graph, as seen in Figure 2.

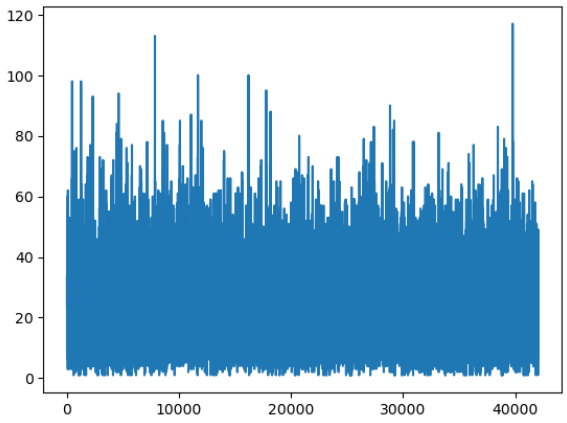

Figure 2: Word Count For Training

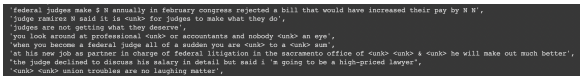

Figure 3: Training set example sentences

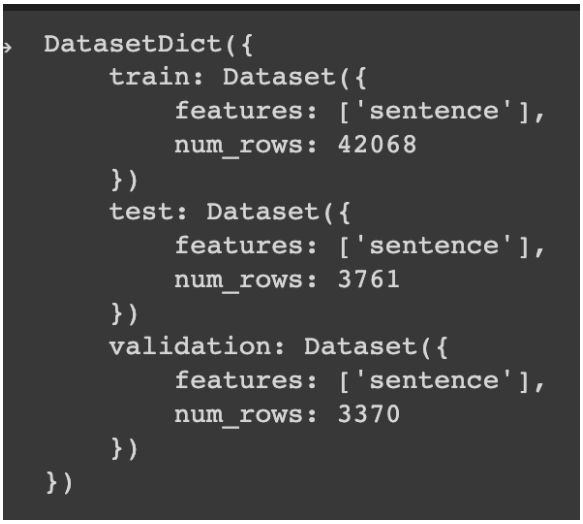

Figure 4: Penn Treebank from huggingface

Examples of entries in train set can be seen in Figure 3, along with the structure of the dataset being shown in Figure 4.

2 Methodology

This section explains how to get word tokens from sentences and the approach used for the task of designing and training a language model.

2.1 Text Preprocessing and Tokenization

Every language model has a lexicon from which it takes its tokens.(3) If the vocabulary for English word-level language modeling contains 100K words, the model must decide which one (out of the 100K) to forecast as the next word given any starting collection of words.

Since the dataset provided to us already had <unk> tokens for unknown words, we can bypass the padding process. However, to let the model know beginning and end of sequences, we will add <start> and <eos> tokens in our vocabulary. After tokenizing and creating vocabulary of all the words including <start> and <eos>, we came up with a lexicon of 10001 words.

For tokenization and vocabulary, split method was used and the vocabulary used was also limited to the provided dataset. We don't use any pretrained embeddings or tokenization functions such as spacy, nltk, torchtext etc.

2.2 Model Implementation

Language models (LMs) assign a probability to any given sentence of n words W = (w1, w2, . . . , wn). According to Bayesian rule, it can be decomposed as the product of the individual word probabilities given its word history(4): $P(\mathbf{W}) = P(\mathbf{w_1}, \mathbf{w_n}) = \prod_{t=1}^{n} P(\mathbf{w_t} \mid \mathbf{w_{t-1}}, \ldots, \mathbf{w_1})$

In recurrent neural network language models (RNNLMs), the word probability can be written as: $P(\mathbf{w}_t \mid \mathbf{w}_{t-1}, \ldots, \mathbf{w_1}) \approx P(\mathbf{w}_t \mid \mathbf{w}_{t-1}, \mathbf{h_{t-2}}) = P(\mathbf{w}_t \mid \mathbf{h_{t-1}})$

where $h_{t-1} \in R^M$ is the hidden vector that represents the previous history $(w_{t-1}, \ldots, \mathbf{w}_1)$. The RNNLM can be generally divided into three parts: the projection layer, the recurrent layer and the output layer. The projection layer projects the vector $w_t \in R^N$ into a continuous space $\chi \subseteq R^M$ as x_t, where N is vocabulary size and usually $M \ll N$. The projection layer is followed by the recurrent layer, which calculates the hidden vector by repeatedly applying a gating function: $\mathbf{h_{t-1}} = g(\mathbf{x_{t-1}}, \mathbf{h_{t-2}})$

