Language Understanding Systems
Second project

Jacopo Gobbi
194438
jacopo.gobbi@studenti.unitn.it

Abstract

In this project we trial different recurrent neural network architectures and present their results, comparing them to different (and slightly simpler) models, particularly conditional random fields and weighted finite state transducers. Again, we are working on natural language data from the movie domain, trying to map words to concepts.

1 Introduction

The task is the same as in the first project: given a sequence of words of the form $w_1, w_2, ..., w_n$, try to find the correct concept tag sequence $t_1, t_2, ..., t_n$; we will try this out with different neural network architectures, keeping an eye on the reward of adding more parameters and making a more complex model in the presence of not so much data.
Following this introduction, there will be a brief data analysis section, summarizing the analysis of the first project and adding some considerations for what concerns word embeddings; after that we present the different architectures which we have experimented with and the experimental setup, we evaluate them and finally we discuss errors.
Implementation has been done with python and pytorch, and the code can be found in the repository.

2 Data Analysis

Our data includes 3338 sentences and a lexicon of size 1728 for the train set and 1084 sentences and 1039 unique words for the test set; the OOV rate is equal to 0.24, with the concepts **0** and **movie.name** being the most coupled with unknown words.
The data is consistent with Zipf's Law, and the distribution of concepts is dominated by **0** followed by movie names, director names and actor names; the train and test distributions are substantially similar.
For what concerns word embeddings, we have 80 tokens (65 lemmas) and 33 tokens (27 lemmas) missing a representation in the train and test set respectively. Moreover, if we treat every number and the lemma "@card@" as if they were the token "number" we obtain 66 tokens (64 lemmas) missing an embedding for the train data and 26 tokens (26 lemma) for the test set. What remains is that what misses an embedding is either punctuation, a mistype, a very peculiar word or a rating name; what we did was to remap every rating name (pg-13, r-rated, nc-17, g-rated) to "rating" and "punctuation" to "punctuation"; given also that the data is very little, we considered mistypes in the train set and used them to filter data before feeding it to the model, i.e. "scorscese" would be remapped to "scorsese" and then mapped to its word embedding.
For a more extensive data analysis, please refer to the first project.

3 Architectures

In this section we present the different architectures that have been tested, briefly describing them and the inspiration behind them, be either intuition or results from scientific research.
All the architectures are based on various kinds of recurrent neural networks, were the activation value of hidden units does not only depend on the current input but also on previous ones, having such information passed in the form of the previous output value or a hidden state which changes as the network elaborates consecutive inputs in a sequence.
In our case, the input sequence is a sentence where

every input will be the word embedding of the current token; for every architecture we have experimented with a bidirectional variant, and we also allowed the word embeddings to "unfreeze" and be changed during training.

3.1 Baseline architectures

3.1.1 WFST

Weighted finite state transducers are a generative model based on automatas with weighted edges, once learned on data, we look for the most probable sequence of tags related to the input sentence by minimizing the cost of traversed edges, usually using the Viterbi algorithm.

3.1.2 CRF

Conditional random fields are a graph based discriminative model where features are joined by feature functions, each function is weighted and, while functions themselves are to be defined a priori by the user -which can actually be an advantage since it allows us to add expressivity-, their weights are what is going to be learned through data, usually with an optimization algorithm such as gradient descent.
Formally, when we predict, we look for the label(s) maximizing the summation of the features functions over all cliques in the graph:

$$P(Y \mid X) = \frac{1}{Z(X)} \exp(\sum_{c}^{C} \sum_{k}^{K} \lambda_k \, f_k(y_c, X, c))$$

$$Z(X) = \sum_{y}^{Y} \exp(\sum_{c}^{C} \sum_{k}^{K} \lambda_k \, f_k(y_c, X, c))$$

3.2 RNN

The first, and simplest, architecture we have experimented with is an Elman recurrent neural network; here we have a hidden state that depends on the current input and the previous hidden state, while the output will depend on the new hidden state, in the following way:

$$h_t = \sigma(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma(W_y h_t + b_y)$$

Where W_i, U_i, b_i are the network parameters and sigma is an activation function, usually tanh or relu. This architecture allows us to barge into sequential inputs, but it has difficulties when it comes to training on long sequences due to a vanishing or exploding gradient, luckily our sentences are quite short and the simplicity of this model allowed it to get a decent performance.

