Capstone Project

Machine Learning Engineering nanodegree

Pavel (Pasha) Gyrya

November 8, 2017

Abstract

In this practice project I explore application of neural machine translation to the practical problem of text normalization that is an integral part of translating free-form written text into spoken form. Following recent research papers, I would demonstrate a sample recurrent neural network architecture that is capable enough to generate adequate translations, agile enough to be train-able using limited resources, and intuitive enough to offer practical ideas for applications in multiple fields. I would also overview training approach that helped me navigate through some of the hurdles of network tuning, and avoid settling on 'trivial' translations that may be associated with local minima of loss function.

Project Overview

Background

In this project I would like to tackle the task of automatic machine translation, inspired by recent paper published by Google research team. The problem of automated text normalization, picked up from Kaggle competition for data scientists seems like a perfect practice grounds to learn and practice the details necessary to successfully execute this task while preparing to tackle many other media-to-sequence translations (e.g. music annotation, speech-to-text transcription, etc.)

The original motivation for this problem is as follows. Many speech and language applications, including text-to-speech synthesis (TTS) and automatic speech recognition (ASR), require text to be converted from written expressions into appropriate "spoken" forms. This is a process known as text normalization, and helps convert 12:47 to "twelve forty-seven" and $3.16 into "three dollars, sixteen cents."

However, one of the biggest challenges when developing a TTS or ASR system for a new language is to develop and test the grammar for all these rules, a task that requires quite a bit of linguistic sophistication and native speaker intuition.

Problem Statement

In this competition, we were challenged to automate the process of developing text normalization grammars via machine learning, focusing on English. In other words, we are looking at a scalable algorithm that would learn to recommend translations (e.g. step-by-step translations, one word at a time), based on the input sequence of symbols. In this context, the task could be thought of as a classification problem among many classes representing differed words where word recommendation is expected at every step of generating translation text.

I would further narrow this project down to normalizing numeric text (e.g. numbers) so as to setup an initial 'proof-of-concept' algorithm and estimate resource needs. Once this process is setup, I would discuss options for scaling it to other text forms offered in the competition, and ultimately other languages and problems.

Evaluation Metrics

In this problem, I aimed for 100% translation accuracy on both development and test dataset, using translations provided in the competition as 'ground truth'. I define accuracy as % of phrases correctly transcribed into text that can be directly vocalized into a spoken form. In case of some phrases incorrectly tagged, I also visually review them for quality, augmenting data-driven evaluation with common sense review.

Analysis

Data Exploration

As part of the competition, we are provided with a large corpus of text including various form of text, already transcribed into a spoken form and broken down into logical parts that can be transcribed independently. This data is provided as table with before column (containing raw text) and after column (containing normalized text).

The text is categorized into various text entity types such as for example

- Cardinal (e.g. both '12' and 'XII' become 'twelve')
- Digit (e.g. '747' becomes 'seven four seven' after normalization)
- Ordinal ('1st' becomes 'first')
- Decimal (0.1 becomes 'point one')
- Fraction ('1/2' becomes 'one half')
- Money ($50 becomes 'fifty dollars')
- Measure ('6 mi' becomes 'six miles')
- Date (e.g. 2010 becomes 'twenty ten')
- Telephone ('911' becomes 'nine one one')
- Time ('8 a.m.' becomes 'eight a m')

Note that desired text normalization for 2010 could be either 'two thousand ten', 'twenty ten' or 'two zero one zero', depending on whether it appears in a text as representing cardinal, date or phone number. For the purposes of this project, I do not focus on inference of which type of text we are looking at – but rather focus on the process of learning text normalization for specific text types – cardinals. Within cardinal category, there is still some variety, as

a) both Roman and Arabic denomination is represented, and
b) Arabic cardinals can be separated by commas for simplicity of visual parsing as in 4,584,273

The training dataset downloaded from Kaggle web site has just under 10 Million text entities across text types, initially organized in sentences, and represented in a datasets with corresponding tags of which sentence this text belongs to (sentence ID) and text location in the sentence (token ID).

