Capstone Project

Nanodegree – Machine Learning Engineer

Pavel (Pasha) Gyrya

November 15, 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 contrast two recurrent neural network architectures that are both 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 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 or areas of flat gradient.

Table of Contents

Abstract 1
Project Overview 2
Background 2
Problem Statement 2
Evaluation Metrics 2
Analysis 3
Data Exploration 3
Modeling Approach 4
Primary Approach 4
Benchmark Approach 5
Implementation 7
Data Preprocessing 7
Model Details 7
Primary Model Architecture 7
Benchmark Model Architecture 8
Training Procedure 9
Developing Intuition 10
Challenges and Solutions 11
Training of a Primary Model with Attention Mechanism 12
Training of the Benchmark Model 13
Results 15
Model Evaluation and Validation 15
Further Development and Applications 17
Improving Model Accuracy 17
Related Problems 17
Conclusion 18
Appendix 19
Background – LSTM cell 19
Background – Dropout in Neural Networks 21
Acknowledgements 22

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 may or may not 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. The following diagram illustrates distribution in 'before' text length (symbols) and 'after' text length (words).

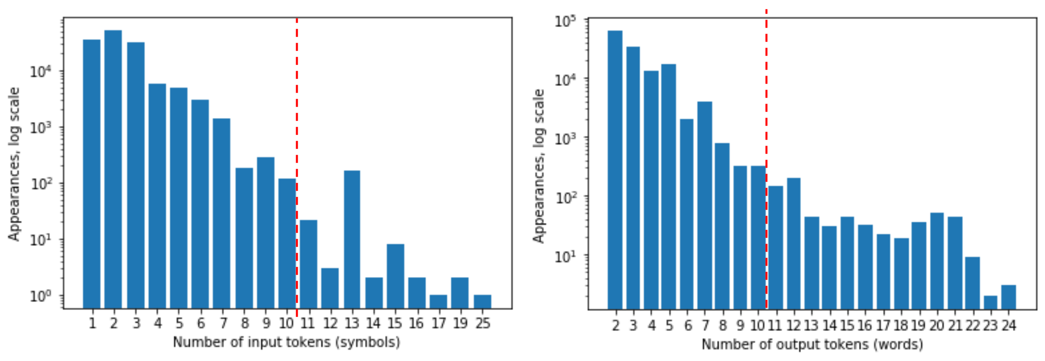


Figure 1: Length of input and output sequences – log 10 scale

Note that even among regular cardinals the task is non-trivial, since some words in the translation would be corresponding to several more symbols. For example, cardinal '11' should be translated to 'eleven', not 'one one', and '110' would be translated to entirely different 'one hundred ten'.

Since numeric algorithms typically require prescribing sequence length (may grow in calibration complexity as this length increases), and most of the numbers in the dataset provided are of length less than 10 – I chose to limit my calibration to sequences of length at most 10.

This dataset was then randomly 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.

Modeling Approach

To address this translation task, I explored sequence-to-sequence neural network design based on Long Short-Term Memory (LSTM) to ultimately translate input symbol sequences into output word sequences. The internal structure of LSTM networks is discussed in this very illustrative post, please also refer to introduction in the Appendix.

More precisely, I would ask the model to separately estimate likelihood for each word to appear at any given place in the translated sequence. The translation mechanism would then pick the most likely word candidate for each spot, until the model generates 'end of translation' token. Notice that this approach also allows us to

- come up with translation confidence score – product of probabilities associated with each word
- review alternative translations, if there are multiple translation candidates with similar confidence

Primary Approach

My primary translation algorithm design models on a 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 **attention** mechanism that would sequentially generate [the most relevant] context vector for subsequent decoder to interpret; This would effectively focus translation on parts of the encoder output
- step-by-step **decoder** that would generate machine translation tokens (words) based on attention context vectors

An example of how this translation mechanism may be applied for translations from Chinese to French, for example – could be reviewed at Google research blog, see also illustration below.

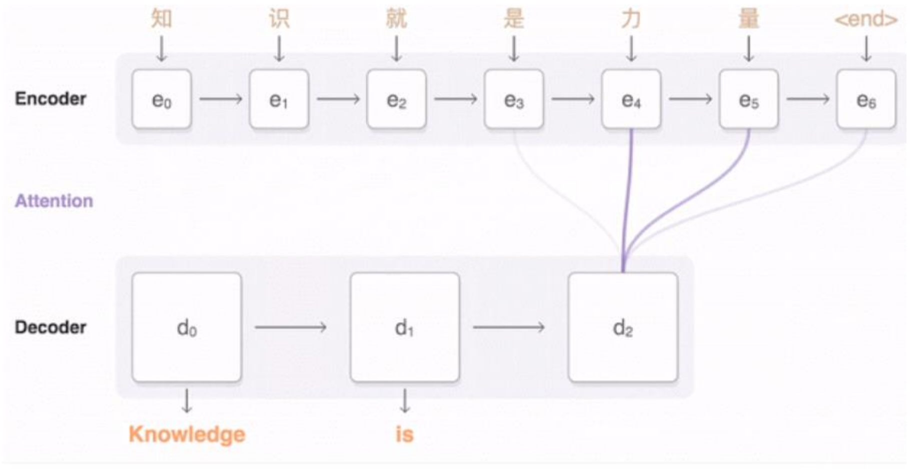


Figure 2: Step-by-step translation via neural network with attention mechanism; source – Google Research Blog https://research.googleblog.com/2016/09/a-neural-network-for-machine.html

Another illustrative post that looks at application of attention beyond sequence processing can be found here.

