



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Morphological segmentation using ConvLSTM networks

BACHELOR'S THESIS

Author

Géza Velkey

Advisor

Judit Ács

December 6, 2017

Contents

Kivonat	i
Abstract	iii
1 Introduction	1
2 Related work	3
3 Data	7
3.1 Preprocessing	8
3.2 Analysis	9
3.2.1 Sentence dataset	9
3.2.2 Word dataset	10
3.2.3 Morpheme boundary analysis	11
3.2.4 Padding analysis	12
3.3 Data generator	12
4 Neural Network Architectures	15
4.1 LSTM	16
4.2 Bidirectional LSTM	18
4.3 Feedforward Convolutional Networks in 1 Dimension	19
4.4 Convolutional LSTM	20
5 Models	23
5.1 Stacked bidirectional FC-LSTM with conv1D	24
5.2 Dynamic stacked bi-FC-LSTM with conv1D	25
5.3 Stacked fully convolutional biLSTM	27
5.4 Losses	27
5.4.1 Least absolute deviations	28

5.4.2	Least square errors	28
5.4.3	Cross Entropy	29
5.5	Optimizers	29
5.5.1	Gradient Descent	29
5.5.2	RMSProp	30
5.5.3	Adam	30
6	Experimental Setup	33
6.1	Computers	33
6.2	Software Stack	34
6.2.1	Tensorflow	34
6.2.2	PyTorch	35
6.2.3	Tensorflow Fold	36
6.2.4	Other libraries	37
6.3	Environment, Train Helpers	37
6.3.1	Hyperparameter optimization	38
6.3.2	Data pipeline	40
6.3.3	Saving and logging	41
6.4	Evaluation metrics	42
6.5	Inference	43
7	Experiments	45
7.1	Loss function selection	45
7.2	Dynamic and static network comparision	46
7.3	Optimizers and learning rate	47
7.4	Sentence experiments	48
7.5	Word experiments	50
8	Conclusion	53
	Bibliography	59

HALLGATÓI NYILATKOZAT

Alulírott *Velkey Géza*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2017. december 6.

Velkey Géza
hallgató

Kivonat

A szakdolgozat célja morfológiai szegmentációs modellek létrehozása LSTM és konvolúciós LSTM modellek felhasználásával. Magyar nyelvű szavakon és mondatokon kísérletezünk többféle neurális hálózattal annak érdekében, hogy minél precízebb modelleket fejlesszünk ki, melyek túlteljesítik a legkorszerűb mélytanuló megoldásokat is. Továbbá sebességük és konvergenciabeli különbségek alapján összehasonlítjuk a statikus és dinamikus hálózatokat. Bemutatjuk a főbb, gradiens alapú minimalizálást alkalmazó mélytanuló algoritmusokat, különös tekintettel a konvolúciós és rekurrens neurális hálózatokra. Ismertetjük az alkalmazott módszereket, amelyekkel a magasabb pontosság elérése érdekében komplex architektúrákat állítottunk össze. Az architektúrák létrehozásánál nagyban segítettek munkánkat a tudományterületen végzett legújabb kutatások, eredmények. Mivel a modellek adattal jól skálázódnak, a bemutatott hálózatok teljesítménye a jövőben tovább növelhető nagyobb korpusz bevonásával. Hyperparaméter optimalizálást végeztünk a legjobban teljesítő hálózatok megtalálásához, mellyel feltérképeztük az architektúrában hasonlító, de más paraméterekkel bíró modellek pontosságát. Semmilyen magyar nyelv-specifikus szabályt nem alkalmaztunk a megalkotásuknál, így a modellek teljesen nyelvfüggetlenek. Ennek köszönhetően a későbbiekben más nyelvekre is könnyedén átvihető modellek készültek, melyek a morfológiai szegmentáción kívül más, címkézéssel kapcsolatos feladatot is képesek ellátni. Összehasonlítottuk a legjobban teljesítő szó és mondat alapú modelleket annak érdekében, hogy megtaláljuk a legjobb magyar nyelvű szegmentáló algoritmust. Ezek teljesítményét automatikusan értékeltük a tanítás során, elkerülve a nagyobb modellek túltanítását. Készítettünk továbbá Python-alapú notebookokat, melyek segítségével a modellek eredménye vizsgálható, illetve értékelhető, hogy mennyire közelíti az emberi pontosságot a modellek által szolgáltatott predikció. A teljes kódot nyílt forráskódúvá tettük, valamint az adatot és a legjobban teljesítő modelleket csatoltuk a repository-hoz.

Abstract

This thesis consists of experiments on morphological segmentation using convLSTM and LSTM networks. We define multiple new neural network architectures to reach state-of-the-art performance on the morphologically rich Hungarian language. This work provides a comparison of static and dynamic neural networks by describing the current deep learning frameworks and benchmarking their speed and convergence capabilities. We give a brief description of the main machine learning architectures, such as convolutional networks and recurrent neural networks and how we can connect them together to reach better performance. Our best performing models utilize most of the recent improvements from the deep learning field. The thesis introduces new architectures which can be further improved due to their completely data-driven behaviour. We also perform a wide hyperparameter search on all kinds of models to drive further current neural network approaches for character-level segmentation. All the introduced models are completely language-independent and they can serve not only morphological but other sentence-tagging tasks due to their very natural and highly recurrent architecture. We compare the best performing models trained with sentence and word inputs. The experimental results are automatically evaluated during the training in order to help the development of the networks and to prevent large models from overfitting the train dataset. An inference framework is provided for testing the machine learning models on unseen words, in order to evaluate the results of the segmenting algorithms. All the notebooks and code can be found in the open-source repository along with the model descriptions and data.