Most of the text is plain and does not need to be translated, however as mentioned above, some elements of this plain text could also provide useful context to ultimately normalize numeric text. For example, if the sentence starting with *'call me at …'* we could anticipate the next entity to represent either phone number or time rather than date, and we could parse it according to this expectation.

Only 133K of these 10 Million records represent cardinal category chosen as practice grounds for this project. This dataset was then manually split into:

- 50% training set used to tune neural network parameters,
- 25% validation set used to decide which network parameters to prefer, and
- 25% test set not used for modeling only used to review model performs as expected.

Algorithms and Techniques

I explored LSTM-based sequence-to-sequence neural network design that translates input sequences consisting of symbols - to output sequences consisting of words. Note that some translation words would be corresponding to one or more symbols (e.g. cardinal '11' should be translated to 'eleven', not 'one one').

My translation design models on common approach that included encoder, decoder and an attention mechanism. Specifically, I have implemented a version of sequence-to-sequence neural machine translation architecture using supervised deep neural network fed with examples of translations – interpreted as sequences of symbol tokens as inputs and sequences of word tokens as outputs[1].

Specifically, I have based my design on the following architecture:

- pre-processing of input and output sequences into vector representation using one-hot encoding
- encoder of input sequences into fixed-dimension vector (that is either being augmented at every step when new token is read) or sequence of vectors (separately output)
- step-by-step decoder that would generate machine translation tokens (words)
- attention mechanism that decoder would be using to focus translation on parts of the encoder output

Benchmark

I consider translations provided as part of the modeling dataset - as primary benchmark, in my quest for 100% translation accuracy of neural machine translation algorithm.

As of writing of this document, Kaggle leaderboard scores suggest that close to 100% accuracy is indeed possible, at least on the combined problem of normalizing text across multiple categories.

[1] Note that sequences of translated text could have different length compared to text being translated

Methodology

Data Preprocessing

To translate text into vector form ready to be consumed by a model, I leveraged one-hot encoding – symbol level for input and word level for translated (normalized) text.

In other words, I tokenized input text in 'before' column on a symbol-by-symbol level by first representing each symbol by vector of fixed size 27 (since there are 27 symbols possible in the cardinal category), and then concatenating resulting vectors into a matrix of size 10*27, essentially focusing on sequences of length up to 10. For sequences of length less than 10 (e.g. 3), I use zero vectors as representing trailing symbols (e.g. positions 4-10).

Similarly, I have normalized text in the 'after' column on a word-by-word level into a matrix of size 10*45 - to minimize the tokenized length of the translated sequence. One difference in tokenization for normalized text is that I have added a token '\n' to represent 'end of phrase' so as to facilitate translation process once the model is calibrated.

To ensure the translation process focuses only on tokens that are actually part of the sequence, I have added a weight to the training process. Separate weight was assigned to a pair of (phrase, decoder time step). This weight was set to one if decoder is expected to produce output at this time step, and to zero otherwise.

Algorithm Implementation

I have setup both encoder and decoded as a recurrent neural networks with long short-term memory (LSTM), see diagram below. As these networks take time to train, especially with attention mechanism – I decided to limit network to relatively shallow depth and dimensionality necessary to result in meaningful translation and manage-able training time; specifically, I used

- Depth 1 for encoder and depth 2 for decoder which in a sense has more responsibility over output generation since it needs to pay attention to all the details of English language
- Hidden state space for both encoder and decoder was set to be of dimension 64, to allow plenty of room for storing relevant information

Further, encoder was setup as a bi-directional recurrent network, to enable representing each symbol as a vector based on all the input symbols appearing on the right as well as on the left of the symbol being encoded. This could help figure out whether the symbol refers to thousands or millions, for example.