which often relies on sigmoid activations in conventional RNNLMs Long short-term memory (LSTM) RNNLMs can be utilized to solve the vanishing gradient problem associated with standard RNNLM training. The output layer applies a softmax activation on the hidden vector $\mathbf{h_{t-1}}$ to calculate the word probabilities for both traditional RNNLMs and LSTM-RNNLMs:

$$P(\mathbf{w}_t \mid \mathbf{h}_{t-1}) = \frac{\exp\left((\mathbf{V}h_{t-1})^{indx(\mathbf{w_t})}\right)}{\sum_{n=1}^{N} \exp\left((\mathbf{V}h_{t-1})^n\right)} \quad (1)$$

where $(\mathrm{V}h_{t-1})^n$ signifies the n-th elements of $(\mathrm{V}h_{t-1})$ and indx (w_t) is the index of the word w_{t°, $\mathbf{V}$ is the projection matrix of the output layer that projects the hidden vector back to vocabulary space.

For this assignment, we considered 2 RNN based models to train and test, each of which have been described below.

1. *Baseline RNN*(5): An artificial neural network containing internal loops is known as a recurrent neural network (RNN). These internal loops cause the networks to have recursive dynamics, which introduces delayed activation interdependence among the network's processing elements (PEs). A second type of distributed representation is feasible in recurrent

networks, but most neural networks use distributed representations, where information is encoded across the activation levels of numerous PEs. Information can also be represented in RNNs by changing the activation of one or more PEs over time. These networks are also known as spatio-temporal networks since information can be encoded geographically, spanning several PEs, and temporally.

2. *Long Short Term Memory (LSTM)*(5):Long Short Term Memory Network is an advanced RNN, a sequential network, that allows information to persist. It is capable of handling the vanishing gradient problem faced by RNN. A recurrent neural network is also known as RNN is used for persistent memory.

3 Experiment

3.1 Train-Test Split

Since the data provided was already split into training, validation and test set, we didn't have to manually perform the task to split the data into three parts. As mentioned in the dataset subsection under introduction, we have 42068 entries for training, 3761 for validation and 3370 for testing.

3.2 Training

3.2.1 Baseline RNN

We use the tokenization techniques mentioned in subsection 2.1 followed by which we move to building our model. Our recurrent neural networks receive vectors rather than strings as input. As a result, we develop an index to word and word to index mapping between words and indices. Keeping in mind that our objective is to anticipate the following word, y is simply the x vector with the last element being the <eos> token. We then create a basic 2 layered model with an embedding layer and RNN layer as our baseline. We also use sequence length this time to make sure every iteration runs unifromly through the data and use getbatch function to move words across in order to maintain the sequence length. Following this, we use Cross Entropy and Adam as our loss and optimization function respectively. As a form of regularization, we will use a dropout layer before each of the embedding, RNN, and output layers. Tuneable hyperparameters are sequence length, hidden dimensions, embedding dimensions, learning rate, dropout, batch size, epochs.

3.2.2 LSTM

The model we'll create will match Figure 5. An embedding layer, LSTM layers, and a classification layer make up the three essential parts. We are already aware of the LSTM and classification layers' functions. Each word (provided as an index) is mapped into an E-dimensional vector by the embedding layer so that subsequent layers can learn from it. Because they presume there are no relationships among words, indecies, or one-hot vectors, are regarded as inadequate representations. Additionally taught during training is this mapping. We will apply a dropout layer before the embedding, LSTM, and output layers as a type of regularization. There

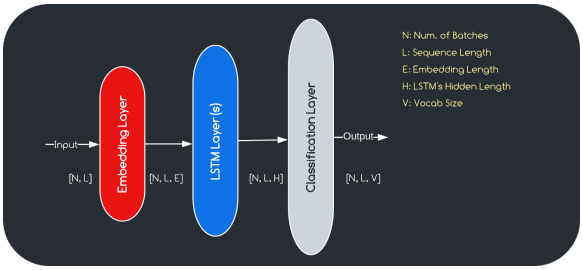

Figure 5: LSTM Architecture used

are a few things tweaked in the implementation:

1. *tie-weights(6):* Making the embedding layer share weights with the output layer is the idea here. Because it has been demonstrated that the output weights also learn word embeddings in a certain sense, this reduces the number of parameters. Keep in mind that the hidden and embedded layers must be the same size for this to operate.