Chapter 1

Introduction

Morphological analysis is the task of segmenting a word into morphemes, the smallest meaning bearing elements of natural languages. It is a difficult problem for learning algorithms and until recent improvements in computer based natural language solutions it needed rule-based solutions. These solutions were created with thousands of hours of work by professionals for every single language. With the rise of computational resources in the last 5 years neural networks became deeper and deeper, so they became more capable of learning complex problems.

The power of deep learning architectures can be used to learn the segmentation rules by only learning from already labeled data. As mentioned it's a supervised learning task, with the main advantage of having a model which is suitable for different languages too. It's even possible that a neural network based solution will create better answers for unseen words and sentences, which may be impossible using rule-based solutions.

The idea of processing natural language with computers is not new, all the state-of-the-art spell checkers, advisors, translators already use neural networks in their pipeline. Statistical methods can also be applied in many cases, but nowadays deep neural networks dominate the machine learning field of such complex tasks.

In morphological segmentation we usually have a large training set with good quality, because the already existing rule-based systems help the data generation and data augmentation too. The best training set is usually hand-labeled by people, and in this field it is not uncommon, because the studied languages are heavily used by millions of people.

Recently researches experimented with recurrent neural network architectures, for example LSTMs, bidirectional LSTMs, and even convolutional recurrent networks.

They are the natural architecture of sequence processing and labeling, so most of the NLP applications use them as the core of their model.

The goal of this thesis is to prove the capabilities of LSTM models with convolutions in natural language processing, and to showcase the usage of dynamic neural network architectures for tasks with varying input and target lengths.

Chapter 2

Related work

Morphological segmentation is a complex task in languages where the number of morphemes is high. In English the average word does not usually contain much morphemes, while Hungarian is one of the morphologically rich languages.

“Segmentation is usually the side product of full morphological analysis done by finite state transducers. However, these solutions do not support languages where word boundaries are not denoted.” described [1]. Our segmentation goal is the same as in [1], so we are strictly focusing on only segmenting the words into morphemes. Last semester we worked with variational autoencoders¹ for word compression. The results can be useful for future models to boost classification performance.

The languages can be classified according to their morphology into three classes [10]:

- isolating languages: little or no morphology, e.g. Chinese
- agglutinative languages: where a word can be decomposed into a large number of morphemes, e.g. Turkish
- inflectional languages: morphemes are fused together, e.g. Latin.

One of the first machine learning solutions is memory-based morphological analysis on Dutch words [36]. Their model has proven its learning capabilities, but back then the computational resources were much lower, so they could not achieve better performance than 65% on unseen words.

There are solutions for morpheme segmentation which only use unsupervised learning methods, called ULM [4]. These models have the huge advantage compared to supervised solutions, that in this case there is no need for labels, just the training

¹<https://github.com/evelkey/vahun>

dataset. With this solution we can eliminate errors from the labels. These solutions are highly in research state, but the results are very promising. [8] also uses unsupervised learning for morphological segmentation, and applies the same algorithm on 4 languages successfully. Morfessor [7, 38] is the most known unsupervised solution for linguistic morpheme segmentation. The algorithm is available on the Internet, and can be easily used. [34] compared Adaptor Grammars, a nonparametric Bayesian modeling framework, with unsupervised learning, and analyzed them on 5 different languages. Morfessor 2.5’s performance is very surprising, because of its pure unsupervised learning approach. It does not require labels to complete its task, and achieves around 0.60 in F1 score in different languages.

In 2017 the first shared task on Cross-lingual Word Segmentation and Morpheme Segmentation[41] was organized specifically on supervised segmentation. Hungarian is not included in the dataset, but the algorithms can be applied assuming the same data format. A team submitted a solution for all 8 languages in the shared task [9]. Their model used a BiRNN-CRF solutions which outperformed the Morfessor baseline in every language as well as all the other solutions. In particular, they achieved 0.907 F1-score on morpheme boundary prediction on Finnish, the only Uralic language in the dataset.

Segmentation quality in [10] reached with the new unsupervised learning algorithm is good enough to improve grapheme-to-phoneme conversion. Their model is fully language independent, so it can be applied on any morphologically rich language.

Another language independent solution is [11], which only uses the Wiktionary as its input. The approach is completely data driven, similarly to our solution. Their system automatically acquires the orthographic transformation rules of morphological paradigms from labeled examples, and then learns the contexts in which those transformations apply using a discriminative sequence model.

A supervised approach is [32], where conditional random fields were used to define morpheme boundaries. [40] uses a very similar architecture for Chinese language. This solution requires low resources, on the other hand it needs labeled data, which can be a drawback in cases where labeled data is expensive and not precise enough.