Please refer to section Primary Model Architecture for detailed model architecture of the neural network with attention mechanism.

Benchmark Approach

As a benchmark, I have calibrated a simpler encoder-decoder neural network (without attention mechanism). It is similarly focused on generating translations one word at a time, however the main difference is that instead of relying on attention to guide decoder – it would store relevant information internally and make it available to decoder at every translation step. I empowered this network with a single recurrent layer for

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

encoder and single layer for decoder, and plenty of hidden nodes to ensure sufficient internal memory is available to include entire text being read.

In situations when translation provided in the data disagrees with the best model predictions, I have also reviewed machine translation for accuracy to see whether model indeed fails to produce a uniquely correct translation, see Results section.

Implementation

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.

Data Preprocessing

To translate text into vector form ready to be consumed by a model, I leveraged one-hot encoding calibrated on cardinal category – 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.

Model Details

Primary Model Architecture

In my primary approach, I have setup both encoder and decoded as 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.

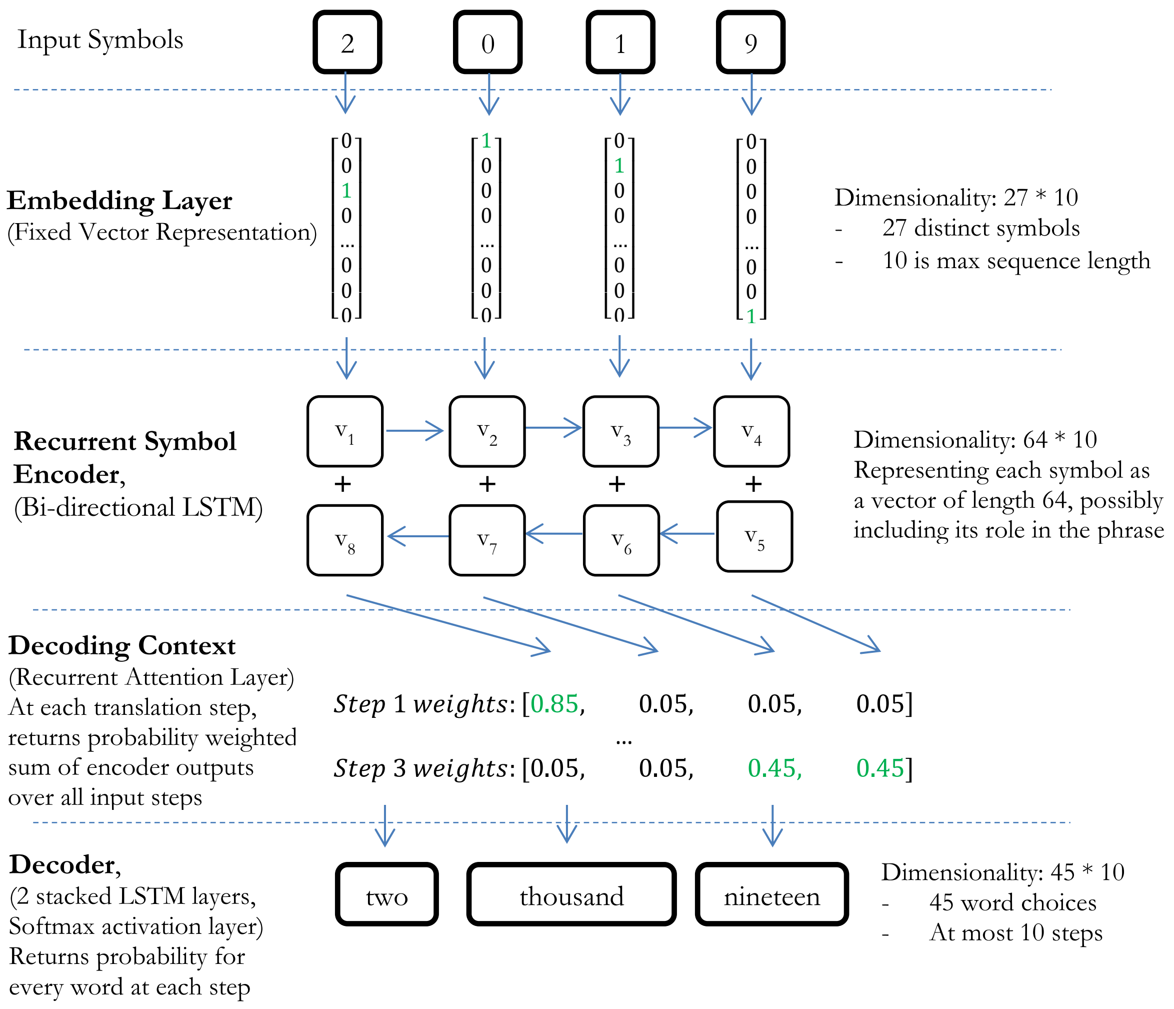

Input Symbols
2
0
1
9
Embedding Layer
(Fixed Vector Representation)
Dimensionality: 27 * 10
- 27 distinct symbols
- 10 is max sequence length
Recurrent Symbol Encoder,
(Bi-directional LSTM)
v1 v2 v3 v4 v5 v6 v7 v8
Dimensionality: 64 * 10
Representing each symbol as a vector of length 64, possibly including 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,
(2 stacked LSTM layers, Softmax activation layer)
Returns probability for every word at each step
two
thousand
nineteen
Dimensionality: 45 * 10
- 45 word choices
- At most 10 steps

Figure 3: Primary neural network architecture – illustration of information flow for automatic machine translation

This model architecture has about 115 thousand trainable parameters (weights).