2. *self.init-weights():* The weights will be initialized as in this paper. They advise equally initializing all other layers in the range [-1/sqrt(H), 1/sqrt(H)] and the embedding weights in the range [-0.1, 0.1]. We must go over each of the LSTM's layers in order to initialize the hidden to hidden and hidden to next layer weights.

We also use *detach-hidden()* function to let PyTorch know that our hidden states are independent as each of them have different sequence lengths. For optimizer, we use Adam and CrossEntropy as our loss function. We also calculated the number of trainable parameters which came out to be 27,044,625. Tuneable hyperparamters were dropout rate, batch size, number of layers, hidden

Model	Parameter	Range
RNN	LR	e-4-0.1
RNN	Seq Len.	20-60
RNN	Hid Dim	300 - 2000
RNN	Emb dim	300 - 2000
RNN	n-layer	1-4
RNN	Dropout	0.10 - 0.65
RNN	Epochs	2-10
RNN	BS	64-2048
LSTM	LR	e-4-0.1
LSTM	Seq Len.	20-80
LSTM	n-layer	1-4
LSTM	tie-weight	0/1
LSTM	Emb Dim	300-2000
LSTM	Hid Dim	300-2000
LSTM	Dropout	0.10-0.70
LSTM	Epochs	10 - 100
LSTM	BS	32-2048

Table 1: All tuneable hyperparameters

dimensions, embedding dimensions, learning rate, epochs. Since we were using tie-weights function, the embedding and hidden dimensions were kept same.

3.3 Hyperparameter Tuning

The range used for hyperparameter tuning are given in Table 1 for each model.

After trying all the possible permutation and combinations, we were able to identify the combination providing us with the best possible training, validation and test accuracy. Table 2 gives us the exact figures and parameters used for each model. The model and hyperparameters that gave us the best perplexity out of all can be identified in Table 3. Of course, many more iterations of training were run, but for the sake of the space all couldn't be included here.

3.4 Evaluation

Language model evaluation metrics include perplexity. What makes us want to utilize it, though? Why can't we just focus on our final system's accuracy and loss for the task that matters to us?

In fact, there are two methods we might employ to assess and contrast language models:

1. *Extrinsic Assessment* This entails testing the models' performance by applying them to a real-world task (like machine translation) and analyzing the results in terms of accuracy and loss. This is the ideal choice because it allows us to actually examine how various models affect the task that interests us. However, because it necessitates training a complete system, it can be computationally expensive and slow.(7)

2. *Intrinsic Assessment* Finding a metric to assess the language model without taking into account the particular tasks it will be utilized for is required. Although intrinsic evaluation is not a final criteria that is as "excellent" as extrinsic evaluation, it is a valuable tool for swiftly comparing models. Perplexity is an intrinsic evaluation method.(7)

MODEL	PARAMETERS
RNN	Learning Rate: e-4, Sequence Length: 50, Hidden Dim.: 1024, Embedding Dim.: 1024, n-layers: 2, dropout:0.5, Batch Size: 128, Epochs: 5
LSTM	Learning Rate: e-3, Sequence Length: 50, Hidden Dim.: 1024, Embedding Dim, 1024, n-layer: 2, dropout: 0.5, Batch Size:128, Epochs: 50, tie-weights: 1

Table 2: Model parameters with best Perplexity

We saw in subsection 2.2 how we judge a language model's performance, which is by obtaining per word probability upon historic words. The perplexity PP of a language model $p_M($ next word $w \mid$ history $h)$ on a test set $T = \{w_1, \ldots, w_t\}$ is just (8):

$$PP = e^l \tag{2}$$

$$l = 1/N \; log \; P(w_1, ..., w_n) \tag{3}$$

or the inverse of the (geometric) average probability assigned to each word in the test set by the model. As its logarithm is an upper bound on the amount of bits per word anticipated in compressing (in-domain) text using the measured model, perplexity is conceptually accurate.