To address morphological segmentation in Arabic languages researchers created a trigram language model [20] to determine the most probable morpheme sequence for a given input. Their solution is semi supervised. They used a large corpus of 155 million words, and a small subset of about 110 thousand manually segmented words. Their solution reached 97% word accuracy, which was evaluated on an independent dataset. Another improvement for the morphologically rich Arabic language was introduced in [30]. The authors provide a minimally supervised algorithm, which

reduces the unknown words at translation time by using their morphologically segmented format to eliminate unknown word forms.

Researchers connected eye movements while reading with the interpretation of compound words in [2] in English. The results showed that, the human interprets these words as retrieving the whole compound word’s representation, instead of tearing the words into morphemic constituents. This may also help the future work with convolutional networks, which were inspired by the human vision’s neural networks.

The most recent result for Hungarian language, and current state-of-the-art solution is introduced in [1], where bidirectional GRU architectures were used for segmentation, and they reached an F1 score of 0.927, which is higher than the shared task’s results for the similar Finnish language. We considered these results as the baseline for our experiments with new architectures.

Chapter 3

Data

The data used for training segmentation algorithms contains high precision hand-labeled data from the Szeged Korpusz¹. The description of the creation process is defined in [1] as the following: “It is the only known Hungarian corpus which provides gold standard morphological analysis [37]. When it was created, it used its own tagset, but recently it has been converted to Universal Dependencies and also to the *e-magyar* framework’s tagset. *e-magyar* contains a wide variety of NLP tools, and also includes a HFST-based morphological analyzer called *emMorph* [27, 26]. The HFST [21] output can be easily converted to segmentation, and since no corpus contained the needed segmentation, the new corpus was created by directly exporting the output of HFST. The automatically converted *e-magyar* corpus was used in the present corpus for disambiguation. It was found that less than 82% of all tokens could be matched to a single HFST output. The others had zero or more than one matches. If there were more than one matches, then always the first one was chosen for the given word. In case of zero matches, partial matching solved the problem in most of the cases by leaving out some of the morphemes from the given word. The match with fewest missing morph tags was selected as the disambiguated one.

Hungarian morphology is considered to be concatenative, but not every inflection can be segmented strictly at character-level due to assimilation, low-vowel lengthening, etc. We call the result of the segmentation a segment, to emphasize that this segmentation is strictly orthographic (character based). For our silver segmentation corpus we decided to use the HFST output, because it is already disambiguated, and ready to use.”

We used the same dataset for our task as [1], so the results are comparable to each other. The corpus was then splitted into a tsv file, which contains the morpheme

¹The source data can be found at http://avalon.aut.bme.hu/~judit/resources/szeged_morph_disambig_corp.tar.gz

labels, boundaries, and classes. The sentence boundaries are usually marked with empty line in the dataset, and the words are in the same order as their position in the sequence.

For our task we did not need the morpheme classes, only the boundaries were important. These boundaries were extracted from the extracted corpus described above. The data was converted and analyzed with a Python script² to form the needed datasets for the neural networks. The script is re-runnable at anytime, and if a new corpus with relevant segmentation is released, our dataset can be recreated easily. In this morphological segmentation task the labels were marking the morpheme boundaries only. In our dataset it was given on a word level approach with spaces as segment limiters.

To create the morpheme boundary labels we created a sequence of ones and zeros, which had the same length of the given input sentence or word. Label one meant that the character at the same position in the input sentence is the start of a new morpheme, while label zero meant the opposite. This is a dense representation of the boundaries and can be used by neural networks for sequence tagging problems.

For the sentence-based training the data was shuffled on sentence level, because it was read from different books in a given sequence. Without the shuffling the neural networks would have over-fitted to the context of the currently processed book. The shuffled data was then split into train, development and test datasets in a ratio of 90%-5%-5%. All the datasets were selected randomly from the same distribution in order to avoid anomalies in the training procedure.

For the reference training with word level segmentation, which lacks the context of the word, the training database was created in the same way. The only difference was that the data was shuffled on word level.

3.1 Preprocessing

During the preprocessing of the data a vocabulary was constructed from the datasets, which contained all the characters. The size of this vocabulary was the size of our one-hot encoding. The characters were mapped into one-hot representation to train the neural networks. These encoded vectors were then concatenated to a matrix. Because the sentence and word lengths were indeterminate, the matrices had a shape of [length, vocabulary_size]. Because of the speedup and convergence boost provided by mini-batch gradient descent compared to SGD[5], we collected a batch

²create_database.ipynb

Dataset	Sentence	Word
Size	80567	135469
Length average	109.897	10.024
Length standard deviation	80.836	3.650

Table 3.1: Dataset analysis

of these samples together in a Python list object. We experimented with varying batch size from 1(SGD), to 256. The batch size was limited by our computer’s GPU memory, because the full batch, along with the computed forward and backward pass tensors needed to fit in the memory. For this reason we also truncated the length of the sentences, because there were extremely long sentences(500+ characters), which would have been processed by the network, but the didn’t fit in the memory.

3.2 Analysis

Our data is non-standard in a way that the length of the sentences is indeterminate and the sizes even vary inside the batches. This means that we cannot say that our batches have a shape, but they are lists of sentences or words. Before we applied any machine learning algorithms, we studied the database to ease the development of learning algorithms. As it can be seen in Table 3.1, while the datasets have been created from the same source, their length distribution is very different. This difference can even cause convergence problems in the learning algorithms too. Since our focus is to provide morphological segmentation on sentences, the models were focused on this task. For the word-based approach we also tested our and other algorithms and compared the results to the different architectures.

