RNN Language Model Using Vanilla LSTM With Penn Treebank

Hanusha Sabbavarapu (Matriculation Number:225018)
University of Trento
Via Sommarive, 9, 38123 Povo,Trento TN
hanusha.sabbavarapu@studenti.unitn.it

Abstract:
The work focuses on Training RNN using state of the art architectures Vanilla LSTM for language modelling with The Penn Tree Bank (PTB) dataset. It also shows the performance obtained and the results on the 2- and 3-Layers LSTM model used for training.

1. Introduction:

One of the first and most important of any NLP task is to analyse a given phrase to understand its underlying meaning. Here I will be taking a PTB dataset which consists of sentences from newspaper articles to train a Vanilla LSTM model.

Language modelling

Language models are a semi-supervised approach where a model receives a partial sequence of characters (in character-level language models) or words (in word-level language models) as input and is expected to predict the next character/word token in the sequence.

Language modelling is useful to illustrate RNN use as it is a straightforward task that has well understood non-neural implementations, but yet is extremely important in a range of real-world applications and is now commonly implemented with a neural architecture.

A language model attempts to learn the properties of a language such that when given for example n tokens, the model can accurately predict the $n+1^{th}$ token. For example, given the single token "Merry" the following word is highly likely to be "Christmas". Here just a 1-word context window is enough to help make a good prediction. Normally some more context is required. Given the string "turned the", we would have many reasonable candidates for the next word, but likely none that dominate. However, given some more context, such as "she jumped into the car and turned the", a language model might reasonably expect the word "keys" or "key" to be predicted with relatively high probability, while words like "wheel" "engine", or "ignition" might also be given a relatively high probability.

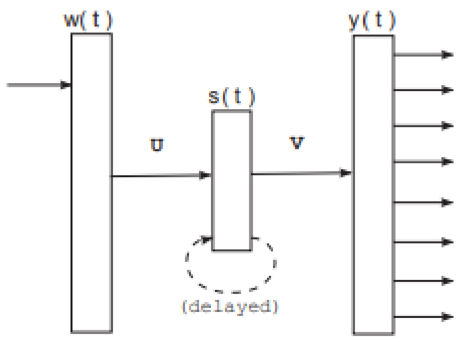
Linguistic context clues can of course be much further back in the text. Consider the example: "He sat with the ring in his pocket, perspiring, anxious, waiting for the perfect moment that would never come. Finally, he slid his hand from his pocket, and asked Marie if she would". Perhaps "marry" would be a suitable next word given this longer context. But looking back only two, three, or even four words would make this unlikely.

Until recently a language model was built statistically using n gram methods. An algorithm (the estimator) would parse a large body of text and build up tables of all 1-grams, 2-grams, 3-grams and perhaps 4-grams and 5-grams, and use some relatively straightforward probability calculations to give probabilities for individual words given a context.

This statistical form of language modelling is an important tool in Natural Language Processing tasks. Python NLTK (natural language toolkit) has a simple implementation of an NGram Language Models that you could use to estimate your own language models from texts. Let's use it to parse some simple texts and then use it to make predictions.

Recurrent Neural Networks are one of the most popular Neural Network architecture types due to their importance in processing sequential and time series data. While time-series data usually refers to real valued data with a temporal dimension such as stock market prices and sensor outputs, sequential data is a much broader data type which also includes audio, text, and video.

We begin with an overview of basic RNNs with a simple example. Over the next few weeks we will build on this with more coverage of language modelling and then a look at the more sophisticated topics of LSTM and transformer models for sequence processing after the Easter break.
In this exercise we will train a word-level language model on the PTB dataset using RNNs.
I will be training LSTM with 2 layers and also experimenting with 3 layers to compare the accuracy of the models.
The *Pytorch* is used for the training which is Geometric is a library for deep learning on irregular input data such as graphs, point clouds, and manifolds.
Using RNN the recurrent neural network-based language model provides further generalization: instead of considering just several preceding words, neurons with input from recurrent connections are assumed to represent short term memory. The model learns itself from the data how to represent memory. While shallow feedforward neural networks (those with just one hidden layer) can only cluster similar words, recurrent neural networks (which can be considered as a deep architecture) can perform clustering of similar histories. This allows for instance efficient representation of patterns with variable length



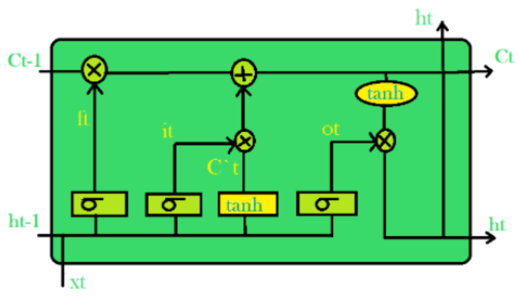
Simple recurrent neural network

2. Problem Statement:

The Basic Problem Statement here is to Implement an RNN Language Model using one of the most famous architectures i.e., Vanilla (aka Elman neural network), LSTM.
Training the Penn Tree Bank (PTB) dataset with 2 layers and also experimenting with 3 layers to compare the accuracy of the models.

