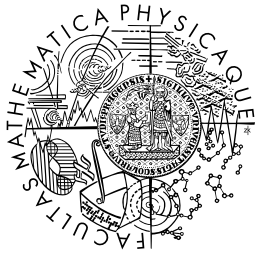


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

František Trebuňa

Generating text from structured data

Institute of Formal and Applied Linguistics (ÚFAL)

Supervisor of the bachelor thesis: Mgr. Rudolf Rosa, Ph.D.

Study programme: Computer Science (B1801)

Study branch: General Computer Science Bc. R9 (NIOI9B)

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Dedication.

Title: Generating text from structured data

Author: František Trebuňa

Institute: Institute of Formal and Applied Linguistics (ÚFAL)

Supervisor: Mgr. Rudolf Rosa, Ph.D., Institute of Formal and Applied Linguistics (ÚFAL)

Abstract: Abstract.

Keywords: text generation structured data natural language processing neural networks

Contents

Introduction

1. Problem statement

In september 2020 I read a blog by [Karpathy, 2015]. He created a neural network consisting of only one LSTM cell and trained it to predict a character based on all the previous ones. The network was trained on a corpus of all the plays by Shakespeare. During inference the last predicted character was fed as the input to the network and this way it could create a really good looking Shakespeare-like text. Then I began to explore the possibilities of generating a text conditioned on some input parameters. How to construct a network that could be told to generate a sad, happy, or sarcastic sounding text?

1.1 Data to text generation

Known datasets (WIKIBIO, WeatherGov, RoboCup) -¿ short description, the generated summaries are one-two sentences long. Rotowire -¿ really long summaries although only a fraction of the number of unique tokens from the WIKIBIO dataset. Only short description of the dataset, the statistics and observations are in the second chapter.

1.2 Fantasy sports

What it is, where it originates, the relation to basic optimisation problems, why NLGenerated summaries could be added value.

1.3 My goal

Fluent text which captures the important statistics from the table

2. Data

The goal of the thesis is to explore the possibilities of generating natural language representation of a sports match. Several questions arise from the assignment. Are we trying to create a general solution which would work for any sport ? What is the format of the input data ? How should the generated text look like ? In this section I try to answer these questions and present the challenge which we will deal with throughout the thesis.

2.1 Narrowing the Task

The differences in the number of players, the environment, the rules etc. prevent us from creating one system for all the sports, therefore the task will be narrowed to one particular sport, the basketball.

During a match of basketball, a set of statistics about teams and players is gathered. We want to generate a text, which will summarize and pick the most important information from the statistics.

2.2 Data requirements

Our approach is to train a deep neural network model. We opt for supervized training regime where each input table is associated with a target text, and the model learns a conditional probability $p(text|input\_table)$.

In the introduction I already pointed out that neural networks are excellent at learning a language model. However we want to learn to generate a text conditioned on the *concrete match statistics*.

This places high demands on the quality of the texts which are used as targets during training. The summaries should be connected to statistical data about the match, and contain the least possible amount of subjective, emotional information. Otherwise the model wouln't have possibility to learn how to interpret its inputs, and instead it would learn how to generate some invaluable emotional phrases.

It is known that a lot of data is needed to train a deep neural network (e.g. [Sennrich et al., 2016] trains the neural machine translation system on 4.2 million English-German sequence pairs). Therefore it's unfeasible to collect and prepare our own dataset, and we choose to select from a high variety of publicly available datasets.

I've chosen the RotoWire dataset [Wiseman et al., 2017]. It contains a set of statistics-summary pairs, summaries being extracted from the portal focused on fantasy sport news, `https://www.rotowire.com/basketball/`.

2.3 Summaries

In section 2.2 we have already discussed our demands on the summaries which will be used as targets.

[Wiseman et al., 2017] experimented with one other summary origin, `https://www.sbnation.com/nba`, where articles are written by fans for fans. However such summaries haven't met the requirements ("many documents in the dataset focus on information not in the box- and line-scores") and the neural networks trained on these kind of summaries "performed poorly".

Therefore we experiment only with the summaries from Rotowire fantasy sports news.

2.3.1 Fantasy sports

According to [Tozzi, 1999], the origins of the phenomenon of fantasy sports can be dated to the beginning of 1960s. The article tells a legend about a restaurant called La Rotisserie Francaise. There a group of "sportswriters, accountants and lawyers" started the first fantasy sport league. Rotowire, which name is derived from the name of the restaurant, is the source of all the target summaries in the dataset.

Let's proceed and discuss the rules of the fantasy sport according to Rotowire[1]. The fantasy league is created on top of a real world league, e.g. NBA. If you want to play, you choose a subset of NBA players and gain points based on their performance in the real-world NBA matches. You have limited resources and better players cost more. The selection must contain a basketballer for each position (therefore it is not possible to draft e.g. 5 point guards and no power forward). The points are awarded according to the role, so that defensive and offensive players gain you points for different achievements in a match. The leagues differ in their specifics and I advise an interested reader to check the Rotowire fantasy league rules[1].

2.3.2 Fantasy sports news

To succeed in the fantasy sport, one must keep a good track of the player statistics and injuries, the tendencies of team performances, the trends causing the emergence of a future star or a burn-out. From the beginning there exists a form of news specializing directly on fantasy league players. `https://www.rotowire.com/` is one of the most known ones. The articles tend to summarize the most important statistics as well as present a deeper understanding of the events that happened during the match. As noted by [Wiseman et al., 2017], the articles are the ideal candidates for the target summaries of the structured data from the dataset.

2.4 Structured Data

The structured data is in form of multiple tables of statistics related to a particular NBA match. In this section I examine the tables, and hypothesize about their possible roles in the generation of a summary.

[1] `https://www.rotowire.com/basketball/advice/`

To begin the examination, each datapoint contains a date when the match was played. This information isn't leveraged in the summaries[2], therefore I opt not to use it.

2.4.1 Team Statistics

A datapoint contains a separate table with statistics, called the *line score*, for each of the opposing teams. Each *line score* contains 15 fields, some of which we will discuss in this section. The full listing can be found on the official github of the dataset [3]

Firstly there are fields with the team name and the city where the team is located.

Next there is a set of numerical statistics. We can divide them to the contextual statistics (the number of wins and looses prior to the actual match), the team-totals (*Team-total* means the overall team statistics. E.g. *TEAM-PTS* is the sum of all the points scored by the players of the team), and the per-period point statistics (*TEAM-PTS_QTR1* - *TEAM-PTS_QTR4*). Table 2.1 is a part of both line scores taken from the Rotowire dataset.

Name	City	PTS_1	AST_2	REB_3	TOV_4	Wins	Losses	...
Raptors	Toronto	122	22	42	12	11	6	...
76ers	Philadelphia	95	27	38	14	4	14	...

Note: The statistics are accumulated across all the team players

$_1$ Points; $_2$ Assists; $_3$ Rebounds; $_4$ Turnovers

Table 2.1: An example of the team statistics from the development part of the Rotowire dataset

2.4.2 Player statistics

The player statistics are gathered in a table called *box score*. In each row there is an information about the player name, the city he plays for (since there are 2 teams originating in Los Angeles, the distinction is made by calling one *LA* and the other one *"Los Angeles"*), his position on the starting roster (or the *N/A* value if he isn't in the starting lineup), and another 19 fields with statistics summing up the impact of the player in the match (e.g. points, assists, minutes played etc.). Again, the full listing can be found on github[3] and table 2.2 contains an example of a *box score* from the dataset.

2.5 Relation of summaries and tables

In this section I would like to show the relationship between the structured data and the target summary. Let's observe the data-point from the development

[2]This isn't technically true, as the majority of summaries contain a phrase like *"Team A defeated Team B on monday."*, however since the date is in form *DD/MM/YYYY*, there isn't any chance the network could deduce the day of the week from the date.

[3]`https://github.com/harvardnlp/boxscore-data`

Name	Team City	S_POS$_1$	PTS$_2$	STL$_3$	BLK$_4$ ...
Kyle Lowry	Toronto	G	24	1	0 ...
Terrence Ross	Toronto	N/A	22	0	0 ...
Robert Covington	Philadelphia	G	20	2	0 ...
Jahlil Okafor	Philadelphia	C	15	0	1 ...

Note: N/A means that the statistic couldn't be collected because it is undefined (e.g. player didn't appear on starting roster therefore his starting position is undefined)
$_1$ Starting position ; $_2$ Points; $_3$ Steals; $_4$ Blocks

Table 2.2: An example of the player statistics from the development part of the Rotowire dataset

part of the dataset in figure 2.1. The non-highlighted text shows one-to-one correspondence with the structured data. The blue-highlighted text marks the information which is present in the input data only implicitly. However the level of implicitness varies. It is relatively easy to see that since Terrence Ross's starting position is "N/A", he must have started off the bench. On the other hand the fact that "The Raptors came into this game as a monster favorite" requires comparison of the winning-loosing records of both teams. The yellow-highlighted text labels the information that isn't deducible from the input data and the network would have to learn to hallucinate to be able to generate such text.

Somebody may argue that the observations do not sound favorable for the dataset. I see it as a big challenge and a possibility to apply advanced preprocessing as well as interesting architectural design of the neural networks. I will further elaborate on the issue of too noisy data in section 3.7.

TEAM	WIN	LOSS	PTS_1	FG_PCT_2	REB_3	AST_4 ...
Raptors	11	6	122	55	42	22
76ers	4	14	95	42	38	27

PLAYER	City	PTS_1	AST_4	REB_3	FG_5	FGA_6	S_POS_7 ...
Kyle Lowry	Toronto	24	8	4	7	9	G
Terrence Ross	Toronto	22	0	3	8	11	N/A
Robert Covington	Philadelphia	20	2	5	7	11	G
Jahlil Okafor	Philadelphia	15	0	5	7	14	C
DeMar DeRozan	Toronto	14	5	5	4	13	G
Jonas Valanciunas	Toronto	12	0	11	6	12	C
Ersan Ilyasova	Philadelphia	11	3	6	4	8	F
Sergio Rodriguez	Philadelphia	11	7	3	4	7	G
Richaun Holmes	Philadelphia	11	1	9	4	10	N/A
Nik Stauskas	Philadelphia	11	2	0	4	9	N/A
Joel Embiid	Philadelphia	N/A	N/A	N/A	N/A	N/A	N/A
...							