3.2.1 Sentence dataset

The sentence dataset, which can be seen on Figure 3.1 has a much higher standard deviation than the word dataset. For machine learning libraries which only support static computational graphs, it is a computationally expensive property of this task. For these frameworks the we can’t add elements of different shapes to the batches, which means we need to pad the sentences to the same length. This length distribution proposed difficulties during the prediction of long sentences because of the vanishing gradient problem [14].

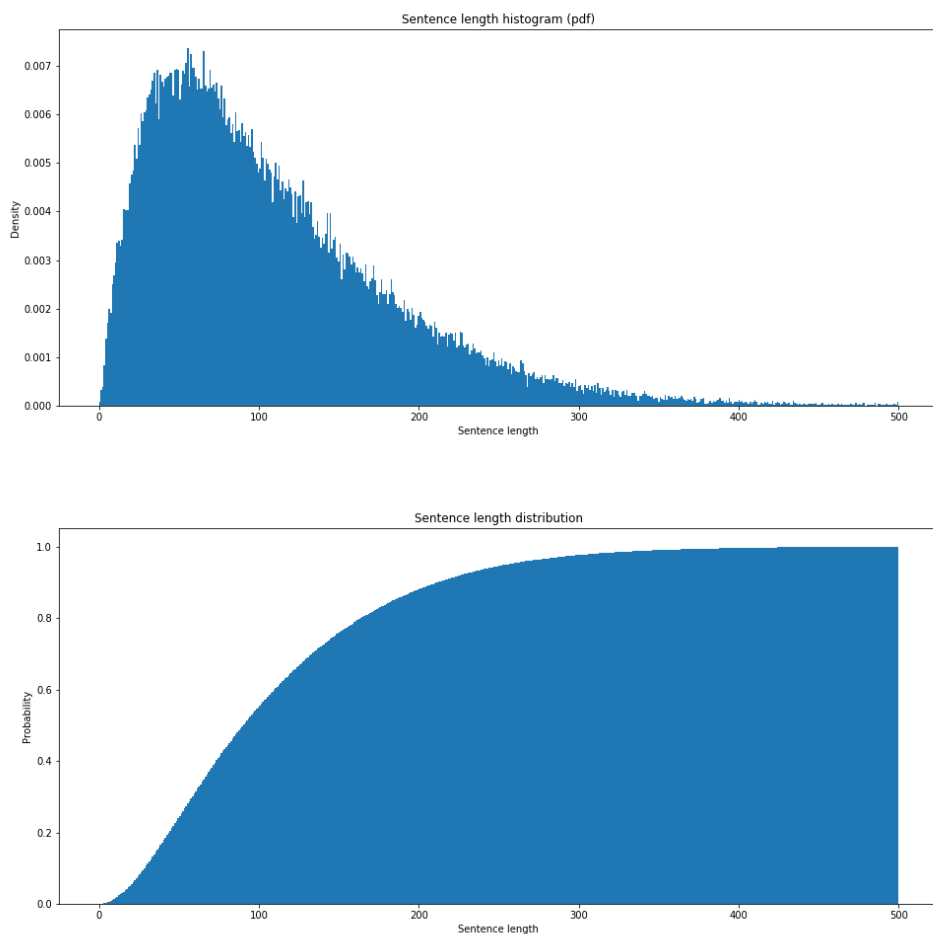


Figure 3.1: Sentence length analysis

A big difference between the sentence and word datasets is very important when we evaluate the model performances. In the word dataset the words are unique in the train-test-validation datasets, so there is no overlap between them. Considering the sentences dataset, the sentences are also unique, but not on word level. It means that the model may have seen the same word in a sentence during training and testing. On word level accuracy it gives a large boost to the sentence models, because the already seen words are segmented with almost 100% accuracy.

3.2.2 Word dataset

Most of the currently existing neural network solutions use word level datasets for natural language processing tasks. The so-called word vectorizing algorithms can map the semantics and syntactics of the words in a given language to an n-dimension space. In this space the words are defined by vectors. The state-of-the-art neural