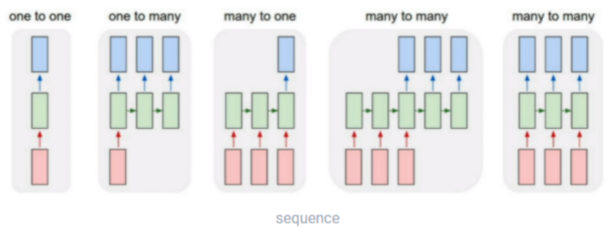
3. Model:

Long Short-Term Memory: Long short-term memory (LSTM) belongs to the complex areas of Deep Learning. It is not an easy task to get your head around LSTM. It deals with algorithms that try to mimic the human brain the way it operates and to uncover the underlying relationships in the given sequential data. Let us try to understand LSTM in length in this 'What is LSTM? Introduction to Long Short-Term Memory' blog.



Vanilla Recurrent Neural Network: Recurrent neural network is a type of network architecture that accepts variable inputs and variable outputs, which contrasts with the vanilla feed-forward neural networks. We can also consider input with variable length, such as video frames and we want to make a decision along every frame of that video.

Process Sequence:



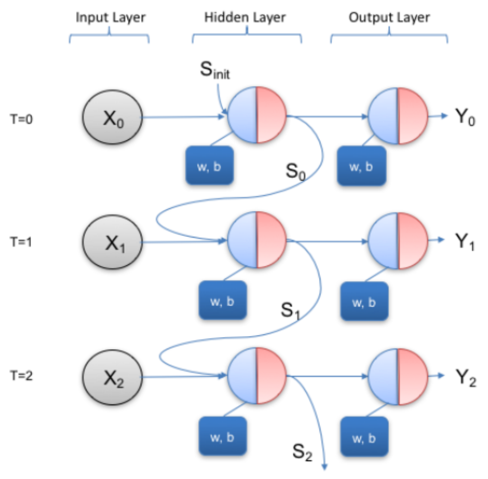
We can process a sequence of vectors **x** applying a recurrence formula at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$

Time step of an input vector is represented by x[t] and time step of a hidden state is represented by h[t]. Thus, we can think of h[t - 1] as the previous hidden state. The production of hidden state is simply a matrix multiplication of input and hidden state by some weights **W**.

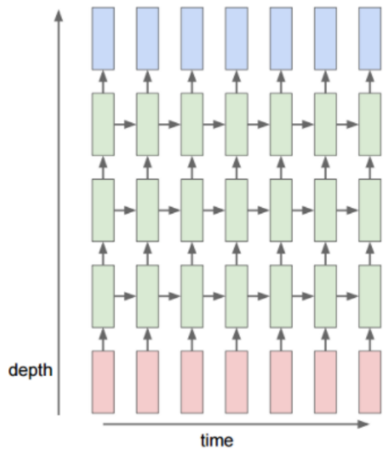
Operationally the hidden or recursive unit behaves like a typical hidden layer in that a logit is first computed and from this an activation is calculated. We will assume that the activation value is a hyperbolic tangent function, but this need not be the case. The activation function for our vanilla RNN could be a logistic function or other suitable activation function.

Our visualization above simplifies the temporal nature of the RNN as it shows the output of the RNN unit being fed directly back in to the same unit. For clarity it is useful to think about the network as having to be unrolled out over time to deal with sequences of inputs. We visualize this below by extending the network out to a sequence of 3 input units at T=0, T=1, and T=2.



Referring to the above we can see now how the output of the hidden layer at time T is passed as input to the hidden layer at T+1. Specifically the S_{T} is concatenated with the actual input X_{T+1}. At time T=1 we don't have a true hidden state from a previous step to concatenate with our input X_{0}. In this case we typically use an initial value for S which we set to 0.

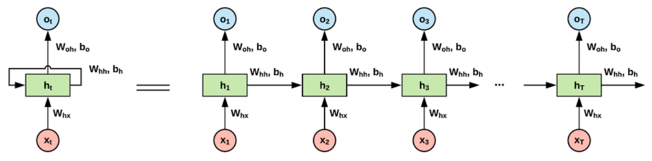
Multi-layer RNN:
We can construct a multi-layer recurrent neural network by stacking layers of RNN together. That is simply taking the output hidden state and feeding it into another hidden layer as an input sequence and repeating that process. However, in general RNN does not go very deep due to the exploding gradient problem from long sequences of data. Also, for most natural language problems, there isn't a lot of incentive to go deep for every time step. The key thing is long sequence data.



multi-layer-rnn

$$h_t^{layer} = tanh\left(W^{layer}\begin{pmatrix}h_t^{layer-1}\\ h_{t-1}^{layer}\end{pmatrix}\right)$$

In this case, the W[l] is a (hidden_dim, 2 * hidden_dim) matrix.