The host Toronto Raptors defeated the Philadelphia 76ers , 122 - 95 , at Air Canada Center on Monday . The Raptors came into this game as a monster favorite and they did n't leave any doubt with this result . Toronto just continuously piled it on , as they won each quarter by at least four points . The Raptors were lights - out shooting , as they went 55 percent from the field and 68 percent from three - point range . They also held the Sixers to just 42 percent from the field and dominated the defensive rebounding , 34 - 26 . Fastbreak points was a huge difference as well , with Toronto winning that battle , 21 - 6 . Philadelphia (4 - 14) had to play this game without Joel Embiid (rest) and they clearly did n't have enough to compete with a potent Raptors squad . Robert Covington had one of his best games of the season though , tallying 20 points , five rebounds , two assists and two steals on 7 - of - 11 shooting . Jahlil Okafor got the start for Embiid and finished with 15 points and five rebounds . Sergio Rodriguez , Ersan Ilyasova , Nik Stauskas and Richaun Holmes all finished with 11 points a piece . The Sixers will return to action on Wednesday , when they host the Sacramento Kings for their next game . Toronto (11 - 6) left very little doubt in this game who the more superior team is . Kyle Lowry carried the load for the Raptors , accumulating 24 points , four rebounds and eight assists . Terrence Ross was great off the bench , scoring 22 points on 8 - of - 11 shooting . DeMar DeRozan finished with 14 points , five rebounds and five assists . Jonas Valanciunas recorded a double - double , totaling 12 points and 11 rebounds . The Raptors next game will be on Wednesday , when they host the defensively - sound Memphis Grizzlies .

Note: $_1$ Points; $_2$ Field Goal Percentage; $_3$ Rebounds; $_4$ Assists; $_5$ Field Goals; $_6$ Field Goals Attempted; $_7$ Starting Position; N/A means undefined value

Figure 2.1: A data-point from the development part of the Rotowire dataset. Yellow-highlighted text isn't based on the input data. Blue-highlighted text implicitly follows from the input data.

3. Preprocessing and Statistics of the Dataset

To generate text from structured data I choose the Deep Neural Networks and specifically the Encoder-Decoder (ED) neural architecture (section 4). The ED suited to process sequential one dimensional data, however we cope with two dimensional tables. In this chapter I present the statistics of the dataset *before any preprocessing*. Next I elaborate about the methods of preprocessing and cleaning of the data. In the end I show the statistics of the dataset with all the transformations applied.

3.1 Transforming Tables to Records

At first let's define a table. A table is a two dimensional data structure, where the information is stored not only in the actual values in cells, but also in the positional information. Values in the same column have the same type, whereas values in the same row belong to the same entity. An example of a table as we have defined it is in figure 3.1.

	type_1	type_2 ...
entity_1	$\text{value}_{1,1}$	$\text{value}_{1,2}$ ...
entity_2	$\text{value}_{2,1}$	$\text{value}_{2,2}$ ...
...		

Table 3.1: An example of structured data

I use the same notation as [Liang et al., 2009]. Table $\mathcal{T}$ is transformed into a sequence of records $\mathbf{s} = \{r_i\}_{i=1}^{J}$, where r_i denotes i-th record. To fulfill our goal of keeping the most of the positional information from the table, each record contains field *r.type* denoting the type of the value, the actual value *r.value* and the entity *r.entity* to which the record belongs. At the end, I transform table 3.1 to sequence of records 3.1.

{*type*: type_1; *entity*: entity_1, *value*: $\text{value}_{1,1}$}

{*type*: type_2; *entity*: entity_1, *value*: $\text{value}_{1,2}$}

{*type*: type_1; *entity*: entity_2, *value*: $\text{value}_{2,1}$}

...

Figure 3.1: An example of records obtained by transforming table 3.1

3.2 Dataset Statistics

I believe that the challenges posed by the RotoWire dataset can be summarized in a set of statistics. In this section I want to present the most important ones to help reader to understand the nature of the problem.

Firstly, the target summaries as well as the sequences of input records are really long compared to other datasets modelling the same task (e.g. WikiBIO [Lebret et al., 2016], WeatherGOV [Liang et al., 2009], RoboCup [Chen and Mooney, 2008]).

Secondly, many words occur rarely and the generation system cannot learn a good representation of them (it is known as a *rare word problem*). It should be noted that the problem can be resolved by the means of clever preprocessing (section 3.6) or with help of advanced neural architectures (section 4.3).

Thirdly, there are many values which represent facts, e.g. values that cannot be deduced from the context (e.g. points a player scored in a match etc.) but must be selected and copied from the table.

The original dataset as prepared by [Wiseman et al., 2017], is already divided to train (3398 samples), development (727 samples) and test (728 samples) sets. *In the statistics presented below I state that there are only 3397 samples in the train set because one of the samples is the famous Lorem Ipsum template.*

3.2.1 Length-wise Statistics

The input tables contain huge amount of information. 2 teams and up to 30 players participate in a match of basketball. After transformation to a sequence, a player is represented by 24 and a team by 15 records. The type field *r.type* is the only trait distinguishing the team and player records. Table 3.2 summarizes the length statistics of the input sequences.

Set	Max Number of Records	Min Number of Records	Avegage Number of Records	Size
train	750	558	644.65	3397
development	702	582	644.66	727
test	702	558	645.03	728

Table 3.2: Statistics of tables as used by [Wiseman et al., 2017]

There is much greater variance in the lengths of output summaries. The longest sequence of input records is 1.34 - times longer than the shortest one, while the factor between longest and shortest summaries is more than 5. The size of inputs and outputs places high memory and computation demands on the GPUs used for training, and needs a special treatment (as will be explained in section 5.1).

3.2.2 Occurrences of Unique Tokens

While the length of the inputs and the outputs increases computational demands, another common issue is the *rare word problem*. If a token appears only sporadi-

Set	Max Summary Length	Min Summary Length	Avegage Summary Length	Size
train	762	149	334.41	3397
validation	813	154	339.97	727
test	782	149	346.83	728

Table 3.3: Statistics of summaries as used by [Wiseman et al., 2017]

cally in the train data, the generation system can't recognize how to use it. After discussions with my advisor I think it is reasonable to expect that the system should learn a good representation of a token if it appears at least 5 times in the train set.

There is about 11 300 unique tokens in the dataset (in the union of train, development and test set). In table 3.4 I present the statistics regarding the occurrences of the tokens. We can see that only 42 % of all the tokens appear at least 5 times in the train part of the dataset.

However I expect that even if some anomaly in the real word happens (e.g. team scores 200 points, although at the time of writing no team in the history of NBA scored more than 186) the system should be able to simply *copy* the value of *TEAM-PTS* record without reasoning about the actual value. Consequently we are interested in tokens that cannot be copied from the table. Since most of the named entities are directly copiable, there is no need to preserve casing. All the aforementioned statistics are summarized in table 3.4.

In the end we see that under our assumptions about 60 % of all the unique tokens cannot be learned by the generation system.

Set	Unique Tokens	>= 5 Absolute	>= 5 Relative
train	9779	4158	42.52%
train_wop$_1$	8604	3296	38.31%
train_wopl$_2$	8031	3119	38.84%

Note: $_1$ train_wop is training set with all the player names, city names, team names and numbers extracted $_2$ train_wopl is train_wop lowercased

Table 3.4: Occurrences of tokens in summaries from dataset RotoWire

In table 3.5 we can see how many of the unique tokens learned during training can be found in the respective development and test datasets. Under our assumptions we can expect the generated text to share less than 65 % of the vocabulary with the gold references.

3.3 Transformations of Input Tables

Firstly I want to present what kind of data is stored in the input tables, and how it is preprocessed. After that I show the final format of a record which is fed to the generation system.

Set	Unique Tokens	Train Overlap	Train$_{>=5}$ Overlap
valid	5625	88.18%	66.63%
test	5741	87.46%	65.72%
valid_wop$_1$	4714	86.36%	61.92%
test_wop$_2$	4803	86.03%	61.13%
valid_wopl$_3$	4442	86.74%	62.36%
test_wopl$_4$	4531	86.32%	61.37%

Note: train$_{>=5}$ overlap is a set of all the tokens from the development/test dataset with more than 5 occurrences in the train dataset summaries $_1$, $_2$, $_3$, $_4$ have the same meaning as in table 3.4

Table 3.5: Overlap of train dataset summaries and valid/test dataset summaries

3.3.1 Tabular Types

There are 39 types (different headers of columns as discussed in section 3.1). A type is associated to textual or integer value which describes either a team or an individual. There are only 7 types bound to textual values, out of which 2 are related to teams (*TEAM-NAME*, *TEAM-CITY*) and 5 to individuals (*FIRST_NAME**, *SECOND_NAME**, *PLAYER_NAME**, *START_POSITION**, *TEAM_CITY**)

3.3.2 Numerical values

The other 32 types desribe absolute (*TEAM-PTS*[1], *FTM*[2] ...) or relative integer values (*TEAM-FT_PCT*, *FT_PCT*[3], ...). During preprocessing no changes are made to any tabular numerical value[4].

3.3.3 Textual values

Regarding the textual values, I consider each as a single token. Since the names of teams are already one word long (with one exception needing a transformation *Trail Blazers* $\rightarrow$ *Trail_Blazers*) the transformation is rather trivial. The similar observation applies to names of cities (with 6 exceptions: *Oklahoma_City*, *San_Antonio*, *New_Orleans*, *Los_Angeles*, *Golden_State*, *New_York*), and start positions.