Benchmark Model Architecture

Benchmark model would be a simpler version of encoder-decoder architecture with attention mechanism replaced by a bigger internal memory comprised of 500 nodes. Encoder network is thus asked to generate only one output which would encode all the relevant inputs it has processed. Encoder network is also simplified to single-direction. This architecture results in a total of about 4 Million trainable parameters, which result in a network with significant potential to overfit. To limit this possibility, I have also added regularization by means of 5% dropout for both encoder and decoder LSTM network.

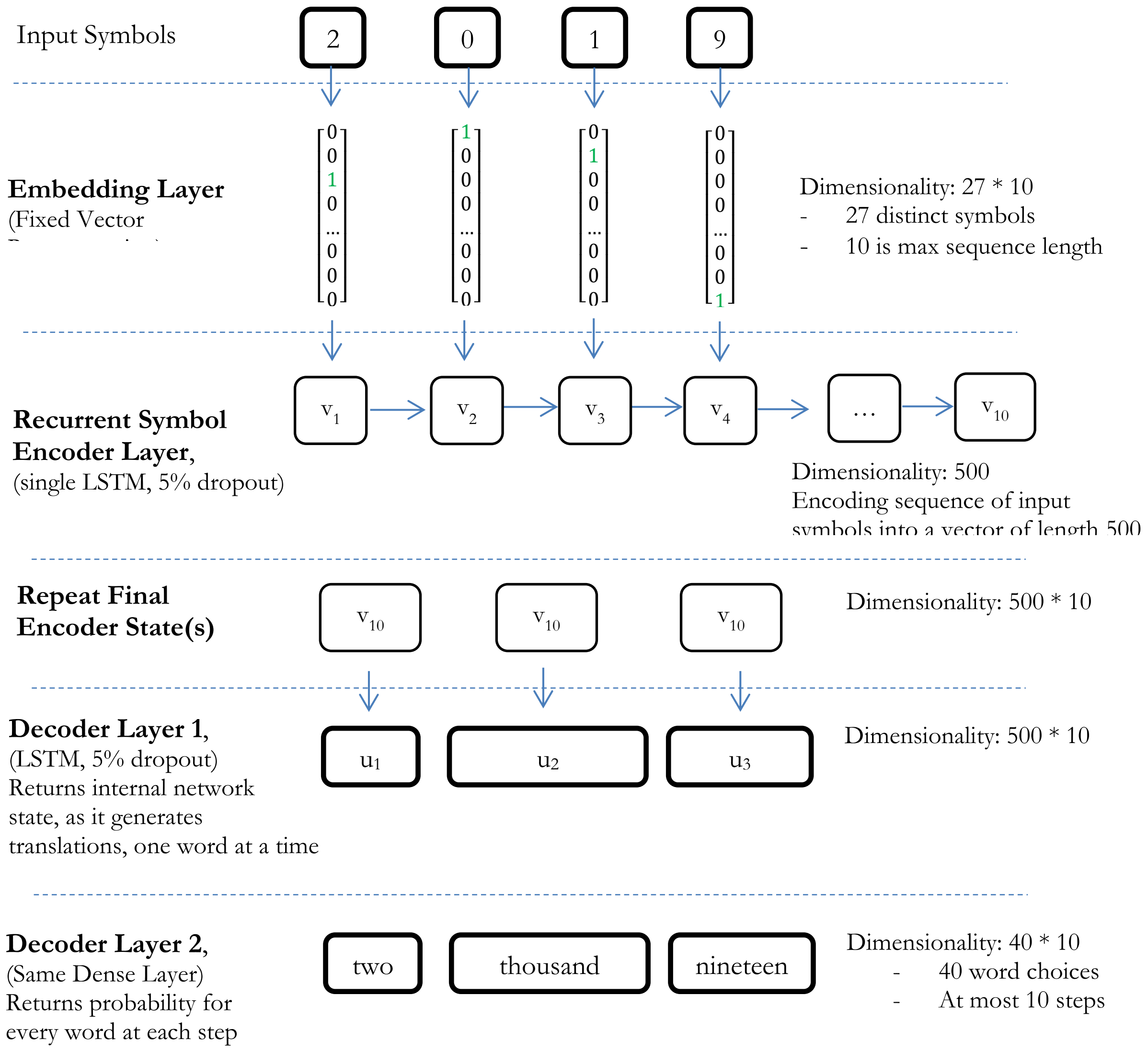


Figure 4: Benchmark neural network architecture – illustration of information flow for automatic machine translation

Note the number of possible output words was reduced when we restrict input data to input sequences of length at most 10.

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 and only 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.

The next subsection introduces an analogue I came up with to help myself develop intuition of appropriate tuning process. The rest of this section is devoted to layout of challenges and solutions, as well as laying out the settings used to tune the two models.

Developing Intuition

Tuning network parameters is all about navigating a very large multi-dimensional parameter space in search of a global minimum of a loss function that is generalize-able to validation and test data. To find global minimum, we should try to avoid local minima that may be difficult to escape by only looking around locally.

Let's think about this space of model candidates (parameters) as a landscape, even though it is generally a very large space with thousands or even millions of dimensions.

Variants of gradient descend mechanism are analogous of a traveler navigating a boat with the intention of generally following the water flow, occasionally looking around, looking at the travel history map and updating her speed and direction as she goes.

There are various flavors and parameters of stochastic gradient descend optimizers such as

- learning rate is analogous to frequency of decisions; 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
- in mini-batch mode, batch size is analogous to how carefully do we look around to decide on the next direction, or how many people we ask for directions on the street
- momentum term in stochastic gradient descend algorithms is analogous to inertia – a physical or mental tendency to maintain direction despite other stimuli

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' in any given area would depend on model architecture, underlying data and the choice of loss function.