3.3 LSTM

Long short-term memory rnns try to tackle the vanishing gradient problem by introducing more complex mechanisms related to how information is stored or deleted, with the cost of more trainable parameters and a more complex model due to the system of gates it uses.

$$i_t = \sigma(W_{ii} x_t + b_{ii} + W_{hi} h_{t-1} + b_{hi})$$
$$f_t = \sigma(W_{if} x_t + b_{if} + W_{hf} h_{t-1} + b_{hf})$$
$$g_t = \tanh(W_{ig} x_t + b_{ig} + W_{hg} h_{t-1} + b_{hg})$$
$$o_t = \sigma(W_{io} x_t + b_{io} + W_{ho} h_{t-1} + b_{ho})$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$
$$h_t = o_t \odot \tanh(c_t)$$

The memory of our model is represented by the cell state c_t, at each step we decide what information to remove from it using the forget gate f_t, while the input gate i_t will decide which values will be updated and the cell g_t will actually compute the new candidate values. The newly computed cell state will concur in the formation of the next hidden state together with the output gate o_t, h_t will represent the output for the current token but will also be used by the network in the next iteration. The activation function σ is the sigmoid.

3.4 GRU

Gated recurrent units use a reset and an update gate which are two vectors of weights that will decide what information is deleted (or re-scaled) from the current hidden state and how it will contribute to the new hidden state.

$$r_t = \sigma(W_{ir} x_t + b_{ir} + W_{hr} h_{t-1} + b_{hr})$$
$$z_t = \sigma(W_{iz} x_t + b_{iz} + W_{hz} h_{t-1} + b_{hz})$$
$$n_t = \tanh(W_{in} x_t + b_{in} + r_t \odot W_{hn} h_{t-1} + b_{hn})$$
$$h_t = (1 - z_t) \odot n_t + z_t \odot h_{t-1}$$

Again, W_i, b_i are the network parameters, while r_t is the reset gate, which specifies which (and how much) information should be dropped, z_t is the update gate which weights will be used to combine the old state and the new gate n_t in order to obtain

a new hidden state, which will also be the output for the current input, this also allows for fewer parameters compared to a LSTM model.

3.5 LSTM-2CH

Inspired by [2], we use both pre-trained embeddings and dynamic ones that are learned at training time; we have two separate LSTM layers, one will receive the sequence of trained embeddings, while the other will receive the dynamic ones, their output will be concatenated and passed through a fully connected layer to map every token to the tag space.

3.6 CONV

As in [3] and [2], we first see each sentence as a matrix of shape $(num\ words,\ size\ of\ embedding)$ and do convolution on it, using kernels of different size in order to pass over our sentence word by word, bigram by bigram and trigram by trigram, the result is used as a starting hidden memory for a GRU rnn, which we will use on our embedded tokens; what we tried to do was to start our rnn with information on the sentence at a global level.

3.7 FC-INIT

Same as in CONV model, except that the pre-elaboration of the hidden state is done by fully connected layers.

3.8 ENCODER

As in [4], we treat our problem as a sequence to sequence translation and use a GRU rnn to encode the sentence to a fixed vector (the hidden state), we then pass it to another GRU rnn, the decoder, which will use it as a starting hidden state while being given as first input a start-of-sentence token, and its own output (predicted tags) as successive inputs.
Differently from [4], we did not observe any improvement by feeding sentences in a reverse order to the encoder.

3.9 ATTENTION

As in ENCODER, but we also employ an attention mechanism on the outputs of the encoder, which allows the network to focus on a specific part of the sentence. Our attention mechanism is quite simple, we compute attention weights with a single fully connected layer receiving the embedding of the current word concatenated to the last hidden state as an input, obtaining a weight for each time step; we then apply our attention vector to the matrix of outputs of the encoder, and concatenate the result with the word embedding before feeding it to the GRU.

3.10 LSTM-CHAR-REP

Here, each word is represented as a $(\#chars,\ embedding\ size\ of\ chars)$ matrix by using character embeddings, and similarly to the works of [5] and [6] we do convolution on it in order to obtain a new vector representation.
This vector (for each word) is then concatenated to the word embedding and passed to the LSTM as an input for the current word.