Out of three types connected to player credentials, only *PLAYER_NAME* (describing player full name with all the attributes e.g. *Johnny O'Bryant III*) is multi-token.

The original take on the problem is different to mine. The authors of the dataset [Wiseman et al., 2017], as well as the authors of one of the more successful approaches to the task [Puduppully et al., 2019] make use of three special

[1]Team Points

[2]Number of converted free throws by an individual, *"free throws made"*

[3]*team/player free throw percentage*

[4][Wiseman et al., 2017] already converted all the relative values to integers. I don't consider this as a preprocessing since only the converted values are available in the dataset.

types, *PLAYER_NAME*, *FIRST_NAME* and *SECOND_NAME* which allow to distinguish if *James* refers to the first name of star player *James Harden* or to a second name of the legend of *LeBron James*.

My approach is based on the idea that more dense data leads to easier learning path for the generation system. Therefore I transform the name of each player to a single token. To make copying possible a preprocessing of the output summaries as described in 3.4.2 must take place.

Consequently *FIRST_NAME* and *SECOND_NAME* records aren't needed anymore which results in another benefit, shorter inputs. (as there are 22 - 30 players involved in the match, the size of inputs becomes about 10 % (44 - 60 records) shorter)

3.3.4 Entities

The type information tells us what represents the number or text in the value field. However it is the entity field which brings together all the records describing the same player or team. Let's show an example of records about a star player, *Stephen Curry*. His name is stored in a record of type *PLAYER_NAME*, and value *Stephen_Curry*. To link all the information about him together, each record has an entity field labelled *Stephen_Curry*. Similarly all the records of a team with *TEAM_NAME* : ***A*** have the same entity field, ***A***.

At last, we should notice that the overall team information is the union of the accumulated team stats (e.g. the number of points scored by all the players of a team) and the collection of statistics of the individuals playing for the team. Therefore the record also contains *HOME/AWAY* field which brings together all the statistics about the home side and the away side.

3.3.5 Record Format

The records fed into the generation system contain the following fields:

- *Type*
- *Value*
- *Entity*
- *Home/Away flag*

The generation system should be able to understand the meaning of a record and shouldn't rely on a specific organization of the table. This is modelled by emplacing the team records at the end, so that the system will need to search for the team statistics. Since the size of the input sequence isn't uniform, the team records can start anywhere between 500th and 720th record. The organization of the input sequence will be further explained in the chapter about experiments 6.

3.4 Preprocessing of Summaries

I would like to reiterate that our motivation is to avoid the *rare word problem* and to make copying words from the sequences of input records as easy as possible.

{*type*: PTS; *entity*: Stephen_Curry, *value*: 25; *ha*: HOME }

{*type*: TEAM-PTS; *entity*: Warriors, *value*: 122; *ha*: HOME}

...

Figure 3.2: An example of a player and a team record.

Therefore we opt for methods which reduce the number of tokens, increase their average frequency (because the system couldn't learn the most sporadic ones anyway), and transform the tokens describing the tabular data to the same form as is used in the table (so copying is trivial).

3.4.1 Number Transformations

Just as [Wiseman et al., 2017] and [Puduppully et al., 2019], we represent the numbers only by numerals. This preprocessing method partially fulfills both of our goals. Obviously it decreases the unique token count, but on top of that it makes copying easier. E. g. the sentence *"Isaiah Thomas once again excelled , scoring 23 points,* ***three*** *assists and* ***three*** *rebounds."* is transformed to *"Isaiah Thomas once again excelled , scoring 23 points,* ***3*** *assists and* ***3*** *rebounds."*. Under this setting, the network still has to learn the correspondence between record type and the summary token *"AST"* $\cong$ *"assists"* but without the need of linking *"three"* to *"3"* the connection of the phrase with record {*AST; 3; Isaiah_Thomas; Home*} should be much clearer. However we preserve the word *"three"* when it forms a part of a basketball terminology (e.g. *three pointer*) to differentiate between these different meanings of the word. The transformations are done with the help of the *text2num* library[5] which is also used by the authors cited above.

3.4.2 Player name transformations

The generation system should be able to create a summary of player's actions in the game based on the records describing his match-statistics. It is common that at first a player is mentioned by his full name (e.g. *Stephen Curry*) and after that only by his second name (*Curry*). Also more than 97 % of all the players have exactly 2 names (first, last). This leaves out 17 players with longer names the most extreme case being *Luc Richard Mbah a Moute*, who is represented in the whole dataset by 6 different combinations of ellipsis in his name.

Since only the full name concatenated to a single token is contained in the input records I developed an algorithm which transforms all [6] the references to a player to that specific token.

I haven't measured the accuracy of the algorithm, however it passes an eye-test as during the development of neural models I have inspected a great amount of produced summaries which haven't contained any discrepancies.

[5] `https://github.com/allo-media/text2num`

[6] Although technically speaking this is not true as It doesn't transform any pronouns and the transformations follow simple path: *some part of a name* $\rightarrow$ *full name*

At first we gather all the player names from the input tables. The transformation then happens in three steps, which are described on the example sentence from figure 3.3.:

- **1. Extraction of player names from the summary**
  The summary is at first divided to sentences, using NLTK [7] library. Then we traverse each sentence and extract the longest subsequences of tokens which appear in the set of the player names. This way, one-token name *James* and two-token name *James Harden* is extracted. (Although *James Harden* hasn't played in the game and therefore the network cannot learn to copy his name, we extract it to densify the data, so the player is represented by the same token in all the summaries)

- **2. Resolution of one-token references and creation of transformation dictionary**
  James Harden is a two-token name matched in the first phase, so we assume that it is already full name of the player and we add a trivial transformation *James Harden* → *James_Harden*. *James* is a one-token name which needs resolution. At first we look if anyone whose second (third ...) name is *James* hasn't already been mentioned in the summary. If not we proceed to searching through all the players in the match statistics. There we spot *LeBron James* and add the transformation *James* → *LeBron_James*. Note that we create a unique transformation dictionary for each summary and we assume, that no player is called only by his first name.

- **3. Application of transformations**
  The summary is traversed for the second time and the longest subsequences appearing in the transformation dictionary are substitued.

While King James struggled , James Harden was busy putting up a triple - double on the Detroit Pistons on Friday.

↓

While King LeBron_James struggled , James_Harden was busy putting up a triple - double on the Detroit Pistons on Friday.

Figure 3.3: Example of transformation of player names leveraging the knowledge of players on the rosters as well as of all players from the train set.

3.5 Vocabularies

During preprocessing we collect the vocabulary of all the words from the training set. Each token is represented as an index to the vocab. This representation is fed to the initial layers of the neural network (embedding layers which will be discussed in section 4). Therefore the network learns to process only the tokens belonging to the vocabulary and no new tokens can be introduced during inference.

[7] `https://www.nltk.org/`

We collect 3 different vocabularies, one for record types, one for home/away flags and one for all the entities, number values and tokens from the summaries.

3.6 Byte Pair Encoding

The Byte Pair Encoding (BPE) [Sennrich et al., 2016] is a method of preprocessing of a text. It was developed to enable the Neural Machine Translation models to operate on subword units. It allows them to generate and accept sequences of subwords and thus handle words unseen during training. (E.g. there are words 'high', 'low', 'lower' in the training dataset, therefore under regular setting the network couldn't be able to generate or process word 'higher'. When operating on subwords 'high', 'low', 'er' this isn't an issue anymore since the network can learn to chain subwords 'high', 'er' to form the word unseen during training.)

Since in NBA there is a fixed set of cities, teams and players (only about 10 players from development and 10 from test set were unknown from training) this isn't our main concern. However we can use the BPE to lower the size of the summary token vocabulary (E.g. looking at the previous example, even if the word 'higher' had been a part of the input vocabulary, the vocabulary of subwords would have still contained only subwords 'high', 'low', 'er'.) Thus the average frequency of a token increases as well as the generational capacity of the network.

In this section I begin with a short explanation of the algorithm and conclude with the statistics of the fully transformed dataset.

3.6.1 Algorithm

I use the implementation of the algorithm from the authors of the BPE paper[8], which is also downloadable as a standalone python package. The idea is to divide each token from the train set to characters, and add them to the *symbol vocabulary*. Iterating over the text each time we merge together the most common pair of succeeding characters to create a new symbol. The symbol is added to the vocabulary and each occurrence of the pair is substitued by it. At the end (after N iterations, where N is the hyperparameter of the algorithm), the symbol vocabulary contains all the characters and newly created symbols.

In practice it is more efficient to create a token vocabulary (tokens are weighted by the number of occurrences in the corpus), and iterate over it instead of over the whole corpus. Also to allow easy detokenization to the original text, a special $<eow>$ token is appended to each token. An excerpt from the original paper shows the minimal possible implementation of such approach 3.6.

At the test time the text is divided to characters and the longest subsequences of characters appearing in the symbol vocabulary are transformed to the corresponding symbol.