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 and requiring more care in following the flow.

If we select a travel strategy (e.g. following water flow precisely), different starting points and different paths could potentially lead us to different destinations, and/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 much smaller compared to 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.

Challenges and Solutions

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 or keeps decreasing very slowly and takes a lot of calculations without clear hope of acceleration in sight, due to either
    a) settling on a local minimum
    b) getting in the area of flat gradient, where gradient descend process becomes very slow
2) loss function may become noisy over time, as model parameters fluctuate significantly following chosen optimization process – this could derail optimization as loss becomes unexpectedly higher
3) loss function on development set may become noticeably smaller than on a validation set – a situation known as over-fitting

The first issue (stabilization of solution loss) could happen at a very high level of loss function, or further down the line. This could result in model translation process that is essentially trivial (e.g. the model immediately generates 'end of translation' token) or subpar (model was able to learn meaningful structure, but solution is clearly far from perfect).

It may be difficult to distinguish situation 1a) from 1b), yet applying some of the solutions below could help.

To address situation 1a (network settling on a local minimum), I considered the following options to help avoid and/or escape the 'valleys' of local minima of loss function:

i) if the optimization process stalls, try to begin optimization with a different randomized state
ii) leverage optimizer with momentum[2] that could help tunnel through local extrema
iii) decrease batch size, which could help optimizer locate a 'flat minimizer' that could generalize better according to this interesting paper
iv) adjust network to simpler architecture so as to decrease curvature of the loss function 'landscape'

To address situation 1b (major slowdown), I considered the following approaches to accelerate convergence:

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 of convergence by leveraging prior weight update history – e.g.
    - Stochastic Gradient Descend (SGD) with Nesterov momentum,
    - Root Mean Square Propagation (RMS Prop), or
    - Adaptive moment estimation (ADAM)
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 (e.g. 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

[2] Note for such optimizers to work, we need to maintain history of weight update steps, which is a standard approach in Keras. However, every time the model is compiled w.r.t. a new optimizer, this history is erased.

The second issue of model weights being noisy over time – can be attributed to weight update steps being too large, as compared to the local curvature of the loss function 'landscape'. Indeed, the problem of large steps could potentially contribute to frequency of the optimization process settling into valleys of local minima – in situations where extrema of the loss function are very common. Since this could significantly derail optimization, I consider the main solution to decrease step size (perhaps selectively), by

i) decreasing learning rate so that optimization takes smaller steps as we are more invested in the solution we are working on developing
ii) limit step size by applying norm or value clipping to weight update step
iii) 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

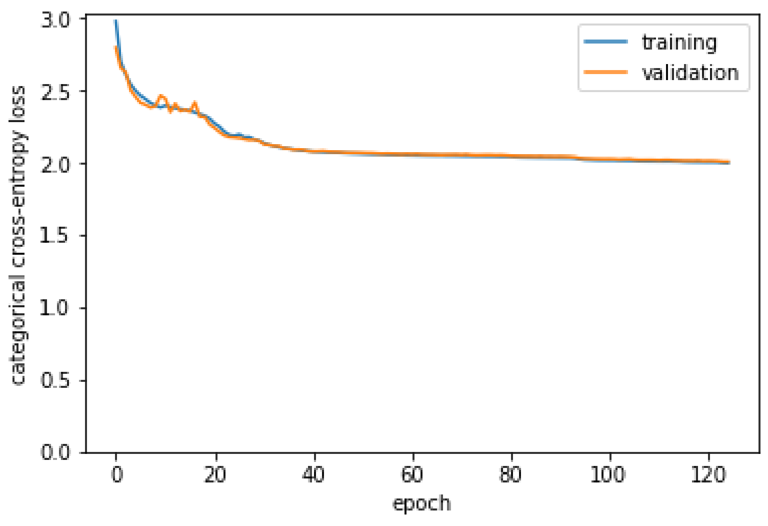
Finally, to address the last problem of over-fitting, I followed a standard practice of early stopping, ultimately picking a version of the model that has lowest loss on a validation sample, rather than on model calibration sample.

Training of a Primary Model with Attention Mechanism

The approach that ultimately helped my model converge to a reasonable translation system is as follows. For these steps, I used "RMSProp" optimizer, which adjusts

- **Epoch 1-10**: starting with standard low learning rate of 10^{-3} and reducing learning rate by 10^{-4} at every step. Batches of size 256 to quickly go through many steps;
- **Epoch 10-200**: decay learning rate by $0.5*10^{-5}$ each step from initial value of 10^{-3}, to ensure learning rates decreases linearly to zero over these 200 epochs
- **Epoch 125-200**: also reduce batch size to 16 to help escape sharp local minima with large curvature

The resulting convergence of model fit can be seen on the charts below (only showing epochs 1-125). Notice that the learning process has several plateaus followed by periods of quick improvement.