To help decoder focus translation process, I leveraged attention mechanism of feed-forward neural network that would be executed at each step of generating output word. The primary role of this attention mechanism is to decide how to weigh different time steps of input. Using soft-max activation layer would represent output of this attention mechanism as probabilities (summing up to 1) with which different time steps should be weighted.

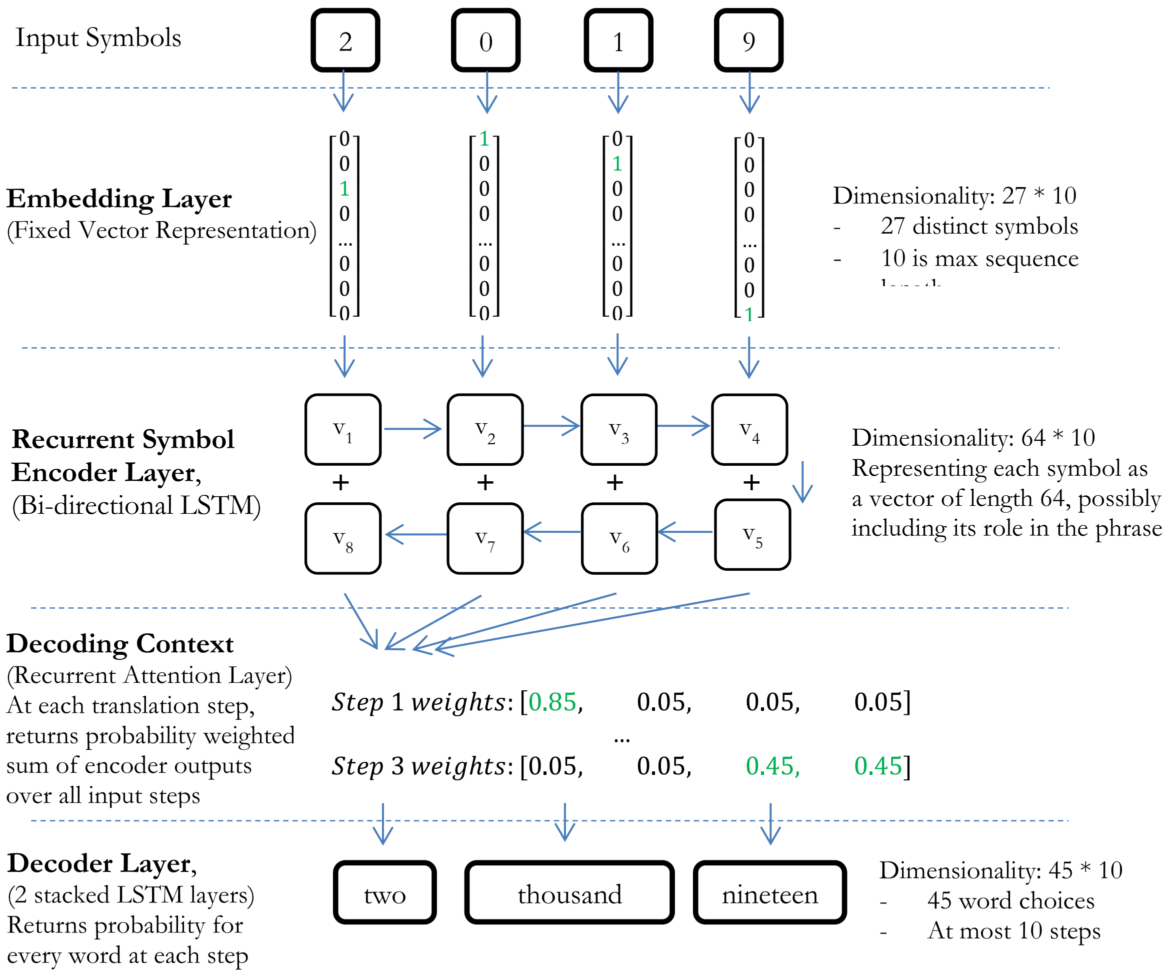

Input Symbols
2
0
1
9
Embedding Layer
(Fixed Vector Representation)
Dimensionality: 27 * 10
- 27 distinct symbols
- 10 is max sequence
Recurrent Symbol Encoder Layer,
(Bi-directional LSTM)
v1 v2 v3 v4
v8 v7 v6 v5
Dimensionality: 64 * 10
Representing each symbol as a vector of length 64, possibly including its role in the phrase
Decoding Context
(Recurrent Attention Layer)
At each translation step, returns probability weighted sum of encoder outputs over all input steps
Step 1 weights: [0.85, 0.05, 0.05, 0.05]
...
Step 3 weights: [0.05, 0.05, 0.45, 0.45]
Decoder Layer,
(2 stacked LSTM layers)
Returns probability for every word at each step
two
thousand
nineteen
Dimensionality: 45 * 10
- 45 word choices
- At most 10 steps

Figure 1: Neural network architecture – illustration of information flow for automatic machine translation

As mentioned above, I have focused translation on the cardinal numeric data to ensure the process generates adequate translation on this example. Once the process flow is established and we could see that the system is able to learn sufficient structure on CPU, one may choose to scale the process to more complex forms of texts, and execute the machine learning process on GPU such as amazon web services (AWS) cloud to leverage the benefits of parallelization.

Training Procedure

To get a neural network which is capable of generating correct translation, the model was trained on the dataset of translated samples, to minimize 'cross-entropy' loss function, which is another way of saying we'd like to maximize likelihood of sample being generated assuming our model captures the true generative process behind translations. In particular, this cross-entropy loss would be zero if in all examples provided, our model would generate 100% confidence score for correct prediction at each translation step.

Once loss function is selected, the network tuning process generally depends on