[8] `https://github.com/rsennrich/subword-nmt`

3.6.2 Application

As stated previously, the BPE should transform the output summaries to a sequence of subwords. However, in certain situations it may be contraproductive. E.g. it may make copying harder by dividing single-token player name to multiple tokens. Therefore I apply the BPE to all the tokens except the ones which correspond to player/city/team names and numerical values. We set the number of iterations to 2000, which means that the overall vocabulary contains around 2800 tokens (there is about 700 players, 29 cities and 30 teams). An example of the final appearance of a summary after all the transformations mentioned in the chapter can be seen in figure 3.4.

the host Toronto Raptors defeated the Philadelphia 76ers , 122 - 95 , at air canada center on monday . the Raptors came into this game as a monster fav⋆ or⋆ ite and they did n't le⋆ ave any doub⋆ t with this result . Toronto just continu⋆ ou⋆ s⋆ ly pi⋆ led it on , as they won each quarter by at least 4 points . the Raptors were l⋆ ights - out shooting , as they went 55 percent from the field and 68 percent from three - point range . they also held the sixers to just 42 percent from the field and dominated the defensive rebounding , 34 - 26 . fas⋆ t⋆ break points was a huge difference as well , with Toronto winning that battle , 21 - 6 . Philadelphia (4 - 14) had to play this game without Joel_Embiid (rest) and they cle⋆ arly did n't have enough to compe⋆ te with a poten⋆ t Raptors squad . Robert_Covington had 1 of his best games of the season though , tallying 20 points , 5 rebounds , 2 assists and 2 steals on 7 - of - 11 shooting . ...

Figure 3.4: A part of transformed summary corresponding to the sample from figure 2.1. ⋆ is used to mark that the following token formed the same word in the original text. Note that in the original implementation @@ is used instead.

3.7 Cleaning

During implementation of the Content Planning approach (discussed in section 5.4.2) I found out that the authors of the method, [Puduppully et al., 2019], have used only subset of training and validation data. They removed all the pairs summary-table where the "gold" summary contained information which wasn't linked to the input table. Figure 3.5 shows an example of such a summary.

I use a subset of the subset used by Puduppully. In addition I also removed all the samples from the dataset about matches between teams from Los Angeles (Clippers and Lakers) where the distinction between different teams wasn't shown, and every player was listed as playing for "Los Angeles" (thus it was impossible to tell if he played for Clippers or Lakers).

After removing all the non-valid pairs, the dataset size was reduced to 3369 train, 721 development and 727 test samples.

There exist datasets based on RotoWire, which contain cleaner data and summaries corresponding better to input tables [Wang, 2019], [Thomson et al., 2020]. However I choose to continue with the RotoWire dataset, as I am already accustomed to the format of the data.

Following a week filled with trade rumors , Paul George came out of the All-Star break in fairly unimpressive fashion . In his first 4 games following the break , Paul George shot a combined 16 - of - 54 (29 percent) from the field and was averaging just 14 points per game over that stretch . However , in his last 2 games , Paul George has flipped the script entirely and rattled off a pair of incredible offensive performances . Following Monday 's 36 - point performance in the Pacers ' loss to the Hornets , Paul George has now scored a combined 70 points over his last 2 games and did so while shooting a scorching 27 - of - 44 (61 percent) from the field and 12 - of - 23 (52 percent) from behind the arc . The performances from Paul George on Sunday and Monday were by far his best back - to - back shooting and scoring performances of the season .

Figure 3.5: An example of a faulty summary. To illustrate how hard it is to tell even which teams played I purposely don't show the input table.

3.8 Statistics of Transformed Dataset

In this section I want to summarize all the transformations applied to the summaries and tables and present the statistics of the summaries after transformations.

At first we converted all the tokens in the value and entity fields of a record to a single token. Next we transformed all the numerical values in the summaries to numerals and all the player names to a single token to allow direct copying from the input records. At the end we applied the Byte Pair Encoding to all the remaining tokens to decrease the overall number of tokens and increase the average frequency of a token.

In table 3.6 we can observe that almost 90 % of all the tokens occur more than 5 times therefore we can conclude that data is definitely much denser (compared to 42 % in the original data 3.4). As a nice side effect the intersection of development/test set and train set is almost 99 % (table 3.7). It is clear from figure 3.4 that each factual information is represented by a single token and can be copied as is from the *r.value* field of a record.

Set	**Unique Tokens**	$>= 5$ **Absolute**	$>= 5$ **Relative**
train	2839	2531	89.15%

Table 3.6: Occurrences of tokens in transformed summaries from dataset RotoWire

Set	**Unique Tokens**	**Train Overlap**	**Train**$_{>=5}$ **Overlap**
valid	2582	98.80%	95.70%
test	5741	98.69%	95.45%

Table 3.7: Overlap of transformed train dataset summaries and valid/test dataset summaries

```python
import re, collections

def get_stats(vocab):
    pairs = collections.defaultdict(int)
    for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols)-1):
        pairs[symbols[i],symbols[i+1]] += freq
    return pairs

def merge_vocab(pair, v_in):
    v_out = {}
    bigram = re.escape(' '.join(pair))
    p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
    for word in v_in:
        w_out = p.sub(''.join(pair), word)
        v_out[w_out] = v_in[word]
    return v_out

vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
          'n e w e s t </w>':6, 'w i d e s t </w>':3}
num_merges = 10
for i in range(num_merges):
    pairs = get_stats(vocab)
    best = max(pairs, key=pairs.get)
    vocab = merge_vocab(best, vocab)
    print(best)
```

OUTPUT:

r · → r·
l o → lo
lo w → low
e r · → er·

Figure 3.6: Python code extracted from paper **Neural Machine Translation of Rare Words with Subword Units** by [Sennrich et al., 2016]
Output represents the learned merge operations.

4. Neural Network Architectures

To generate text from structured data I make use of deep neural networks. In previous chapter (3) I've shown how to transform the structured tabular data into a sequence of records, therefore I've reduced the problem to an instance of the famous sequence to sequence problem. Now, I show how to create a system that transforms the input sequential data (structured records) to the output sequential data (natural language).

The most common way to tackle the sequence to sequence problem is to use the Encoder-Decoder architecture proposed by [Sutskever et al., 2014]. It is the main approach I used throughout this thesis. In this chapter I introduce the concepts behind the encoder-decoder architecture, its shortcomings (fixed vocabulary and thus problems with generation of words unseen during training, divergence and hallucinations) and ways to overcome these shortcomings (the attention mechanism, the copy mechanisms, the further transformations of input sequences). Since it is not the purpose of this work, only the basics of the concepts are presented, and I provide links to papers, books and tutorials which helped me on my path to understanding.

Notation

Many papers diverge on the notation and naming conventions of the architectures. Therefore I choosed to adopt the notation used in *Tensorflow Keras API, version 2.x* [Abadi et al., 2015]. Specifically in the field of recurrencies it is discutable if the paper refers to *tf.keras.layers.RNNCell* or to *tf.keras.layers.RNN*. I believe that they can be used interchangeably in the context of this chapter, hence I (deliberately) choose the latter notation (*without 'Cell'*).

A Note About Embeddings

An embedding is a low dimensional continuous representation of discrete variables[1]. In this work I don't experiment with pretrained word embeddings, and I tune only the embedding-dimension hyperparameter, which adjusts the dimensionality of the trained continuous representation of the input.

4.1 The Encoder-Decoder Architecture

Proposed by [Sutskever et al., 2014] the Encoder-Decoder is composed of 2 recurrent units, called Encoder and Decoder. In this section I briefly introduce the Recurrent Neural Network (*tf.keras.layers.SimpleRNN*), its modification, the Long Short-Term Memory (*tf.keras.layers.LSTM*) [Hochreiter and Schmidhuber, 1997] and the high-level overview of the Encoder-Decoder architecture.

[1] I took the sentence from the accepted answer on stack overflow as it is the best description I have found. It is the first and the last time I "cite" from stack overflow. Source : `https://datascience.stackexchange.com/questions/53995/what-does-embedding-mean-in-machine-learning`

4.1.1 Recurrent Neural Network

Let $\boldsymbol{x} = (x^{(1)}, \ldots, x^{(t)})$ be the input. The standard Feed-Forward Network (*tf.keras.layers.Dense*) has a different set of weights for each input time-step $x^{(t)}$, therefore the number of time-steps of the input needs to be known in advance.

The Feed-Forward Neural Network

$$\boldsymbol{y} = activation(W\boldsymbol{x} + b) \tag{4.1}$$

On the contrary, the Recurrent Neural Network (RNN) [Rumelhart et al., 1988] (*tf.keras.layers.SimpleRNN*) shares the same set of weights between time-steps and in addition it keeps a hidden state. At each time-step the hidden state is updated and used to calculate the output as in equation 4.2. The computation can be visualized either as a loop, or as a feed-forward network with shared weights 4.1.

The Recurrent Neural Network

$$\begin{aligned} h_t &= f_h(x_t, h_{t-1}) \\ y_t &= f_t(h_t) \end{aligned} \tag{4.2}$$

Note: h_t is the hidden state; y_t is the output at t-th timestep; tf.keras.layers.SimpleRNN uses $f_t = id$ as default.

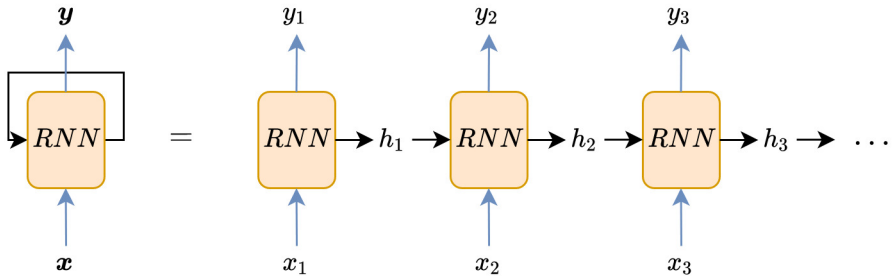


Figure 4.1: Visualizations of the RNN

The network is trained by back-propagation through time (BPPT) [Werbos, 1990]. It has been shown that the RNN suffers from vanishing / exploding gradient problems [Hochreiter and Schmidhuber, 1997].[2] Which cause that either the RNN cannot learn anything or it is really unstable. These difficulties are adressed by more sophisticated architectures such as Gated Recurrent Unit[3] [Cho et al., 2014b] or Long Short-Term Memory [Hochreiter and Schmidhuber, 1997].