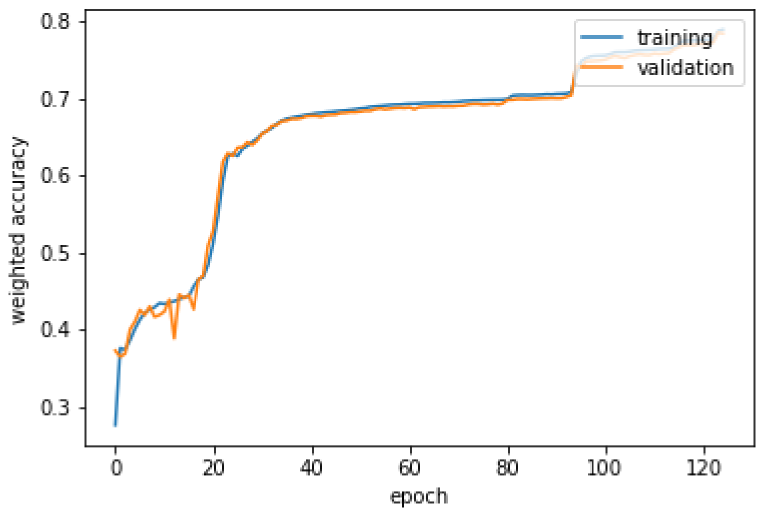


Figure 5: Training history for primary model (with attention), epochs 1-125

This training process took about a day on Intel Core i5 vPro CPU, and could be potentially greatly accelerated on GPU architecture.

Training of the Benchmark Model

Benchmark model has more weights (4 Million vs. 115K), hence takes much more time to train per epoch. That said, 15 epochs were sufficient to arrive at a good accuracy. The following settings were used for this training with "RMSProp" optimizer:

- **Epoch 1**: standard low learning rate of 10^{-3}, trained on batches of size 16 to ensure many small steps
- **Epoch 2-10**: increase batch size to 256 to speed up processing, decay learning rate by 10^{-4} each step from initial value of 10^{-3}
- **Epoch 11-15**: decay learning rate by 10^{-5} each step from initial value of 10^{-4}

The resulting convergence of model fit can be seen on the charts below. Note the accuracy is measured at a word-by-word level. Clearly model training is working well as validation loss and accuracy keeps improving towards very close to 100%. Notice that the benchmark model converges to much lower loss value while having similar accuracy score, meaning the model is more confident in correct predictions.

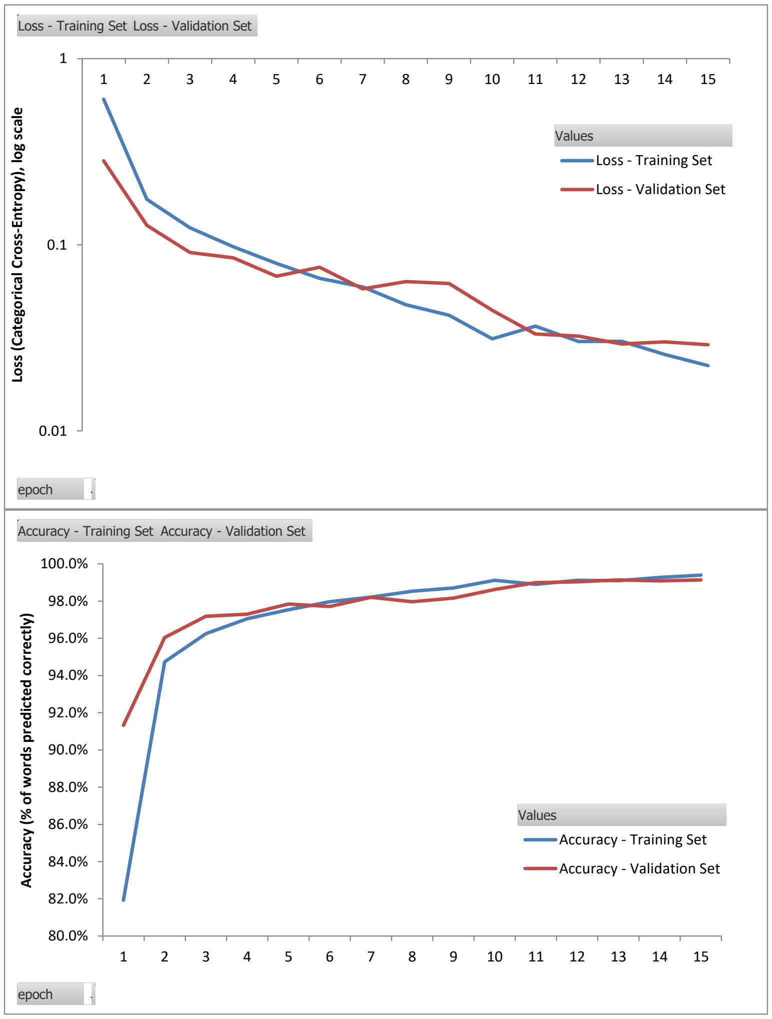


Figure 6: Training history for benchmark model (internal memory)

This training process took about a day on AWS E2 CPU instance of type t2.micro.

Results

Model Evaluation and Validation

Following training procedure performed without GPU parallelization power, the primary model achieves 98% accuracy on the test dataset not used for model development, and the benchmark model achieves accuracy of 98.6% on the same set. Translation accuracy is measured at example level, rather than at word level as done automatically during model calibration.

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 input length – testing set (input length at most 10 symbols)

Sample Text Length (symbols)	# samples in modeling data	Match Rate – Primary Model (with attention)	Match Rate – Benchmark Model (internal memory)
1	8,711	98.0%	100.0%
2	12,772	99.1%	100.0%
3	8,041	99.8%	100.0%
4	1,481	99.0%	99.1%
5	1,171	97.4%	95.4%
6	779	91.4%	82.7%
7	341	68.9%	65.4%
8+	164	16.5%	16.5%
Total	**33,460**	**98.0%**	**98.6%**

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 both models manage to get right.