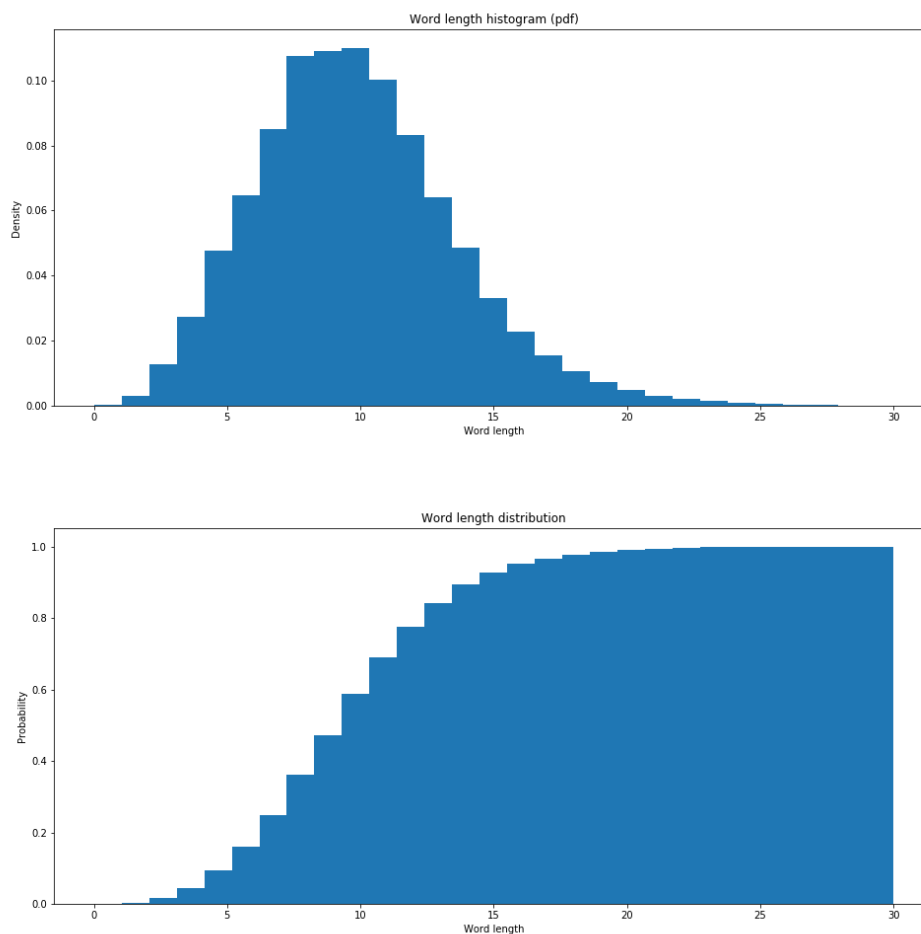


Figure 3.2: Word length analysis

translation systems use these word vectors as their input and apply *seq2seq*, convolutional, and grid-LSTM networks on them.

In the Hungarian language it's possible to have multiple different segmentations of a word based on it's environment. By using only words it's impossible to know the environment of our words, so the best guess of these models in such situations is to predict the most often used segmentation of the given word. With this solution we add an error which cannot be eliminated in any way using this kind of representation. When we created the dataset, it is shown that the fraction of words which has multiple segmentation is below 0.89% in our case. It still means that when we will apply this kind of segmentation algorithm, it may fail in these cases.

3.2.3 Morpheme boundary analysis

In the preprocessing part we introduced that every morpheme boundary was marked with ones in the label sequence, and zeros marked the other characters. The statistical

Property	Value
Zero fraction in sentence labels	91.379%
Zero fraction in word labels	84.913%
Avg morphemes / word	1.512
Avg morphemes / sentence	9.474

Table 3.2: Morpheme boundary analysis

analysis of the labels can be found in Table 3.2. Because the zero fraction in our labels is very high, the models were first going into a local minimum, where every prediction was zero, because this prediction had a character-level accuracy above 90%. The data analysis helped to resolve the first convergence problems, because of the high zero fraction in the dataset. By studying the features, the optimizer for the neural networks was fine-tuned and the training loss was also fitted to the dataset.

3.2.4 Padding analysis

When we were using static computational graphs, the data needed to be padded to the same length. This was achieved on Python level with the help of the numpy library [3]. The padding was added as a wrapper layer around the data generator. When we were training static networks, it was used, but for dynamic networks it was not. We applied zero-padding before the sentence, so the gates of the recurrent cells needed to learn this connection too. Luckily it is not difficult for such a network to learn to predict zeros to a full zero input, so we did not experience any drawback because of this modification.

With adding many zero labels to our dataset, the distribution also changed. That’s why we evaluated the performance of the architecture only on the valid part of the dataset. After prediction we cropped the paddings from the outputs and labels, and then we applied our metrics. This way the padding did not introduce any changes in the evaluation compared to the dynamic networks. The word level accuracies were not affected by the padding, because the paddings were fitted to zero labels, and the networks quickly learned these features.

3.3 Data generator

For machine learning the data pipeline is crucial considering the speed of the training and inference. The data pipeline needs to be at least as fast as the learning algorithm itself. The neural network’s speed depends on the usage. When we are training the

network, the execution speed is much slower, because in this phase we are executing forward and backward pass on the network too. When we are using the already trained network in inference mode, the only needed computation is the forward pass on the model. As these days neural network models in production is becoming mainstream, there is a high need for a good inferencing framework. Nvidia addresses this problem with a continuously developed library called TensorRT³.

Considering our problem, the dataset is small enough to fit in the memory of an average computer. Our data loader loads in a file and creates an object which contains the length of all dataset. It also contains different infinite generators for the train, test and validation datasets. They are infinite, because we may train our network for arbitrary number of epochs on our dataset. These generators can be accessed by just calling their `next()` function. It also gives us the ability to use any kind of wrapper on our data, such as padding. For more details on the data pipeline there is more information in Section 6.3.2.

³<https://developer.nvidia.com/tensorrt>

Chapter 4

Neural Network Architectures

Recently there were many advancements in deep learning related to natural language processing, and many researchers tried to use the newly discovered architectures in morphological segmentation. Most of the ideas were addressing this problem on word level models, and tried to reach the performance of the highest precision hand labeling. The progress in this field was pretty big, because this problem can be interpreted as a sequence tagging task. Every labels means if that character is a morpheme boundary or not. Our dataset is sequence-like, so recurrent neural networks (RNN) seem to be the best architecture. Recurrent neural networks have inner loops in the architecture, so they can keep information by persisting it from cell to cell.