4.1.2 Long Short-Term Memory

The Long Short-Term Memory (*tf.keras.layers.LSTM*) addresses the vanishing gradient problem. It does so by adding a special cell state for capturing long

[2] I believe that discussion about BPPT and exploding/vanishing gradient problems is beyond the scope of this work. Therefore I refer the reader craving for further explanation to [Goodfellow et al., 2016] and to referenced papers.

[3] Although it is one of the most known architectures I used only LSTM for my experiments.

range context and series of gating mechanisms. The latter update the cell state and regulate the flow of gradient through the network (as shown in figure 4.2). The more in-depth explanation can be found in [Olah, 2015].

$$y_t = W_{hy}h_t + b_y \tag{4.3}$$
$$h_t = o_t tanh(c_t) \tag{4.4}$$
$$o_t = \sigma(W_o[h_{t-1}; x_t] + b_o) \tag{4.5}$$
$$c_t = f_t * c_{t-1} + i_t * tanh(W_c[h_{t-1}; x_t] + b_c) \tag{4.6}$$
$$i_t = \sigma(W_i[h_{t-1}; x_t] + b_i) \tag{4.7}$$
$$f_t = \sigma(W_f[h_{t-1}; x_t] + b_f) \tag{4.8}$$

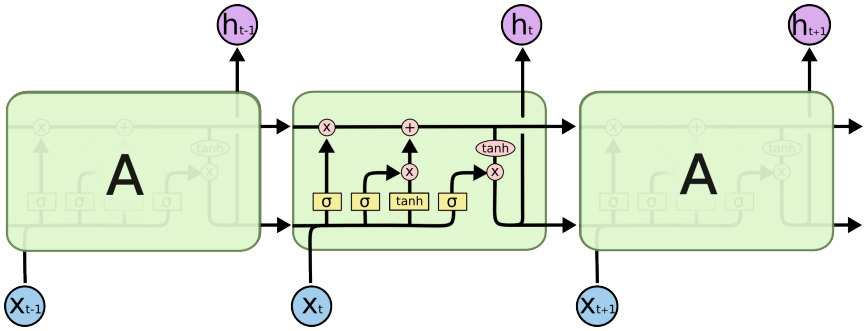


Figure 4.2: Visualization of LSTM [Olah, 2015]

4.1.3 High-level Overview of Encoder-Decoder Architecture

As stated by [Sutskever et al., 2014], the main goal of the encoder-decoder architecture is to model the conditional probability $p(y_1, \ldots, y_n|x_1, \ldots, x_m)$ of the output sequence $y_1, \ldots, y_n$ conditioned on the input sequence $x_1, \ldots, x_m$. It uses two separate *recurrent networks*[4]. The first, called Encoder, produces a fixed-dimensional representation r of the input sequence. r is then used to initialize the hidden states of the second recurrent network, called a Decoder. The Decoder then generates the output sequence as in equation 4.9.

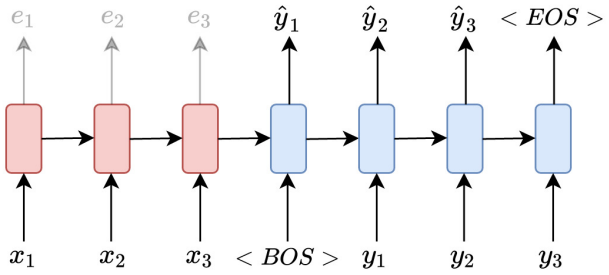
$$p(y_1, \ldots, y_n|x_1, \ldots, x_m) = \prod_{t=1}^{n} p(y_t|r, y_1, \ldots, y_{t-1}) \tag{4.9}$$

The Encoder part is straight-forward, however in the Decoder part two important questions arise:

1. What should be the inputs to the Decoder
2. When should we stop decoding (generating the output sequence)

[4]Here I refer to *recurrent network* as to a complex consisting of at least one RNN/LSTM/-GRU/... rather than to a single recurrent layer.

As visualized in figure 4.3, to tell the Decoder when to *start* generation, a special $<BOS>$ token (*beginning of sequence*) is fed in. The decoding phase is slightly different during training and inference. During training, the inputs of the decoder are targets from the *previous* time-step. This process is called *teacher forcing*. After the last time-step, the target is the special $<EOS>$ token (*end of sequence*). In the inference phase, the last output of the Decoder is fed in as the input, and the decoding finishes after producing the $<EOS>$ token.



Note: Encoder is red, Decoder is blue
none of the Encoder's outputs is used

Figure 4.3: Visualization of the training of the Encoder-Decoder Architecture with *teacher forcing*.

4.1.4 Problems of the Encoder-Decoder Architecture

Despite having many advantages (variable length of the input and output sequence, possibility of extending the number of recurrent layers in the encoder and the decoder) there are some major flaws that need to be overcome in order to generate text from structured data.

Fixed-dimensional Representation of the Input Sequence

It has been shown by [Cho et al., 2014a] that the performance of the Encoder-Decoder architecture "suffers significantly from the length of sentences". [Bahdanau et al., 2014] hypothesize that it may be because all of the information from the source sequence is encoded to the fixed-dimensional vector. Both mentioned papers understand word *"long"* as *longer than 30 tokens*. From the chapter about the preprocessing 3, we know that there are more than 300 records in the average input from the RotoWire dataset. Consequently this particular problem should be seen in our task.

Rare-word Problem and Hallucinations

In the standard Encoder-Decoder, the output is a distribution over the output vocabulary. At the start of the training each word is equally probable in any given context (assuming reasonable weights initialization). During training, model learns the language model on the training data. There are several flaws in the design:

1. It essentially means that e.g. words 'the' and 'Roberta' compete against each other, although one depends purely on the language skill (perhaps the next token after 'the' would be superlative) and the other one on the input sequence (which probably mentions some AI research).

2. As pointed out by e.g. [Gulcehre et al., 2016], (although not on this particular example) word 'Roberta' occurs less frequently in the training data than the word 'the', thus it is "difficult to learn a good representation of the word, which results in poor performance"[5]

3. Networks tend to *hallucinate* the facts. E.g. from the record {*type:transformer; value:GPT*} network generates a sentence "The famous example of transformer architecture is BERT." To put it simply, the network knows it should talk mention a transformer, therefore each word describing some kind of a transformer is somewhat probable, even if it wasn't seen in the actual input.

I have already discussed how to increase the average frequency of a token to minimize the Rare-Word Problem through preprocessing (section 3.6). In the following sections I show the methods which handle hallucinations (section 4.3).

4.2 Attention Mechanism

The Attention mechanism should cope with the issue of fixed-dimensional representation of the input sequence 4.1.4. As stated by [Bahdanau et al., 2014] "The most important distinguishing feature of this approach from the basic encoder-decoder is that it does not attempt to encode a whole input sequence into a single fixed-length vector".

The encoder is a recurrent neural network[6]. From the overall architecture (figure 4.3) we can see that only the last hidden state of the encoder is used, although there are encoder outputs for each time-step. [Bahdanau et al., 2014] propose an architecture which takes advantage of this simple observation. In my work I use a little refinement, proposed by [Luong et al., 2015].[7]

4.2.1 Computation

Let's start with the description of the computation that produces the attention output.

As stated previously, the Encoder encodes the input sequence to *encoder outputs* $\boldsymbol{e} = (e_1, \ldots, e_m)$ and the Decoder RNN is initialized with the last hidden state of the Encoder.

[5]The problem is called *The Rare-Word Problem.*

[6]From now on, I'll stick to refer to *recurrent neural network* as to the neural network consisting of at least one *tf.keras.layers.RNN* or relatives.

[7]Since I don't experiment with the original Bahdanau attention, I only show the Luong's approach. [Luong et al., 2015] shows all the differences between his and Bahdanau's approach in section 3.1 of the paper.

Let d_t be the output of the Decoder RNN at t-th time-step. At first we calculate the *score vector* $\boldsymbol{s_t} = (s_{t,1}, \ldots, s_{t,m})$. Its elements are computed using *a score function* (which we will talk about below 4.2.2):

$$s_{t,i} = score(e_i, d_t) \tag{4.10}$$

According to [Bahdanau et al., 2014], the alignment vector $\boldsymbol{a_t}$

$$\boldsymbol{a_t} = softmax(\boldsymbol{s_t}) \tag{4.11}$$

"scores how well the inputs around position i and the output at position t match". The weighted sum of the outputs of the encoder, is called a *context vector* for time-step t.

$$c_t = \sum_{i=1}^{m} a_{t,i} e_i \tag{4.12}$$

Unlike in the standard Encoder-Decoder architecture, the output of the decoder also depends on the context vector.

$$att_t = tanh(W_c[c_t; d_t]) \tag{4.13}$$

$$p(y_t | y_{<t}, x) = softmax(W_y att_t) \tag{4.14}$$

4.2.2 Score Functions and Input Feeding

[Luong et al., 2015] experimented with three different types of score functions. We adopted two of them, the *dot* and *concat* ones. (The following equations are directly extracted from the Luong's paper)

$$score(e_i, d_t) = \begin{cases} e_i^\top d_t & dot \\ v_s^\top tanh(W_s[e_i; d_t]) & concat \end{cases}$$

The same author also states that the fact that the attentional decisions are made independently is *suboptimal*. Hence the *Input Feeding* approach is proposed to allow the model to take into acount its previous decisions. It simply means that the next input is the concatenation $[y_t, att_t]$ (as shown in figure 4.4).

4.3 Copy mechanism

The copy mechanism is a further extension of the attention mechanism. In this section I discuss the Pointer networks [Vinyals et al., 2015], which are trained to *point* to some position in the input sequence and the Copy Mechanisms [Gulcehre et al., 2016], [Gu et al., 2016], [Yang et al., 2016] which model the decision making (whether to copy from the pointed location or to generate from the actual context).