Before	Correct Machine Translation (both models agree)
6,000	six thousand
XXIII	twenty three
93086	ninety three thousand eighty six

Also here are a few examples where both models give incorrect translation for cardinals of length 8:

Before	Primary Model Translation: (model with attention)	Benchmark Model Translation: (model with internal memory)
-999,999	minus four hundred ninety nine thousand nine hundred eighty seven	minus million nine nine nine thousand ninety nine
20151956	twenty million one hundred one thousand one hundred seventy six	twenty million one hundred fifty one thousand nine hundred fifty …
CLXXXIII	one hundred thirty thirty	one hundred eighty six

From these examples we see that as both models are well on the way to learning to generate good translations, as many of the words are chosen correctly. In the second example, benchmark model translation could continue, as it never generated 'end of translation' token, and got the first 10 words correct.

Analyzing output probabilities it is clear the model with attention is not claiming that it knows the right translation yet; indeed, most likely translation token has assigned probability of below 15%, see figure below.

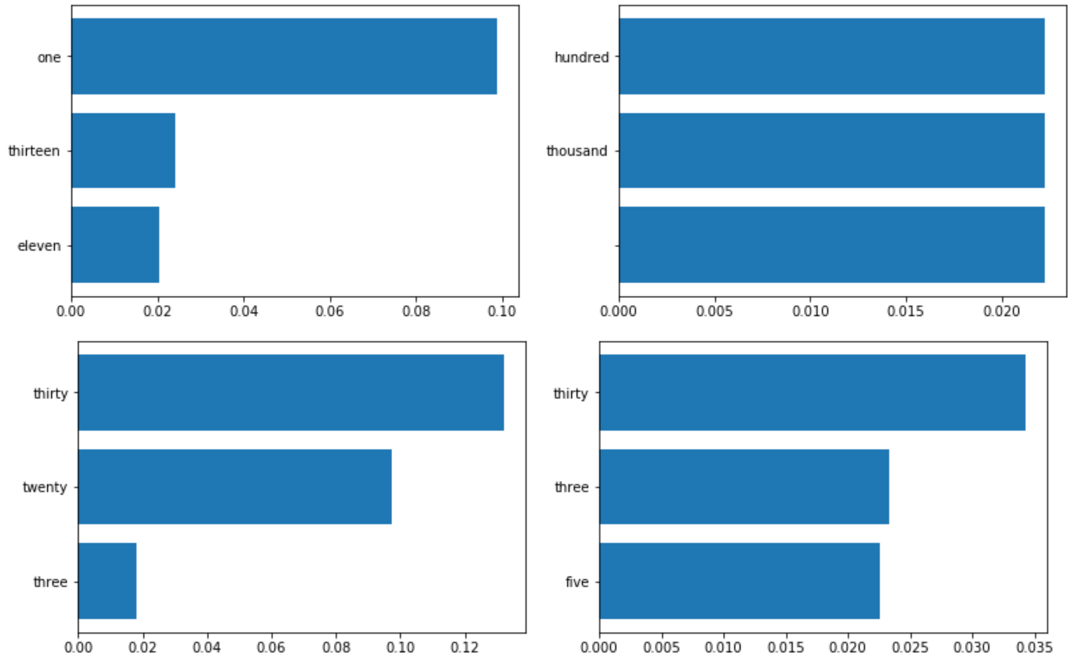


Figure 7: Translating CLXXXIII – top 3 most likely words suggested as translation candidates for words # 1, 2, 3, 4 Primary Model (with attention)

In contrast, benchmark model also generates stronger confidence scores for each word, see diagram below.

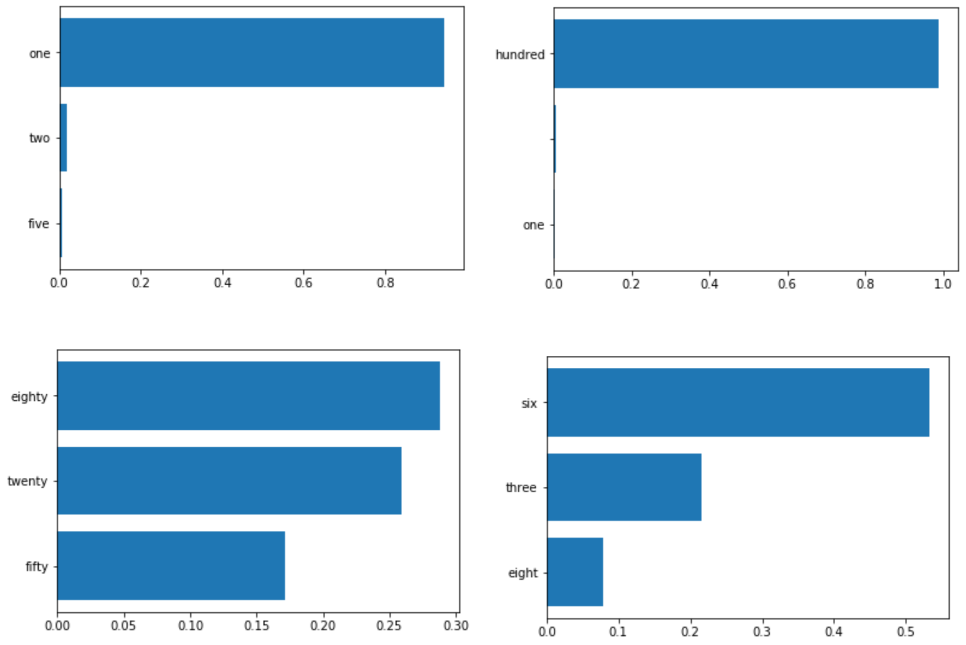


Figure 8: Translating CLXXXIII – top 3 most likely words suggested as translation candidates for words # 1, 2, 3, 4 Benchmark Model (internal memory)

Further Development and Applications

Improving Model Accuracy

While many translations are already properly performed by the primary model, 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, which could be due to the following factors:

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 optimization problem – e.g. due to significant curvature and/or many local extrema
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 minima 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. Alternatively, we could reduce weight of smaller cardinals by removing duplicate records.