Sample RNN structure (Left) and its unfolded representation (Right)

4. Code Modules Explanation:

I have experimented with different calibrations of RNN using Vanilla, LSTM layers, etc.
Currently I have 3 codes and their functions explained below.

a) **Vanilla**
b) **2 Layer LSTM**
c) **3-Layer LSTM**

RNNs in PyTorch
PyTorch provides efficient implementations of a number of recurrent architectures, including the most prominent ones Gated Recurrent Units (GRUs) and Long Short-Term Memory (LSTMs).
While it is possible to build recurrent neural network architectures from basic tensor operations, it is usually much slower due to additional overhead and the sequential nature of RNNs.

a. **For Vanilla:**
Padding of sequences
Function pad_and_batch_sequences()

RNNs are often used for sequence data like audio waves, texts in the form of character sequences or word sequences, or any type of time series.
Invoking the same operations multiple times (once per sequence rather than once per batch of sequences) introduces a lot of overhead. But since sequences can vary in length, it's not possible to just stack them to form a batch tensor, so we need to pad the sequences to make them the same length.
The PTB class instantiated above is a Dataset object. It handles loading of single samples from the dataset but is not responsible for padding, batching or shuffling - that's the job of the DataLoader.

Data Creation

We can easily generate some sample data that corresponds to our usage scenario. For training we will limit all our examples to being constant length (currently 20 units below in the code). We will also set the input data to be in the range -1 to 20. We also generate the Target data straightforwardly by iterating over our dataset and applying a function that checks for the occurrence of negative values, and if found sets the output to 1 from the point at which the negative value is found, until the end of the sequence. We will create a large training data set and a smaller test data set with the same methods.

Tokenization:
Function tokenize()

As it is text data, we have to work on Text manipulation techniques like tokenization.
Tokenizer: splits sentences into char tokens.

Activate Cuda:
As graph and Deep Learning are involved, we need to train the RNN on GPU for faster processing and support.

Unrolling the Vanilla RNN for Training

Above we unrolled the Vanilla RNN over a number of time steps in order to make its operation more transparent. It turns out however that unrolling the RNN is in fact essential to its use.

Lets, consider first the case of using our Vanilla RNN model where the model parameters (i.e., the weights and bias for the hidden and output layers) have already been set. In this case it should be possible for us to feed an input sequence one unit at a time through an unrolled network and produce a set of valid outputs. We have to be careful to copy our output from time point $t$ to time point $t+1$ for concatenation with our actual input, but otherwise there should not be a problem.

However, for training things are a little bit more complex. At the end of an input sequence, we should be able to calculate a total training loss on the sequence. That loss is not just from the error on the final output symbol, but should instead be based on the true versus predicted value for the output Y at each time step. We need to propagate errors back from the last time step through multiple steps.

One question is how many steps should we propagate errors backwards for? There are two extreme conditions. In the first extreme condition we don't propagate errors backwards in time at all. In this case we will not be able to learn temporal dependencies in our data - which defeats the whole purpose of what we are trying to achieve. In the other extreme we allow errors on the final symbol to propagate all the way back, so that in theory corrections to weights at the first-time step could be influenced by an error in the final symbol. While this might be appealing in that it would allow arbitrarily long dependencies to be identified, in practice it does not work for two reasons. The first is a purely resource-based concern. The further back we allow error propagation, the more complex our model and the more resources (time and memory) that are consumed. The second issue however is more problematic and is an instantiation of the vanishing gradient problem. Essentially over a long time step the repeated multiplication of error gradients will result in the gradient approaching either 0 or infinity.

Next, let's define our RNN layer. We need to specify the hidden state size. We also ask pytorch not just to output values for every step in the sequence -- not just the final step. Finally, to help guide pytorch we give it some guidance on the input data dimensions it should expect; here 20 is the length of each example and 1 is the dimensionality of each input element, i.e., just a single number.

Network:
Class Net()

A Recurrent layer of LSTM and output layer with Linear features are added in the network.
Embeddings are then passed to the feed forward layer with a hidden state forward layer and flatten of array.
Compute the forward pass, using nested python for loops. The outer for loop should iterate over timesteps, and the inner for loop should iterate over hidden layers of the stack.

Training
For Training Optimizer used as ADAM with a Linear Decay Rate
For each sample sequence we can use the first word to predict the second, use the first two words to predict the third, use the first three words to predict the fourth, and so on. Each prediction is a classification problem where the number of classes is the dictionary size. The elements of the input sequence are the labels (except for the first element, since there is no input to predict it) and the outputs are the predicted classes (except for the last output, which goes beyond the last label).