4 Experimental setup and regularization

The architectures have been tried on tokens and lemma sequences, with and without using the POS tag as a feature; hyperparameter search was done on a dev set made of about 20% of the train data.
As word embeddings both pre-trained and trained on train data have been employed, while char embeddings have been obtained from the pre-trained word embeddings, which have a dimension of 300, while for the trained ones (both word and character embeddings) we have tested with dimensions 20, 50, 100, 200 and 300.
Our models were regularized by dropout layers (adding between some layers of our model a layer that randomly sets values to 0) and having a maximum value for the norm of the embeddings.
As a sort of word level dropout every word had a 5% chance to be considered unknown, intuitively we want the network to be used to receiving unknown words even when we train our own embeddings on train data, which would result in having no unknown words in the training phase.
Hyperparameters search was done by randomly sampling, for each model, among the following: sequence (lemma, tokens), batch (15, 20, 50, 100, 200), epochs (15, 20, 25, 30), lr (0.001, 0.01, 0.1), embeddings (google and our own of different dimensions), hidden size (100, 200, 400), frozen (true, false), bidirectional (true, false), drop rate (0, 0.3, 0.5, 0.7) and embedding norm (2, 4, 6, 8, 10); we then fine tuned by running combinations that looked promising given the rest of the trials but which were not sampled, i.e. a model not having trials combining a learning rate of 0.001 and google embeddings.

5 Evaluation and discussion

As a baseline we have used the results of the previous project, where we obtained a $F1$ score of 82.96% with WFSTs, and results from applying a CRF by searching the correct features and training parameters within reason, which allowed us to get to a $F1$ score of 85.43%.
The features used by the CRF were the token and its lemma, postfix and prefix, a $(-3, 2)$ window centered on the token that would also consider the pos tags, and $(-2, 0)$, $(-1, 1)$, $(0, 2)$ concatenations that would consider the pos tag as well; moreover for each word we added the word embeddings of the 3 previous words, itself, and the 3 following words as features for each token, also adding the character embeddings of the characters composing the current token.

Architecture	hidden size	epochs	batch size	drop rate	emb norm	¬br fr F1	br fr F1	¬br ¬fr F1	br ¬fr F1	#params for best score
WFST	(1 state, 2994 arcs) ∘ (7907 states, 22195 arcs) = (7907 states, 842178 arcs)									F1=**82.96**
CRF	$(-3, 2)$ window of tokens and pos tags + prefix + postfix + concats								70K params	F1=**82.73**
CRF	all of the above + $(-3, +3)$ word embs + current token char embs								173K params	F1=**85.43**
RNN	200	15	50	0.30	4	74.32	81.72	74.60	**83.96**	1264K
LSTM	200	15	20	0.7	6	70.03	79.98	76.74	**84.57**	1505K
LSTM CHAR-REP	400	20	20	0.7	4	72.39	79.92	76.20	**85.41**	2085K
GRU	200	20	20	0.5	4	73.56	83.68	76.93	**85.47**	1424K
LSTM-2CH	200	20	15	0.3	8	-	-	73.80	**83.76**	1310K
CONV	200	20	20	0.5	4	79.09	84.19	83.45	**86.17**	2646K
FC-INIT	100	30	20	0.3	4	78.87	84.51	81.44	**84.95**	2805K
ENCODER	200	30	20	0.7	4	61.80	70.48	73.32	**79.00**	1559K
ATTENTION	200	15	20	0.3	4	79.25	80.17	79.62	**82.67**	1712K
LSTM-CRF	200	10	1	0.7	6	81.32	85.99	82.45	**87.47**	1507K

Performances on test data for the different architectures with bidirectional (br) and without (¬br), with frozen embeddings (fr) and with embeddings that can be modified during training (¬fr), and both. Only the best performing hyperparameters are reported for each model, so the rest of the scores (frozen, not bidirectional, etc.) are probably not going to be the best we could get with those settings.

Architecture	min F1	avg F1	max F1	std	# params
RNN	81	**82.55**	83.96	0.57	1264K
LSTM	82.67	**83.76**	84.57	0.46	1505K
LSTM-CHAR-REP	82.0	**84.28**	85.41	0.63	2085K
GRU	76.56	**84.29**	85.47	1.20	1424K
LSTM-2CH	81.22	**82.68**	83.76	0.61	1310K
CONV	84.05	**85.02**	86.17	0.50	2646K
FC-INIT	82.22	**83.93**	84.95	0.44	2805K
ENCODER	71.25	**76.39**	79.00	1.55	1559K
ATTENTION	71.86	**79.77**	82.67	1.80	1712K
LSTM-CRF	84.75	**86.11**	87.47	0.63	1507K

Difference in performance based on random parameters initializations, 50 runs for each architecture; hyperparamaters from the first table have been used.