Related Problems

Similar approach could be also applied to many other translation applications, such as normalizing other input text categories, or doing so in other languages which may have different way of representing concepts.

It could also be applied to two step problem of recognizing text category (e.g. phone number) based on surrounding context and only then translating it to normalized form, for example:

- Additional complexity of recognizing text category could be embedded into model calibration design by feeding the whole sentence into the model and propagating states of the encoder-decoder networks while the same sentence is being processed.

- Alternatively, most likely text category of a given text part can be predicted via a separate model taking into account surrounding text (transformed into vector form for example via word embedding, see for example this link by Google team), and as a second step, translation specific to this category could be applied.

Conclusion

Recurrent Neural Network designs are very capable mechanism able to represent complex relationships between input and output data. Overall, running this deep learning exercise was very insightful, as it exposed me to all the aspects of dealing with practical problem and tuning complex network architecture solution to solve it.

The main steps I followed were – explore data, select model objective and inputs, translate them to numeric form, select promising model architectures, tune and regularize the model to ensure good performance and generalization power.

I have constructed two similar neural network models, both proving capable of learning to give reasonable text normalization accuracy especially for small cardinals which comprise majority of the input data.

Both models work similarly by going through input text symbol-by-symbol, and then deciding how to translate it at decoder stage. The main difference between these models is – one relies on a larger internal memory to encode previously seen inputs, and the other relies on attention mechanism to generate context vectors that decoder would ultimately use to perform translation.

In order for the model with internal memory to be definitely capable of encoding large sequences, I gave it about 4 Million trainable parameters, which made training and executing such a model resource-intensive.

In contrast, the model with attention mechanisms had much fewer parameters hence takes less time to train per epoch; however the training process was also slower, requiring more epochs to learn the right representations. After comparable training time, the model with attention was less confident in the translation it was assigning, however the accuracy was clearly comparable.

In my view, the model with attention is more promising, even though it seems to take more to adequately train it:

- It is more interpretable – during translation attention mechanism points specifically to the inputs being processed by decoder, similarly to how humans could articulate their thought process.
- As it relies more on external memory, this model is easier to execute: with only 100K parameters, the model executing time is surely manageable

Several ideas for improving accuracy of a model with attention were discussed in section Further Development and Applications.

Finally, to reflect on the process, some of the challenges I encountered in this modeling process were:

- Feeding data to the model was challenging, as vectorized version of entire modeling dataset does not fit into RAM – a natural solution I followed is to execute model tuning in a step by step manner, leveraging "data generator" function to prepare vectors for just-in-time consumption
- Model tuning process was challenging especially for model with attention – despite model being in principle capable enough, the tuning process would often stall or result in a trivial translation, see section Challenges and Solutions
- I found it important to understand and incrementally test each line in the code, adding feedback for the program to illustrate what is happening – as little details would often derail the modeling process and lead to unintuitive results, for example:

- Forgetting to update variable name during a well-intended generalization update to data generator function – led to model working on data with trivial 'after' column, leading the model to prefer trivial translations – a bug that took long time to identify
- To fit into network architecture, input and output tensor dimensions had to be prescribed, and input sequences had to be padded to arrive at a tensor of fixed size; Adding zero weight for output time points corresponding to components of translation sequences that were padded – ensured more meaningful accuracy metric, meaningful loss function and ultimately meaningful translations
- To translate decoder output vectors into probabilities, I had to apply 'softmax' activation layer. It turns out that applying this activation layer within final recurrent LSTM cell of the decoder – is not the same as applying this layer consecutively to output of this cell. In fact it made my network much slower to converge, exacerbating vanishing gradient problem.

Appendix

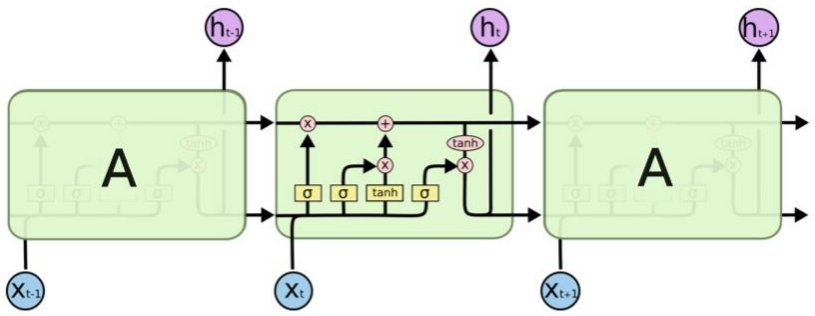
Background – LSTM cell

Long short-term memory (LSTM) networks are specially designed recurrent networks with two fixed-dimension internal memory slots that interact together as new inputs are processed to allow

i) relevant information to be retained, and
ii) irrelevant information to be discarded

LSTM networks were designed to encode (i.e. compress into fixed size vector) and maintain relevant information for much longer compared to regular recurrent neural networks with a single memory slot.

Network architectures leveraging LSTM cells have been successfully used recently in many sequence summarization tasks. Training the networks on a large body using gradient descend algorithms using 'back-propagation' of error over time – allows the network to tune itself, thus learning relevant representations of the underlying sequences that would be most useful in matching the data provided.