a) initial randomized model parameters (weights),
b) specific optimizer following a version of gradient descend to iteratively adjust weights so as to improve overall loss function,
c) parameters of the optimizer such as the learning rate, learning rate decay, etc.

One analogue I came up with to help me develop intuition of appropriate tuning process is as follows. Tuning network parameters is all about navigating a very large multi-dimensional parameter space in search of a global minimum. Let's think about this space as a landscape, even though it is generally a very large space with thousands of dimensions. Variants of gradient descend mechanism are analogous of a traveler following a water flow, updating her direction and speed as she goes, with various flavors such as allowing momentum and various parameters such as frequency of looking at the map somewhat related to the learning rate. Indeed higher learning rate means we rarely look at the map to update our direction and speed – as is appropriate in the beginning of a long travel.

To find global minimum, we also want to try to avoid local minima.

If we were to allow significant rainfall on this landscape, we would expect over time water to gather in both local and global minima of the loss function, making lakes of various sizes. The number and relative size of these 'lakes' is really a consequence of our choice of model architecture and underlying data.

In the meantime, the water would gather into streams that would flow into rivers to reach their ultimate destinations. These rivers could also make a lot of turns, making it more difficult to anticipate where the flow would ultimately lead.

Different starting points and different paths could potentially lead us to different destinations, or vary in time needed to reach destinations. For example, from the top of the mountain ridge, water flowing in different directions may end up in different basins. Also our landscape could have some areas of "flatland" that could take really long to get out of or where water could become stagnant. These flatlands could potentially span large areas we'd prefer to avoid. If size of the rivers is relatively much smaller than the size of the flatland areas, we may want to prefer going with the river flow more precisely, so as not to get stuck on the river bank.