Testing
After getting the output from the NN the output is processed using SoftMax as it helps in multiclass output.
For each predicted word we can calculate the cross-entropy loss and average it over all the predictions of a sequence and over all the sequences of a batch. Implement the training and evaluation functions.

Traditional language modelling was all about counting statistics. The most important statistics are the number of times individual words occur, but also the number of time particular sequences of words occurred. The sequences here would be small, typically sequences of length 2, 3, 4 or more. These sequences are referred to as n-grams, where 2-grams refer to sequences of 2 words, 3-grams refer to sequences of 3 words and so forth. We use 1-grams to denote sequences of length 1 words -- which are of course just individual words. NLTK has lots of tools for pulling out these sequences for us. For example, lets ask NLTK to produce all 1-grams, 2-grams, and 3-grams from the first sentence.

b. For 2 Layer LSTM
The most of the Procedures we same excluding some additional functions and layers

Function hard_sigm()
A modification of the sigmoid function, as described in the article. a defines the slope hard_sigm(x) = max(0,min(1,(ax+1)/2))

Class bound():
Here both forward and Backward propagation is added.

Class HM_LSTMCell(), Class HM_LSTM(), class HM_Net():
Here 3 Classes were defined (CELL, HMLSTM AND HMLSTM NET)

Initialized weight matrices for transition of hidden states between HM_LSTM cells
U_recur means the state transition parameters from layer l (current layer) to layer
U_topdown means the state transition parameters from layer l+1 (top layer) to layer
W_bottomup means the state transition parameters from layer l-1 (bottom layer) to layer

The Cell Class is called under LSTM and LSTM is called under Net.

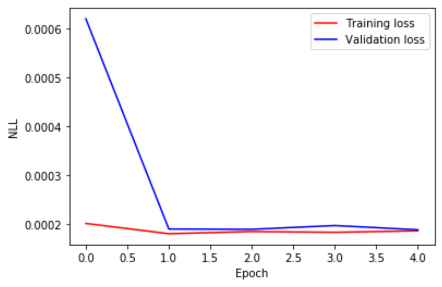
Training and Testing:
Procedures are same as Vanilla mentioned above

c. For 3 Layer LSTM
The most of the Procedures we same excluding some additional functions and layers

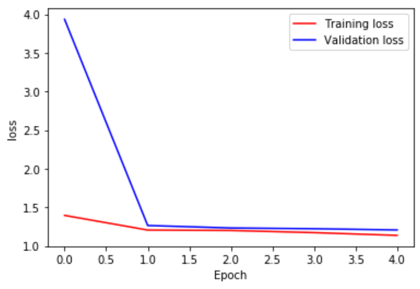
Here just the difference is the addition of extra Linear Layer in HLSTM Network

Training and Testing:
Procedures are same as Vanilla mentioned above

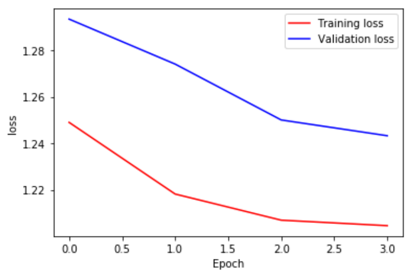
5. Results and Output:



Vanilla



LSTM(2-Layer)



LSTM(3-Layer)

Architecture	Train Loss	Test Loss
Vanilla	0.000185	0.000188
LSTM(2-Layer)	1.136	1.206
LSTM(3-Layer)	1.20452	1.243

Predicted Text:
Vanilla: "i think you are not eatly a <unk> <eos> the state of the state of the state of the state of the state of the state of th"

LSTM(2-Layer): "new a <unk> and the company as a <unk> and the company as a <unk> and the company as a <unk> and the com"

LSTM(3-Layer): "new a <unk> and the company as a <unk> and the company as a <unk> and the company as a <unk> and the com"

Now the above Prediction looks like English words generated.

6. Conclusions and Improvements:

We have successfully Trained RNN module using Vanilla and LSTM
As we can see **Vanilla** Performs Better among all.

We can tune the layer and Hyper Parameter to get better results.

One of the problems is that our model is good at learning the likelihood of characters given the recent data stream, but it isn't that good at learning to pay attention to further back information in complex ways. This problem is related to the challenge of backpropagation in general, but addressing it is an issue for future development.

References:

- https://ieeexplore.ieee.org/document/5947611
- Recurrent Neural Network (RNN) Tutorial for Beginners (simplilearn.com)
- Easy TensorFlow - Vanilla RNN for Classification (easy-tensorflow.com)
- Vanilla Recurrent Neural Network - Machine Learning Notebook (gitbook.io)
- Understanding of LSTM Networks - GeeksforGeeks
- What is LSTM - Introduction to Long Short Term Memory (intellipaat.com)