The repeating module in an LSTM contains four interacting layers.

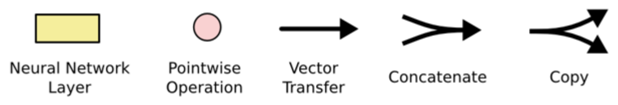
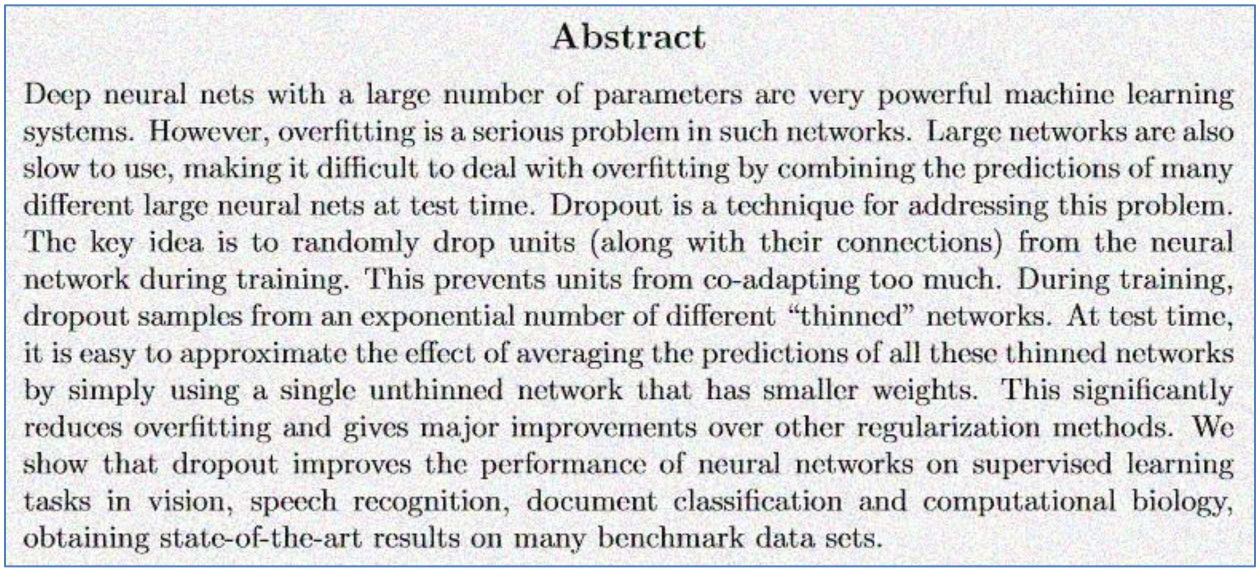


Figure 9: Internal Structure of LSTM networks – source: Colah's blog http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Background – Dropout in Neural Networks

Dropout in neural networks is an elegant regularization technique designed to minimize model over-fitting, see recent paper by a research group at University of Toronto, see abstract below.

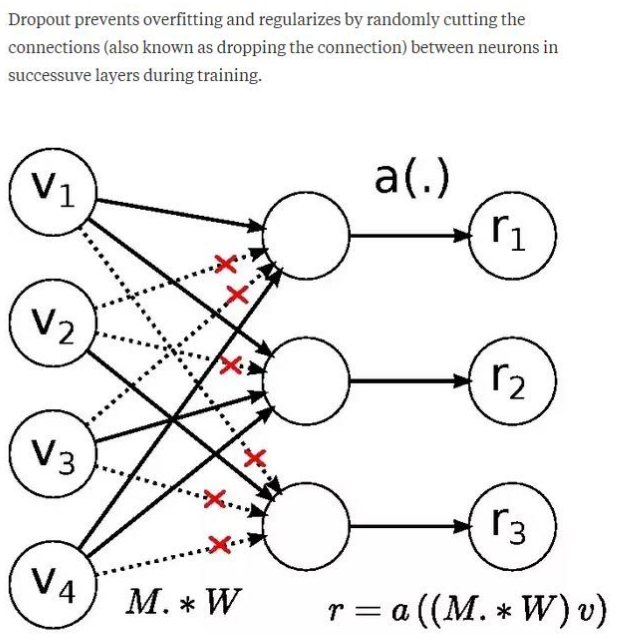


Abstract

Deep neural nets with a large number of parameters are very powerful machine learning systems. However, overfitting is a serious problem in such networks. Large networks are also slow to use, making it difficult to deal with overfitting by combining the predictions of many different large neural nets at test time. Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much. During training, dropout samples from an exponential number of different "thinned" networks. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and gives major improvements over other regularization methods. We show that dropout improves the performance of neural networks on supervised learning tasks in vision, speech recognition, document classification and computational biology, obtaining state-of-the-art results on many benchmark data sets.

Figure 10: Abstract from a recent paper regarding the use of a dropout for neural network regularization

See also an illustration below from a blog post on the same topic.



Dropout prevents overfitting and regularizes by randomly cutting the connections (also known as dropping the connection) between neurons in successuve layers during training.

Figure 11: Illustration of dropout – connections are dropped randomly during training

Acknowledgements

I have leveraged established Python libraries such as Keras and TensorFlow to implement neural translation algorithms. As I was learning neural network architecture with attention, I have started with code base shared as open source by Fariz Rahman on the GitHub repositories called seq2seq and recurrentshop.

I have added a 'softmax' activation layer to ensure the model outputs probabilities summing up to 1 at each step of generating output translation candidates.

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.

Finally, the research and development community effort that led to development and implementation of deep learning techniques, and make them easily accessible on the Web has been an invaluable help to my learning project.