In my experience on this project I noticed several issues when tuning network parameters, by tracking loss on both development and validation dataset:

1) loss function stabilizes at a very high level resulting in model translation process that is essentially trivial – e.g. the model immediately generates 'end of translation' token,
2) after learning meaningful structure, loss function keeps decreasing very slowly and takes a lot of calculations without clear hope of acceleration in sight, perhaps settling on a (local?) minimum
3) loss function may become noisy over time, as model parameters fluctuate significantly following chosen optimization process – this could result in loss becoming unexpectedly higher, and derail optimization
4) loss function on development set may become noticeably smaller than on a validation set – a situation known as over-fitting

To address the first problem of extremely slow convergence, I considered to:

i) adjust network to simpler architecture such as reducing network depth or adding residual connections between early layers and later layers to facilitate learning in the early layers
ii) leverage optimizers that allow acceleration such as stochastic gradient descend with Nesterov momentum

iii) increase learning rate to hopefully go through the training faster, though this would only help if direction of the gradient remains generally the same

iv) keep learning rate low and instead decrease batch size to one so as to go through a lot of smaller training steps. These steps, in turn would reinforce each other if we use adaptive optimizer such as RMSprop – provided they often point to the same direction

I consider the second problem of settling into valleys that could lead us to local minima – could also be related to a third problem of model weights being noisy over time due to large steps. Indeed large steps could potentially derail the gradient flow and bring us into a valley where gradient flow would lead us into some local minimum of loss function. This could be possible in situations where extrema of the loss function are very common. To mitigate these problems, I considered to

i) start with a different randomized state

ii) increase batch sample size so as to ensure we follow gradient direction more precisely, which could help us stay on the main optimization track, which hopefully leads to global minimum

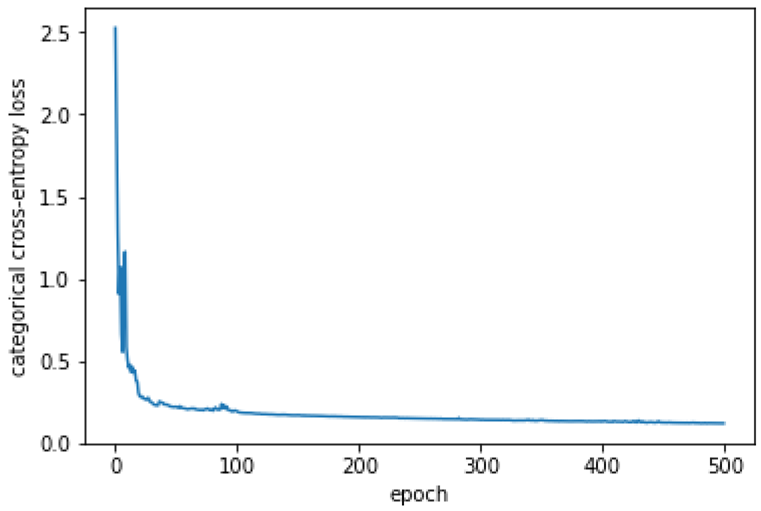
iii) ultimately decrease learning rate so that optimization takes smaller steps

iv) limit step size by applying norm or value clipping to weight update step

To addressing the last problem of over-fitting, I followed a standard practice of early stopping, ultimately picking a version of the model that has lowest loss on a validation sample, rather than on model calibration sample.

The approach that ultimately helped my model converge to a reasonable translation system is as follows:

1) Use "RMSProp" optimizer with standard low learning rate of 0.001, trained on batches of single observations to fit as many small steps as possible when going through the first 10 epochs circulating through input data
2) Increase batch size to 128 when going through epochs 11-20
3) Increase batch size to 256 when going through epochs 21-30
4) Increase batch size to 1024 to quickly go through epochs 31-100
5) Also decrease learning rate to 0.0001 when going through epochs 101-500
6) Among candidate models navigated in the last step, pick the model with the smallest validation loss



Results

Model Evaluation and Validation

This model achieves 96% accuracy on the test dataset not used for model development. The following table represents variation of match rate with complexity of the number. Clearly, the larger the number, the harder is it for translation process to get it right.