4.3.1 Pointer Networks

The Pointer networks [Vinyals et al., 2015] leverage the fact that the alignment vector $\boldsymbol{a_t}$ can be seen as *a pointer* to the input sequence. Consequently instead of computing the weighted sum (*context vector*) and an MLP on top of that as the Attention models, they utilize the *alignment vector* as an output.

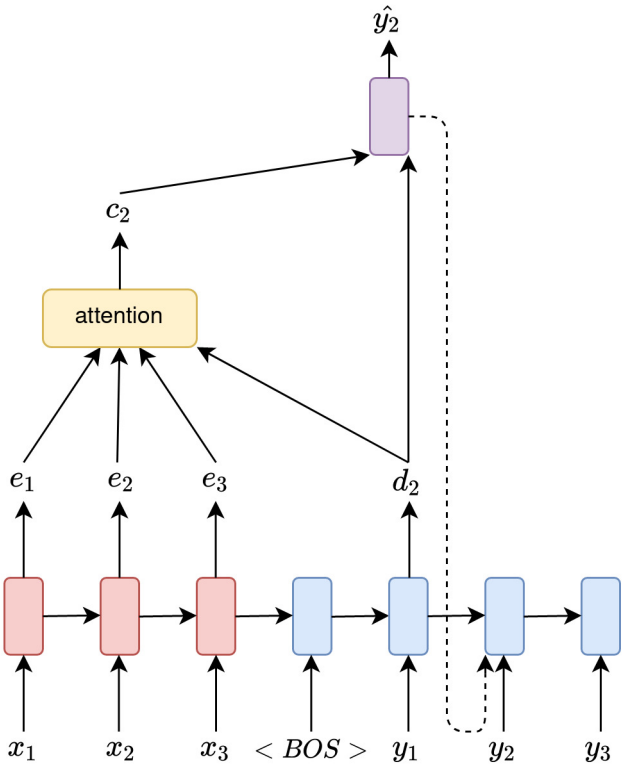


Figure 4.4: The Attention mechanism at the second time-step. Dotted line represents the input feeding approach.

4.3.2 Approaches to Copying

[Gulcehre et al., 2016] note that the ability to point is useless in the generation task if the network is always forced to point. Therefore they introduce a new switching network that outputs a binary variable z_t , which models the probability of the required action being pointing or generating.

Let $\boldsymbol{e} = (e_1, \ldots, e_m)$ be the encoder outputs, $\boldsymbol{x} = (x_1, \ldots, x_m)$ the input sequence, d_t the output of the Decoder RNN at the actual time-step, and $ATTN$ the Attention as presented in the previous section.

The probability of gold output y_t and gold switch decision z_t is decomposed to $p^{gen}(y_t|\boldsymbol{x})$ (the probability that y_t should be generated), $p^{switch}(z_t|\boldsymbol{x})$ (the probability that we should copy) and $p^{copy\_pos}(y_t = x_i|\boldsymbol{x})$ (the probability that y_t should be copied from the input position i).:

$$\boldsymbol{a_t} = ATTN(\boldsymbol{e}, d_t) \tag{4.15}$$

$$p^{copy\_pos}(y_t = x_i|\boldsymbol{x}) = a_{t,i} \tag{4.16}$$

$$\boldsymbol{c_t} = \sum_{i=1}^{m} e_i * a_{t,i} \tag{4.17}$$

$$p^{switch}(z_t|\boldsymbol{s}) = sigmoid(W_{switch}[\boldsymbol{c_t}, d_t]) \tag{4.18}$$

$$p^{gen}(y_t|\boldsymbol{s}) = softmax(W_{gen}[\boldsymbol{c_t}, d_t]) \tag{4.19}$$

[Gulcehre et al., 2016] explicitly model each of these probabilities (therefore the targets contain 3 values for each time-step). [Yang et al., 2016] marginalize out the switch probability z_t, and they model $p = p^{copy} * p^{switch} + p^{gen} * (1 - p^{switch})$.

To be able to follow their path, I take the one-hot encoding of each input and compute the weighted sum:

$$p^{copy}(y_t|\boldsymbol{x}) = \sum_{i=1}^{m} p^{copy\_pos}(y_t = x_i|\boldsymbol{x}) * x_i \tag{4.20}$$

Throughout our task the input and output vocabularies are shared, therefore the weighted sum of the inputs has the same dimensionality as the output generation distribution p^{gen}.

Consequently, the probability of the gold output y_t at time-step t is computed as follows:

$$p(y_t|\boldsymbol{x}) = p^{gen}(y_t|\boldsymbol{x}) * (1 - p^{switch}(1|\boldsymbol{x})) + p^{copy}(y_t|\boldsymbol{x}) * p^{switch}(1|\boldsymbol{x}) \tag{4.21}$$

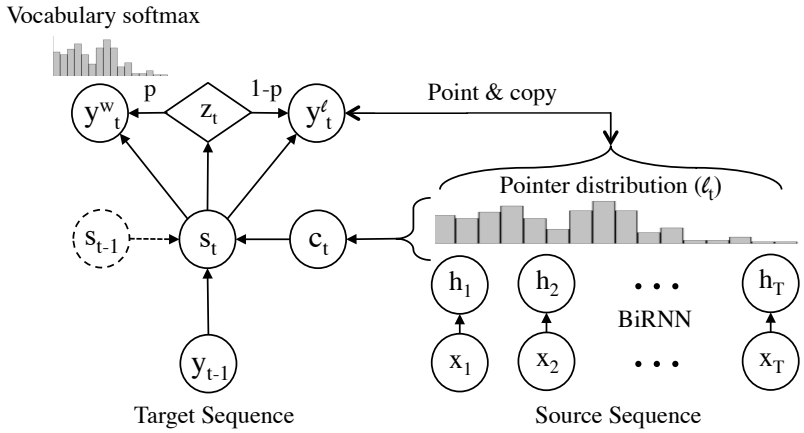


Figure 4.5: Excerpt from **Pointing the Unknown Words** paper by [Gulcehre et al., 2016], showing how the attention alignment can be utilized as a pointer information

5. Experimental Setup

In this chapter I aim to clarify how I set up the neural network models for the experiments (which will be discussed in the next chapter 6). At first I present a *baseline model*. Model, which is sufficiently competitive to be able to generate reasonable texts. Then I present changes to the architecture, which aim to improve the generation.

I follow the path set up by [Wiseman et al., 2017]. I begin with purely end-to-end approach (Encoder-Decoder with attention).Next I try to improve it with the copy mechanism 4.3. Next I choose to divide the task to *content planning* and *text generation* as proposed by [Puduppully et al., 2019].

5.1 Truncated Backpropagation Through Time

It was really challenging to set up even the baseline model. Since the input sequences are about 600 records long and output sequences are more than 300 tokens in average (and the outputs are padded with $<PAD>$ token to approximately 800 tokens), it wasn't possible to fit the model into the GPU memory. (The GPUs at `https://aic.ufal.mff.cuni.cz` have about 8GBs of memory[1])

Computing and updating gradient for each time-step of a sequence of 800 tokens has approximately the same cost as forward and backward pass in a feed-forward network that has 800 layers. [Williams and Peng, 1998] propose a less expensive method. Truncated Backpropagation Through Time (TBPTT) processes one time-step at a time, and every t_1 timesteps it runs BPTT for t_2 timesteps.

To illustrate my implementation of the algorithm, I show how I process an output sequence which is longer than 150 tokens. Let $t_1 = 50$, $t_2 = 100$. At first I let the network predict the first 100 outputs and run the BPTT. I keep aside the 50-th hidden state of the decoder. Next the network (initialized with the hidden state from the 50-th time-step) predicts positions 51 to 150, and again the BPTT is run. Similarly afterwards.

5.2 Baseline model

I use a non-recurrent Encoder and a two-layer LSTM decoder with Luong-style attention as the baseline model. At first I present how the encoding of input records works, then I talk about the text generation. I highlight all the differences between my approach and the one taken by [Wiseman et al., 2017]. The visualization of the model is shown in the figure 5.1.

5.2.1 Encoder

The Encoder should process the input records (the formation of a record is explained in section 3.1) to create the partial outputs at each time-step and the initial hidden state for the Decoder.

[1]The exact GPU used on AIC cluster is NVIDIA GeForce GTX 1080

I described in section 3.3 that each record r consists of 4 different features: *type*, *value*, *entity* and a *home/away flag* depicting if the record belongs to home or away team.

At first each feature is embedded to a fixed-dimensional space. Next the embeddings are concatenated. I choose the same approach as [Wiseman et al., 2017] (who was inspired by [Yang et al., 2016]), and I pass the concatenation through a one layer feed-forward network (*tf.keras.layers.Dense*) with ReLU activation[2], to create the encoding e of the record r. Consequently, the input sequence of records $\{r_i\}_{i=1}^{m}$ is transformed to $\mathbf{e} = \{e_i\}_{i=1}^{m}$.

To create the initial state of the decoder, [Wiseman et al., 2017] calculated the mean of the records belonging to the particular entity and linearly transformed the concatenation of the means. (Simply, the concatenation is passed through *tf.keras.layers.Dense* without any activation function).

To make the implementation simpler I observe that each player entity is represented by 22 records and each team entity by 15 records 3.3. Consequently I approximate the approach taken by [Wiseman et al., 2017] and mean pool over $\mathbf{e}$ with stride 22. (Which means that while means of players are exact, the means of team records are less than approximated) During my experiments I haven't seen any indication that this modification became the performance bottleneck of the model.

5.2.2 Decoder

The Decoder is a 2-layer LSTM network with attention mechanism. The LSTMs are initialized with states prepared in the previous section. I opted to use the Luong style attention with input feeding [Luong et al., 2015]. I described in section 4.2.2 that we use the concatenation of the last attentional decision and the last gold output as the actual input to the first layer of LSTMs. However at the first time-step, when the input is the $<BOS>$ token, there is no *last attentional decision*. Hence for this purpose I use one of the initial states prepared by the Encoder.