MODEL	Train PPL	Val PPL	Test PPL
RNN	88.43	124.98	128.33
LSTM	**44.78**	**94.23**	**87.41**

Table 3: Best results using configurations in Table 2

4 Results

After running multiple training and validation iterations, we came to the conclusion that the LSTM model was able to train better and give a way lower perplexity than RNN. Figure 6 and 7 give us the visual representation of loss and perplexity in the language model. The RNN can preserve information over numerous timesteps more easily thanks to the LSTM architecture. The information in the cell is kept forever, for instance, if the forget gate is configured to remember everything on every timestep. Contrarily, learning a recurrent weight matrix Wh that maintains information in a hidden state is more difficult for a vanilla RNN. Although LSTM does not ensure the absence of disappearing or bursting gradients, it does make it simpler for the model to learn long-distance dependencies.

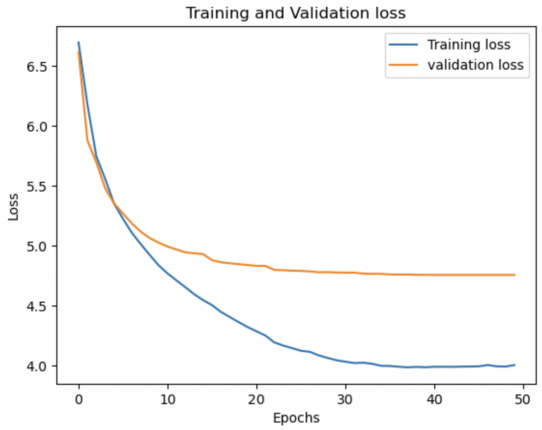

Figure 6: Training vs Validation Loss

LSTMs are successful in the real world. LSTMs reported obtaining state-of-the-art outcomes between 2013 and 2015. Handwriting recognition, speech recognition, machine translation, parsing, and image captioning are examples of successful tasks. LSTM took over as the preferred method. For some tasks, different approaches (like Transformers) have become more prevalent as of 2019, nevertheless. For instance, the summary report for WMT (a MT conference and competition) in 2016 contained "RNN" 44 times, but the report for WMT in 2018 contained "RNN" 9 times and "Transformer" 63 times.

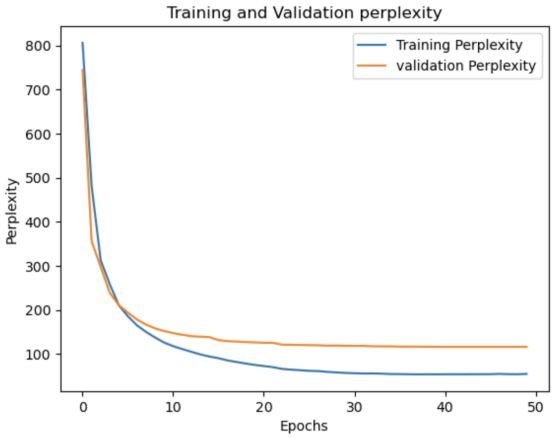

Figure 7: Training vs Validation Perplexity

Using an extremely basic tokenization technique and embeddings, having to run a lightweight LSTM model, these were the reasons our perplexity could barely reach to the double digit bar. Using torchtext tokenizers, pretrained embeddings and using a deeper model could've made a gargantuan leap in our perplexity.

References

[1] Schröer, C., Kruse, F., Gómez, J. M. (2021). A systematic literature review on applying CRISP-DM process model.

[2] Taylor, A., Marcus, M., Santorini, B. (2003). The Penn treebank: an overview. Treebanks, 5-22.

[3] Brown, P. F., Cocke, J., Della Pietra, S. A., Della Pietra, V. J., Jelinek, F., Lafferty, J., ... Roossin, P. S. (1990). A statistical approach to machine translation. Computational linguistics, 16(2), 79-85.

[4] Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., Khudanpur, S. (2010, September). Recurrent neural network based language model. In Interspeech (Vol. 2, No. 3, pp. 1045-1048).

[5] Sherstinsky, A. (2020). Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. Physica D: Nonlinear Phenomena, 404, 132306.

[6] Opsahl, T., Agneessens, F., Skvoretz, J. (2010). Node centrality in weighted networks: Generalizing degree and shortest paths. Social networks, 32(3), 245-251.

[7] Ponte, J. M., Croft, W. B. (2017, August). A language modeling approach to information retrieval. In ACM SIGIR Forum (Vol. 51, No. 2, pp. 202-208). New York, NY, USA: ACM.

[8] Chen, S. F., Beeferman, D., Rosenfeld, R. (1998). Evaluation metrics for language models.