Table 1: Match Rate breakdown by token length – Testing set

Sample Text length	# samples in modeling data	Match Rate
1	8,621	100.0%
2	12,819	99.9%
3	8,018	99.6%
4	1,458	89.6%
5	1,241	71.9%
6	749	47.4%
7	374	43.6%
8+	205	5.4%
Total	**33,485**	**96.0%**

Here are a few examples of input text of length 5 where machine translation generated correct result. Clearly there is a variety in number representation that the model manages to get right.

before	Correct Machine Translation
6,000	six thousand
XVIII	eighteen
1,401	one thousand four hundred one
20881	twenty thousand eight hundred eighty one

Also here are a few examples of incorrect translation for cardinals represented by 8 symbols:

before	Expected Translation	Incorrect Machine Translation
-742,791	minus seven hundred forty two thousand seven hundred ninety one	ten hundred hundred hundred hundred hundred hundred hundred
20151956	twenty million one hundred fifty one thousand nine hundred fifty six	two million eight nine hundred hundred hundred million
LXXXVIII	eighty eight	sixty hundred hundred hundred hundred hundred twenty hundred hundred hundred

Here we see the token 'hundred' is often preferred as it is used many times in the translation process. Other from that it seems the machine doesn't really know what it is doing for these examples, and in fact never generates end of translation token. That said, analyzing output probabilities it is clear the model is not even claiming that it knows what it is doing; indeed for steps 2 onwards, most likely translation token has assigned probability less than 20%, see figure below.

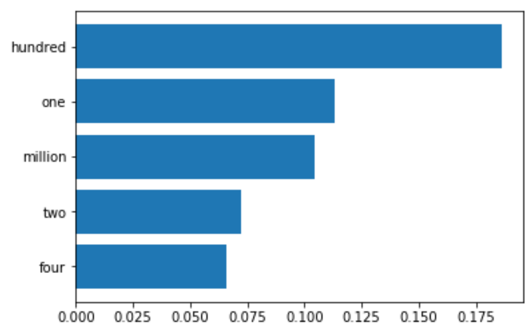


Figure 2: Translating LXXXVIII – top 5 most likely words suggested as translation word # 2

Further Development and Applications

Improving Model Accuracy

While many translations are already properly performed, especially for smaller cardinals – we cannot fully rely on this mechanism until further tuning focused on improving overall accuracy. This is a common hurdle in machine learning applications.

At this point, I assume this result could be due to either

i) insufficient representation power given to the model (dimensionality, number of layers), making it essentially impossible for the model to do better
ii) complexity of network architecture that results in difficult loss function minimization problem
iii) insufficient dataset size or insufficient weight of relevant data in this dataset

To address (i) it may help to imagine a representation that solves the problem, and at least give the model comparable representation power.

To address (ii), it may help to a) limit model complexity so as to save computational resources, and/or b) carefully adjust model training procedure to facilitate convergence. For example, random re-starts or optimizers with momentum could somewhat help get out of local minimum of a loss function; optimizers with adaptable learning rates could help accelerate convergence. Trimming update steps to certain norm or value would prohibit large steps that could side-track the process.

To address (iii), it may help to gather more data or give more weight to examples that are interesting, e.g. where current version of the model is not doing well. This idea is similar to that of a gradient boosting. For example, we could increase weight when translating longer cardinals based on length of the desired translation sequence.

Related Problems

Similar approach could be applied to many other translation applications, such as other input text categories, and languages. Additional complexity of recognizing text category could also be embedded into model calibration design, where previous data is fed into model and states of the encoder-decoder networks are propagated while the same sentence is being processed.

Acknowledgements

I have leveraged established Python libraries such as Keras and TensorFlow to implement neural translation algorithm discussed above. For neural network architecture, I have leveraged code base shared on the following GitHub repositories of seq2seq and recurrentshop shared as open source by Fariz Rahman.

After leveraging this library, I have also discovered another interesting and illustrative implementation of similar approach at GitHub location of keras-attention shared by Datalogue.