5.2.3 Training and Inference

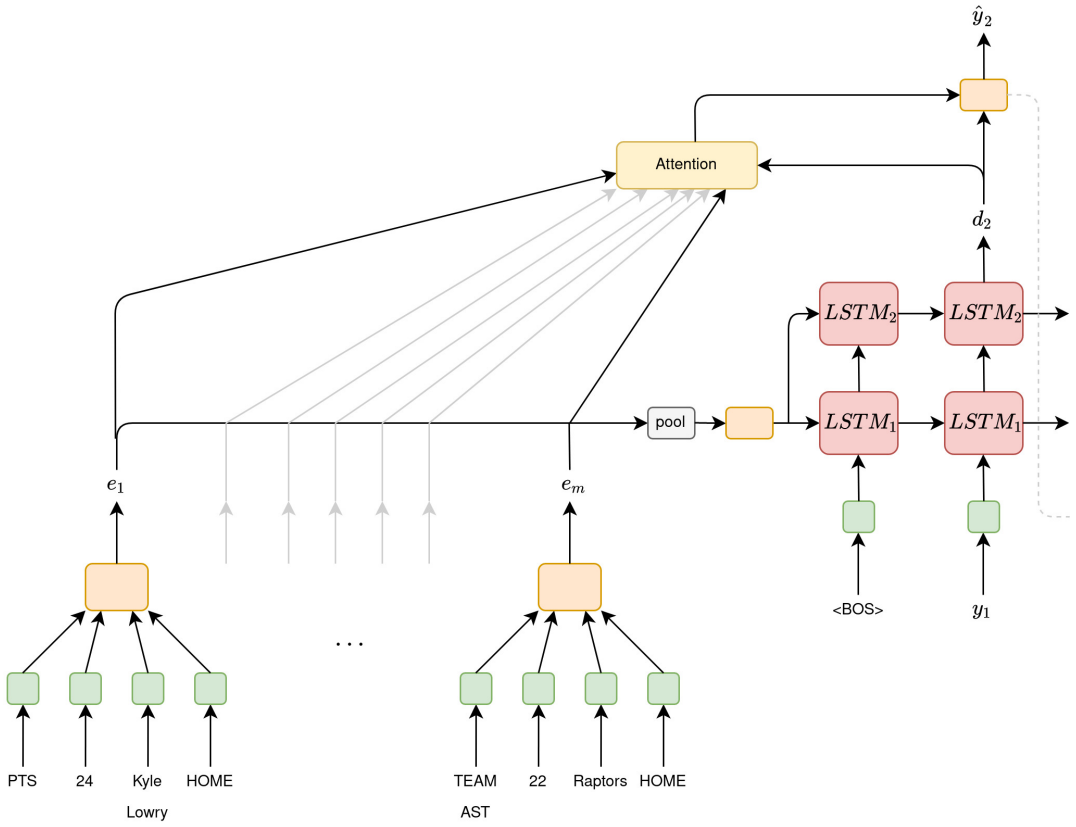
The model is trained end-to-end to minimize the cross-entropy loss (negative-log likelihood) of the gold output summary $\mathbf{y}$ given the sequence of input records $\mathbf{s}$.

$$\min \sum_{(\mathbf{s},\mathbf{y})\in\mathcal{D}} -log\ p(\mathbf{y}|\mathbf{s}) \tag{5.1}$$

We use the *teacher forcing* (as explained in section 4.1.3) during training and during the inference, we greedily approximate

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}'} p(\mathbf{y}'|\mathbf{s}) \tag{5.2}$$

[2]the Rectified Linear Unit activation function $f(x) = max(0, x)$



Note: green cells are embedding layers, orange cells are feed-forward networks

Figure 5.1: The Rotowire baseline model at the second time-step.

5.3 Improving the Baseline with Copy Mechanisms

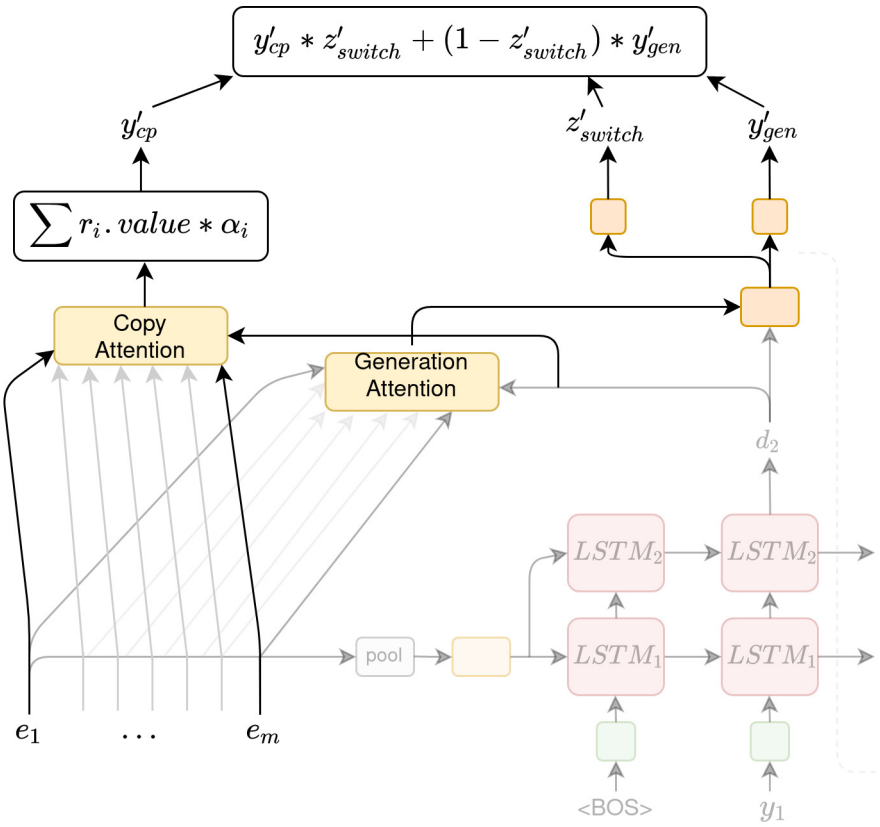
In the previous chapter 4.1.4, I underlined *hallucinations* as one of the problems of the Encoder-Decoder architecture. In the task on RotoWire dataset it basically means that during training, the model learns that it should put *some number* at a specific place in the sentence. We would like it to generate the *exact value*.

Therefore as the next step I incorporate the *Joint Copy mechanism* [Gu et al., 2016], [Yang et al., 2016] (already described in 4.3). The encoder rests intact, however in the decoder I use another attention module to point to specific record from the input table and a feed-forward network which models if the model should generate or copy.

The generational path is the same as in the baseline model. In the copy path I use the alignment vector from the newly added attention as weights, to compute the weighted sum of the value portion of input records. The visualization can be seen in the figure 5.2.

5.4 Content Selection and Planning

Rather than training end-to-end, [Puduppully et al., 2019] suggest to divide the task to *content planning* and *text generation*. The *content plan* describes "what



Note: green cells are embedding layers, orange cells are feed-forward networks
α is the alignment vector produced by the copy attention; $r_i.value$ is the value portion of the i-th input record.

Figure 5.2: The Joint-Copy extension of the baseline model at second time-step.

to say, and in what order". During the *content planning* stage the model selects some records from the input table and organizes them to a *content plan*. During the *text generation* stage, the model generates the text based on the records pointed to by the *content plan*.

Both tasks are modelled with *the same* Encoder and with *separate* Decoders.

5.4.1 Content Selection

[Puduppully et al., 2019] improves the baseline encoder by incorporating *context awareness*. At first the input records are encoded in the same way as in the baseline model 5.2, the self-attention is used to model the "importance vis-a-vis other records in the table".

Specifically, we compute

$$
\begin{aligned}
\forall k \neq t : \alpha_{t,k} &= score(\hat{r}_t, \hat{r}_k) && \textit{the score vector} \\
\boldsymbol{\beta_t} &= softmax(\boldsymbol{\alpha_t}) && \textit{the alignment vector} \\
\boldsymbol{\gamma_t} &= \sum_{i=1}^{m} \beta_{t,i} * \hat{r}_i && \textit{the context vector} \\
\hat{r}_t^{att} &= sigmoid(W_{cs}[\hat{r}_t, \boldsymbol{\gamma_t}]) && \textit{gating mechanism} \\
\hat{r}_t^{cs} &= \hat{r}_t^{att} \bigodot \hat{r}_t && \textit{the content selected representation}
\end{aligned}
$$

The Encoder thus creates a sequence of *context aware* representations $\{\hat{r}_t^{cs}\}_{t=1}^{m}$ (figure 5.3).

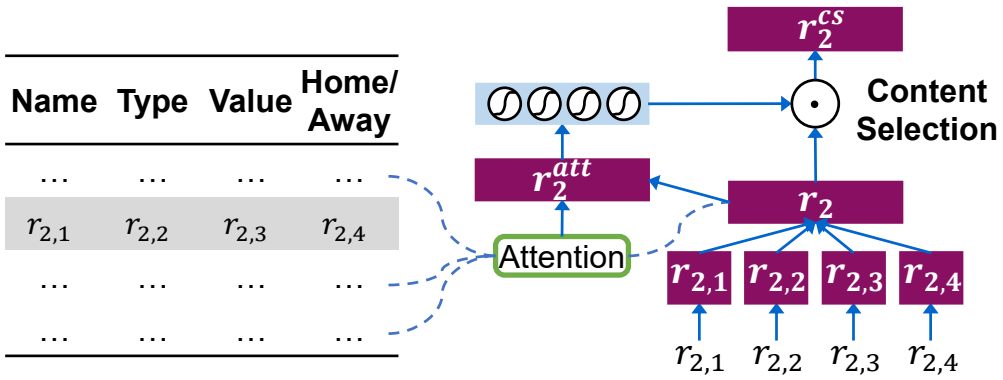


Figure 5.3: Content Selection mechanism (image is directly from [Puduppully et al., 2019])

5.4.2 Content Planning

[Wiseman et al., 2017] experimented with conditional copy approach, in which the latent *switch* probability isn't marginalized out. Hence, there exists pointer sequence for each summary in the train and validation dataset. The sequence corresponds to the order in which entities and values from the sequence of input records appear in the output summary. [Puduppully et al., 2019] suggested that instead of just modelling the switch probability, we can train a Decoder to extract these pointers from the original table.