Going back to neural networks, given the limited amount of data, searching for the correct hyperparameters was not as simple as hoped; fortunately some features were a no-brainer, unfreezing the embeddings and allowing them to be changed during learning and using the bidirectional variant

instead of the "normal" one always led to an improvement; while using POS tags as features or lemmas in place of tokens did not bring to a noticeably better performance.
Again, due to the restricted quantity of data and the closeness of the results, we can not properly say that an architecture would be better than another for every pair, we will try to appreciate simplicity (in the form of a lower number of trainable parameters) together with scores.
The best performances have always been achieved with a learning rate of 0.001 and google embeddings; results based on embeddings trained on train data were not that far behind, leading to a drop of the $F1$ score from 0.5 to 3 points when the embedding size was maintained between 100 and 300, bigger drops would occur if the embedding size would get to 50 or lower; we can attribute this to two reasons: one is that the google embeddings have been trained on way more data, the second is that given the OOV rate, we are going to end up with much more unknown words.
Given the generally better performances of models where the hidden state was pre-elaborated in a simple way (CONV, FC_INIT) with respect to more complex and sequence aware strategies like ENCODER and ATTENTION, it seems like the former tend to do better when we have fewer data; it might be worth exploring more convolution strategies, like convolution on the sentence as a matrix of shape $(\#words,\ \#characters\ in\ word\ *\ character\ embedding\ size)$, which would also allow us to get rid of unknown words completely.
It could also be useful to devise a better way for randomly considering words as unknown, since currently it is a uniform 5% chance on every word.

5.1 Error analysis

Errors are similar to the ones made by the WFST developed during the first project.
We have a class of errors that is caused by a concept appearing few or no times at all in the training data, which are: **award.category**, **director.nationality**, **movie.star_rating** and **movie.type**.
Another type of error is related to confusion, where **I-award.ceremony** is often mapped to **0** due to how the pairs "academy award" and "oscar award" are labeled in the train data. **Movie.release_region** and **movie.location** are also mismatched, since these concepts are both related to country names.
One of the strong advantages of WFSTs compared to our models is needing less data, and that can be noticed in sentences where numbers were written as ngrams including "point", i.e. "twenty one point two" to indicate 21.2; the few examples in the training set were enough for the WFST to build a prior that would map those words to **movie.gross_revenue**; while the neural networks did not completely fail on those instances, it seems that basing the training on a more discrete approach (words and their counts) is helpful when data is limited and a few useful examples risk getting lost in the continuous space; a better pre-processing of the text would probably have helped in this case.
An important aspect of our architectures is that they use word embeddings, and when using pre-trained ones that means we will have access to a big vocabulary even if our train data is limited, leading to a lower risk of meeting unknown words at test time; one example of this is "new york city", since york and "new <unk> city" were not part of our train set, the WFST failed to build a prior on them and later failed to predict on it being a movie location; there was instead a vector (the embedding) available for york, placing the word near other cities in terms of distance.
There were some interesting facts to be noticed about the strengths and weaknesses of different architectures:

- RNN, LSTM and LSTM2CH: while these three perform worse than CONV, they have a much higher $F1$ (85 compared to 71) on sentences containing **award.ceremony**, which seems to be caused by CONV being more prone to go for something different than **0** when in doubt.

- GRU: this model is performing best on character names, ($F1$ of 80) with a 6 points advantage from the second best performer.

- LSTM-2CH has a big drop of performance on **movie.subject** and **person.name** with respect to the average, and when confronted to the standard LSTM model it appears as if the model has become much more confused, probably because we have assigned half the hidden size to each LSTM module

compared to what we have assigned to the normal LSTM architecture.

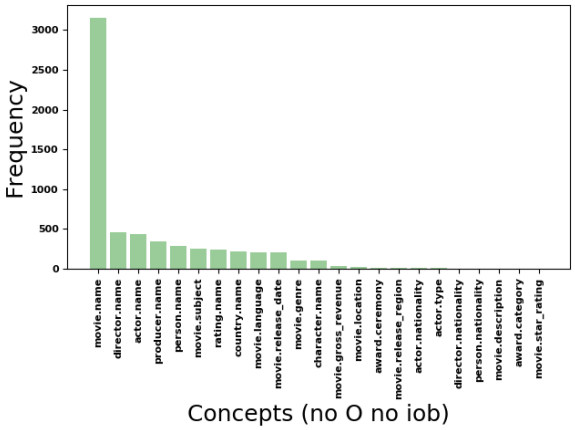
- Interestingly enough, RNN, LSTM, LSTM-2CH and LSTM-CHAR-REP are having the exact same precision and recall (and thus $F1$) on **award.ceremony**, showing that the second channel of LSTM-2CH and the character embeddings of LSTM-CHAR-REP are not helping on this particular category.