However, our dataset contains sentences and they can be longer then a few hundred characters. The standard RNN cell is not able to learn so long-term dependencies to the vanishing gradient problem [14]. All the RNN models are trained with the algorithm called back-propagation through time (BPTT) [39]. With BPTT we can propagate back the error to every time-step. The training has the same problem as very deep neural networks. When we are multiplying many numbers which are smaller then one and bigger then zero, which is the output range of the sigmoid function, the product will converge to zero. To eliminate this problem researchers introduced a new architecture, which can learn long-term dependencies too. Because of it's capabilities they named it the Long Short Term Memory [12]. Recently there has been advancements in this field and new architectures were created in the last few years. One of the best performing, yet simpler model is the Gated Recurrent Unit (GRU) [6]. This architecture uses less gates, and less weights, and still outperforms LSTM in custom tasks. Our growing computational resource is allowing us to experiment with new architectures. In [43] Google researchers used a huge cloud to create new recurrent cells and feed-forward architectures automatically. The model

was a RNN, which created architectures, and it was trained using reinforcement learning.

For morphological segmentation we used fully connected LSTM and convolutional LSTM models on sequences and words. Here we introduce the used neural network concepts and architectures.

4.1 LSTM

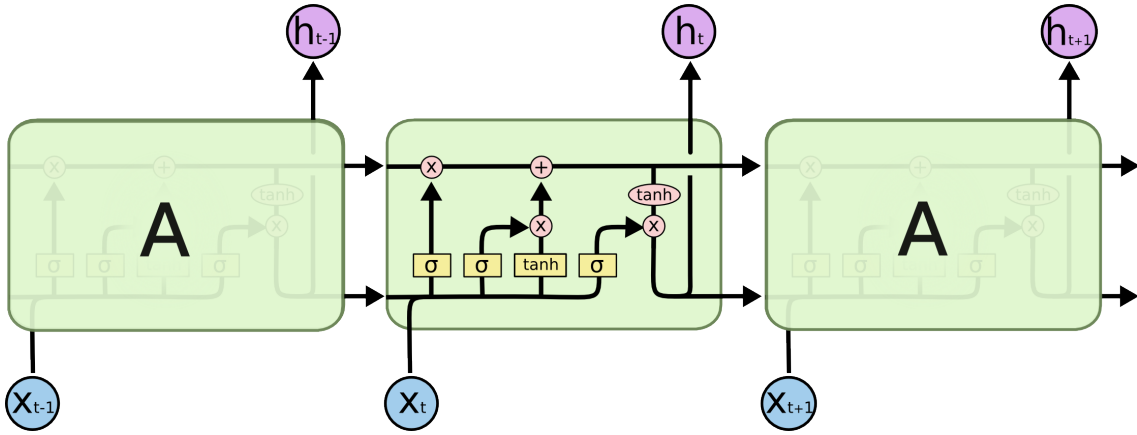


Figure 4.1: The Long Short Term Memory block

Long Short Term Memory addresses the vanishing gradient problem with adding gates to the recurrent cell. These are the input, forget and output gates. The input gate controls what is the relevant information of the input. The forget gate decides what to forget from the last state, and the output gate controls the output. All these gates know the current input and last state of the cell. In fully connected LSTM these gate values are matrix multiplications with teachable weights. The following equations describe the gates in the cell. The letters i, f, o are meaning input, forget output. The c and h are meaning the cell state and cell output at a given time-step. Values marked with W and b are teachable parameters. σ marks the sigmoid function, and \odot is the Hadamard-product of two vectors.

$$\begin{aligned} i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci} \odot c_{t-1} + b_i) \\ f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf} \odot c_{t-1} + b_f) \end{aligned} \quad (4.1)$$

The next cell state is calculated in every timestep as the following equation describes:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \quad (4.2)$$

Then, the cell's output is calculated as:

$$\begin{aligned} o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co} \odot c_{t-1} + b_o) \\ h_t &= o_t \odot \tanh(c_t) \end{aligned} \quad (4.3)$$

These gates make it possible to memorize long-term dependencies too. In many tasks, like addition of sequences, copying sequences, machine translation this architecture is among the state-of-the-art solutions with neural networks [12, 35, 13]. It is also possible with this gated architecture to apply padding on the sequences. The padding can be learned in the first few training steps by the gates, so they will not affect the performance of the predictions. The LSTM is among the best-performing general recurrent cells, and its performance is still being improved by adding more complex optimizers for highly recurrent architectures.

Recently new architectures has been discovered by using LSTM as a core idea. The performance of these models can be increased by stacking multiple LSTMs on each other. As it can be seen on Figure 4.2, the higher-level cells get their input from the lower-level cell's outputs. This performance gain is similar to the deep neural network solution, where we stack more and more layers, to be able to recognize more difficult patterns. The multiple recurrency make it possible for every cell in the stacked architecture to transfer information even for long-term dependencies.