As suggested, the Content Plan Decoder is a one layer LSTM which operates on the *context aware* representations $\hat{\mathbf{r}}^{cs}$. Its hidden states are initialized with $avg(\{\hat{r}_t^{cs}\}_{t=1}^{m})$. [Puduppully et al., 2019] haven't elaborated on the exact approach, hence it is probable that I have diverged a little bit from the intentions of original authors.

The Text Decoder is trained to start the generation when it sees the $<BOS>$ token. Since the Content Plan Decoder operates on $\hat{\mathbf{r}}^{cs}$ I've choosen to prepend a special $<BOS>$ record to the sequence of input records, and also a pointer to the $<BOS>$ record is prepended to each content plan. Therefore instead of teaching the Content Plan Decoder to start generating content plan when seeing a special value, I teach it to do so when seeing *the encoded representation of the special value*. The same approach is taken at the end, with the $<EOS>$ record.

Either baseline Decoder or Joint-Copy Decoder can operate on the generated content plan, to generate the output summary.

5.4.3 Training and Inference

As oposed to Baseline and Joint-Copy models, the Content Planning model has multiple outputs. Therefore we train the model to minimize the joint negative log-likelihood of the gold summary $\mathbf{y}$ and the gold content plan $\mathbf{z}$ conditioned on the sequence of input records $\mathbf{s}$. It deserves a note that we use the gold content

Type	**Entity**	**Value**	**H/A flag**
<<BOS>>	<<BOS>>	<<BOS>>	<<BOS>>
TEAM-CITY	Raptors	Toronto	HOME
TEAM-NAME	Raptors	Raptors	HOME
TEAM-PTS	Raptors	122	HOME
TEAM-CITY	76ers	Philadelphia	AWAY
TEAM-NAME	76ers	76ers	AWAY
TEAM-NAME	76ers	76ers	AWAY
TEAM-PTS	76ers	95	AWAY
...	...	...	...

Note: The extract corresponds to sentence: "The host Toronto Raptors defeated the Philadelphia 76ers , 122 - 95..."

Table 5.1: An extract from the content plan corresponding to the summary from the figure 2.1

plans as the inputs to the text generation part of the model.

$$\arg\min_{\mathbf{y},\mathbf{z}} p(\mathbf{y}|\mathbf{z},\mathbf{s}) + p(\mathbf{z}|\mathbf{s}) \tag{5.3}$$

6. Experiments

In the previous chapters I have presented the dataset (chapter 2), and discussed challenges which arise from its properties (chapter 3). Next I described the neural network architectures (chapter 4) which served as building blocks for the models introduced in chapter 5. Now I want to connect all the parts together, elaborate on the trainining of the models and analyze their results.

Let me recap the main challenges we face, and hypothesize about the ideal generation system. Firstly, the target summaries are really long. The ideal system should remember what has already been generated and shouldn't produce duplications. Secondly, the targets contain a lot of facts based on the input structured data. The ideal system should copy these facts from the input and reduce *hallucinations*. Lastly, the generated text should be as close to English as possible *while meeting the requirements described previously.*

6.1 Evaluation Methods

During evaluation I want to measure which model resembles the hypothetical ideal model the most. To do so, I report the BLEU score [Papineni et al., 2002] of the generated summaries on the development and test sets. Although it is the gold standard, there are many people arguing against its usage as a performance metric [Celikyilmaz et al., 2021], and I found that the networks producing more factually correct statements doesn't score better. Therefore I also manually evaluate a subset of the generated summaries. I focus on the factual correctness, the reduplications and inter-sentential coherence. (If two sentences in the generated summary doesn't contradict each other)

[Wiseman et al., 2017] proposed three custom automated metrics to evaluate the performance of their models. They call them *Content Selection* ("how well the generated document matches the gold document in terms of selecting which records to generate"), *Relation Generation* ("how well the system is able to generate text containing factual (i.e., correct) records") and *Content Ordering* ("how well the system orders the records it chooses to discuss"). The metrics are implemented in outdated neural network framework *Torch* and I wasn't able to execute it on my computers. After discussions with my advisor we agreed to not adopt these methods.

6.2 Baseline Model

At first I describe the hyperparameter choice. I embed the *type* and *home/away* fields to $\mathbb{R}^{300}$ and *entity* and *value* fields to $\mathbb{R}^{600}$. All the hidden states of the LSTMs as well as the representation of the input record (produced by Feed-Forward Network as described in 5.2) are in $\mathbb{R}^{600}$.

6.2.1 Results

6.3 Implementation Details

As stated in the introduction this thesis is highly theoretical and experimental. The implementation serves as proof-of-concept and doesn't aim to be used in the production.

All the models and preprocessing methods were developed in *python 3.8* and *tensorflow 2.4.1*. However the code is compatible with *python 3.6* and *tensorflow 2.3* (the versions used on Artifical Intelligence Cluster (AIC) where the training was executed). The implementation is divided into two modules, *preprocessing* and *training*.

6.3.1 Preprocessing Module

The preprocessing happens in four steps.

1. Filtering out the faulty data-points (section 3.7) from the original dataset.
2. Extraction and transformation of the summaries from the cleaned dataset.
3. Byte Pair Encoding of the summaries. (As explained in section 3.6 I use the *subword-nmt* module by [Sennrich et al., 2016].)
4. Construction of the dataset from the encoded summaries and the cleaned data.

Each step is implemented in *python* and the steps are connected by a shell script.

6.3.2 Training Module

The training module contains implementation of layers and models discussed in previous chapters as well as training, evaluation and inference methods. It makes use of *graph execution*[1] during training and *eager execution*[2] during evaluation and prediction.

The code is available at `https://github.com/gortibaldik/TTTGen/`.

[1] `https://www.tensorflow.org/guide/intro_to_graphs`
[2] `https://www.tensorflow.org/guide/eager`

Conclusion

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014. URL `https://arxiv.org/abs/1409.0473`.

Asli Celikyilmaz, Elizabeth Clark, and Jianfeng Gao. Evaluation of text generation: A survey, 2021.

David L. Chen and Raymond J. Mooney. Learning to sportscast: A test of grounded language acquisition. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, page 128–135, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582054. doi: 10.1145/1390156.1390173. URL `https://doi.org/10.1145/1390156.1390173`.

Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches, 2014a. URL `https://arxiv.org/abs/1409.1259`.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014b. URL `https://arxiv.org/abs/1406.1078`.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O. K. Li. Incorporating copying mechanism in sequence-to-sequence learning, 2016. URL `https://arxiv.org/abs/1603.06393`.

Caglar Gulcehre, Sungjin Ahn, Ramesh Nallapati, Bowen Zhou, and Yoshua Bengio. Pointing the unknown words, 2016. URL `https://arxiv.org/abs/1603.08148`.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks, 2015. URL http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

Remi Lebret, David Grangier, and Michael Auli. Neural text generation from structured data with application to the biography domain, 2016. URL https://arxiv.org/abs/1603.07771.

Percy Liang, Michael Jordan, and Dan Klein. Learning semantic correspondences with less supervision. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 91–99, Suntec, Singapore, August 2009. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/P09-1011.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015. URL https://arxiv.org/abs/1508.04025.

Christopher Olah. Understanding lstm networks, 2015. URL https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei Jing Zhu. Bleu: a method for automatic evaluation of machine translation. 10 2002. doi: 10.3115/1073083.1073135.

Ratish Puduppully, Li Dong, and Mirella Lapata. Data-to-text generation with content selection and planning, 2019. URL https://arxiv.org/abs/1809.00582.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*, page 696–699. MIT Press, Cambridge, MA, USA, 1988. ISBN 0262010976.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units, 2016. URL https://arxiv.org/abs/1508.07909.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks, 2014. URL https://arxiv.org/abs/1409.3215.

Craig Thomson, Ehud Reiter, and Somayajulu Sripada. Sportsett: Basketball - a robust and maintainable dataset for natural language generation. August 2020. URL https://intellang.github.io/. IntelLanG : Intelligent Information Processing and Natural Language Generation ; Conference date: 07-09-2020 Through 07-09-2020.

Lisa Tozzi. The great pretenders, 1999. URL http://weeklywire.com/ww/07-05-99/austin_xtra_feature2.html.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks, 2015. URL https://arxiv.org/abs/1506.03134.

Hongmin Wang. Revisiting challenges in data-to-text generation with fact grounding. In *Proceedings of the 12th International Conference on Natural Language Generation*, pages 311–322, Tokyo, Japan, October–November 2019. Association for Computational Linguistics. doi: 10.18653/v1/W19-8639. URL https://www.aclweb.org/anthology/W19-8639.

P.J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. doi: 10.1109/5.58337.

Ronald Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 2, 09 1998. doi: 10.1162/neco.1990.2.4.490.

Sam Wiseman, Stuart M. Shieber, and Alexander M. Rush. Challenges in data-to-document generation, 2017. URL https://arxiv.org/abs/1707.08052.

Zichao Yang, Phil Blunsom, Chris Dyer, and Wang Ling. Reference-aware language models, 2016. URL https://arxiv.org/abs/1611.01628.

List of Tables