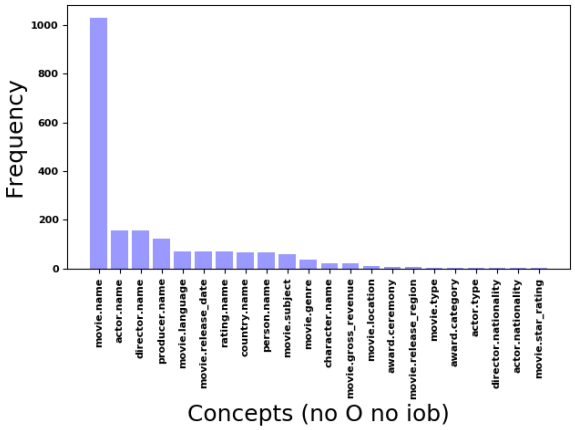
- While performing below average in general, ENCODER and ATTENTION both have a huge drop on **award.ceremony**, having an $F1$ of 0 and 18 respectively (while other models would have $70 - 85$); while we do not have an exact explanation for this, we can notice that the categories for which there is the highest difference in $F1$ compared to the other models are usually the ones belonging to the less frequent concepts in the training data, suggesting that encoding/decoding is much more data hungry than the rest of the approaches.

Appendices

A Corpus

The corpus is about data from the movie domain, in particular, each token has an associated concept that could be of interest in a task such as slot filling for movie queries from users; as an example, we would like to map "star of thor" to "**O O movie.name**" (where a word tagged as **O** means it has no interesting information to us), in order to correctly retrieve what the user is asking about.
The dataset is already split in train and test, having respectively 3338 sentences with 1728 unique words and 1084 sentences with a lexicon of size 1039, the OOV rate is equal to 0.24.
Currently, unknown words in the test set are mapped to **O** 92 times and to **movie.name** 69 times, these two tags dominate the others in "unknowness", this is quite reasonable, movie names are usually kind of original, and while there are many movies produced every year, concepts like a famous director or a genre tend to be more long-term, and thus less prone to be unknown words; for this reason we argue that if the dataset was to be larger, we would see an even bigger gap between what concepts tend to be paired with unknown words and those who are not, with **O** and movie names leading by a big margin.
As expected, both datasets agree to Zipf's law, moreover, they are similar in their distribution, as can be seen in the following charts (green color for the train set, purple for the test set).
As a somewhat marginal detail, we report that the vocabularies of concepts differ, the lexicon of the train concepts has a size of 24, while it is 23 for the test one: **movie.type** is in fact a concept present in the test set but not in the train one, while it is the opposite for **person.nationality** and **movie-description**; given that the amount of tokens being tagged with these concepts is a negligible fraction of the data (shown in the last table), this will not affect results in a meaningful way.





To tune hyperparameters the training data has been split in train/dev sets with the latter having roughly 20% of the original trainining set, the split has been done by random sampling.
Tag counts for train, dev, and test set are shown in the following (and last) table.

TAG	Train Freq.	Dev Freq.	Test Freq.
movie.name	2527	630	1030
director.name	381	74	156
actor.name	354	83	157
producer.name	266	70	121
person.name	229	51	66
movie.subject	207	40	59
country.name	172	40	67
rating.name	171	69	69
movie.language	169	38	72
movie.release_date	156	45	70
character.name	87	10	21
movie.genre	76	22	37
movie.gross_revenue	31	3	20
movie.location	17	4	11
award.ceremony	8	5	7
movie.release_region	7	3	6
actor.nationality	5	1	1
actor.type	2	1	2
movie.description	2	0	0
director.nationality	2	0	1
person.nationality	2	0	0
award.category	1	0	4
movie.star_rating	1	0	1
movie.type	0	0	4

References

[1] Giuseppe Riccardi, unitn course of Language Understanding Systems (2017-2018).

[2] Kim Yoon *Convolutional Neural Networks for Sentence Classification*. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014

[3] Majumder Navonil, Poria Soujanya, Gelbukh Alexander, Cambria Erik *Deep Learning-Based Document Modeling for Personality Detection from Text*. IEEE Intelligent Systems, 2017

[4] Ilya Sutskever, Oriol Vinyals, Quoc V. Le *Sequence to Sequence Learning with Neural Networks*. CoRR arXiv, 2014

[5] Kim Yoon, Jernite Yacine, Sontag David, Rush Alexander M. *Character-aware Neural Language Models*. Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, 2016

[6] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, Yonghui Wu *Exploring the Limits of Language Modeling*. 2016

[7] Mikolov, Tomas and Sutskever, Ilya and Chen, Kai and Corrado, Greg and Dean, Jeffrey *Distributed Representations of Words and Phrases and Their Compositionality*. Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2, 2013