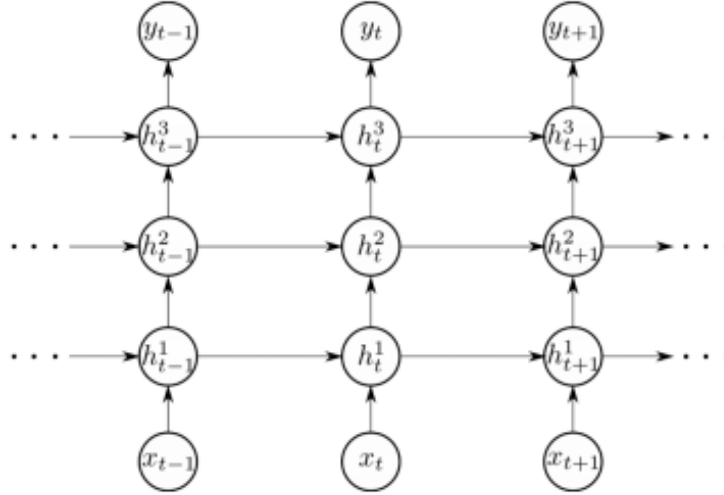


Figure 4.2: Stacked LSTM

A newer and better performing architecture is the Grid-LSTM [15]. This architecture not only stacks layers upon each other, but the states are different in every dimensions in the grid. It contains a more complex cell, which is compatible with arbitrary dimensions, for example existing state-of-the-art solutions use 2D and 3D grid LSTMs. The information and gradient flow is different from the casual LSTMs,

because in this architecture the information gets to the output via different paths. These multiple paths make it possible to model more complex features.

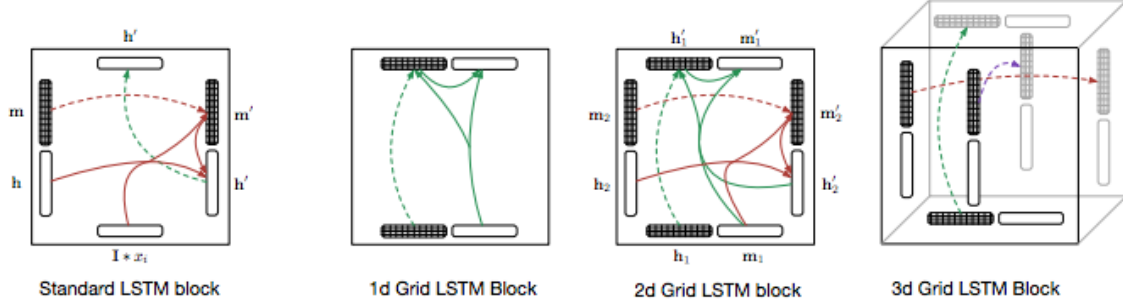


Figure 4.3: Grid LSTM

Considering our problem’s complexity, we only used stacked LSTMs, but for future performance improvements Grid-LSTM may play a big role, due to the many parallel routes given in the architecture.

4.2 Bidirectional LSTM

Bidirectional recurrent models are using the idea of not only passing the data in right-to-left order, but left-to-right order too. With this solution the network will always know the exact context of a given item in the sequence. This is highly relevant in our problem, because for deciding morpheme boundaries it is needed to know what is before and after the the given boundary position.

This bidirectional architectures have the advantage that they can use any kind of recurrent cell. They are compatible with GRU, LSTM, ConvLSTM, NAS and many more cells. The more complex stacked cells are also working fine with this model, so it gives the opportunity to create the best performing neural network for our task. It also means that, if we apply cells, which preserve the long-term dependencies too, then we have a different representation of the sentence compressed into every characters bidirectionally mapped output. This is proven to boost performance and give marginally better results on problems similar to ours.

4.3 Feedforward Convolutional Networks in 1 Dimension

Convolutional neural networks were introduced in [19] for image, speech and time-series classification. This architecture reduced the number of parameters, while keeping the spatial information of the input image. This way the architectures could be much deeper, while keeping the number of parameters low. Since then it is the most used architecture in image processing. The current state-of-the-art solution does not even use fully connected layers anymore, but they solve the problem with Fully Convolutional Neural Networks (FCNN), like [23] solves semantic segmentation problems by using convolutions only.

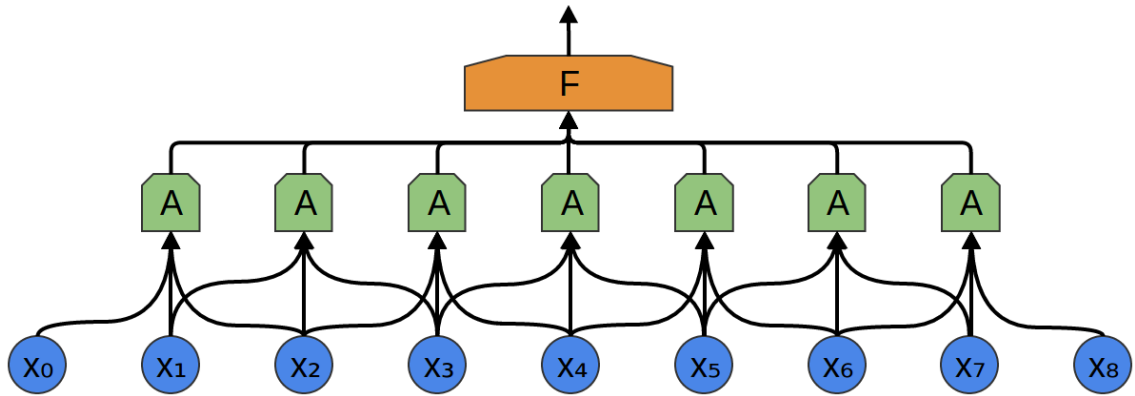


Figure 4.4: 1D convolution: A is the kernel of the convolution, F is fully a connected layer

Convolution is one of the most-used operation in signal processing, and it is the most popular architecture for image classification [19]. Convolution in 1 dimension is implemented as a special case of 2D convolution in Tensorflow. It can be described as convolving a filter on a sequence. The filter has the following shape:

$$kernel_shape = [input_channels, kernel_size, output_channels] \quad (4.4)$$

The *input_channels* means the number of attributes in the input sequence. *kernel_size* means the length of the kernel, we apply on the dataset. *output_channels* is the number of attributes in the output. We can also apply padding on our layers. For the experiments we used a stride of 1, so the filter was run on overlapping patches of the input.

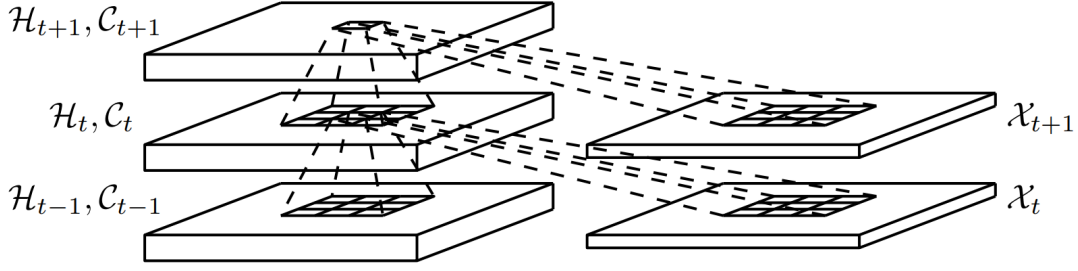


Figure 4.5: Structure of the convolutional LSTM cell [33]

4.4 Convolutional LSTM

Convolutional LSTM was introduced to address the problem of precipitation now-casting in [33]. They replaced the fully connected layers (matrix multiplications) with convolutions. The new architecture can be applied to tensors of any dimensions, because convolution is applicable to them. It is a major benefit, because the fully connected LSTM variant can only handle vectors as it's input making it difficult to use for tasks where the information is spatiotemporal. Similarly to LSTM we describe the fully convolutional cell in the following equations, where $*$ is the convolution operator and \odot is the Hadamard product. All the gate tensors have the same shape as the cell state.

The I and F are our input and forget gate-tensors.

$$\begin{aligned} I_t &= \sigma(W_{xi} * X_t + W_{hi} * H_{t-1} + W_{ci} \odot C_{t-1} + b_i) \\ F_t &= \sigma(W_{xf} * X_t + W_{hf} * H_{t-1} + W_{cf} \odot C_{t-1} + b_f) \end{aligned} \quad (4.5)$$

The next cell state tensor is calculated in every timestep as the following equation describes:

$$C_t = F_t \odot C_{t-1} + I_t \odot \tanh(W_{xc} * X_t + W_{hc} * H_{t-1} + b_c) \quad (4.6)$$

Then, the cell's output and output-gate tensor is calculated as:

$$\begin{aligned} O_t &= \sigma(W_{xo} * x_t + W_{ho} * H_{t-1} + W_{co} \odot C_{t-1} + b_o) \\ H_t &= O_t \odot \tanh(C_t) \end{aligned} \quad (4.7)$$

When we are creating the cell, we need to define the number of *output_channels*, because it defines the shape of the cell and the output. The applied convolutions

are always padded, so their output will always have the same number of items in one channel.

Using stacked architectures of this convolutional LSTM we are able to create models which preserve the spatiotemporal information, while the number of weights is quite low, because it does not contain any fully connected layers. It is also important to point out, that this cell is also capable of learning long-term time dependencies, which is a very important feature in natural language processing on character level.

Chapter 5

Models

In this chapter we introduce our new neural network models for morphological segmentation, and also provide the applied loss functions and gradient-based optimizer algorithms. We introduced new architectures for the task, which reached state-of-the-art performance in Hungarian morphological segmentation. The invention of these architectures was iterative, because we tried out newer and newer solutions to address this problem. The first models we experimented with were simple LSTM networks, to keep the architecture as plain as possible. However, we found out that wrapping the cells only around one direction on the character sentence is not enough to clearly determine the morpheme boundaries. That was the point we decided to use bidirectional recurrent networks. We also experimented with GRU architectures, but the performance of the different cells was almost the same. To keep the number of hyperparameters low, we continued the model research with only Long Short Term Memory variants.

The current baseline solution for this task used a convolutional network to map the features of the characters, and then applied a bidirectional GRU architecture on the convolution's output channels. During the development iterations we created new neural network architectures tailored for sentence tagging. Based on our previous work with dynamic models, we also focused on the native implementation of our networks, without any need for padding on the recurrent and convolutional networks.

None of the architectures contain language dependencies, which means if a proper dataset is given from a morpheme-rich language, the model can be easily trained and compared to other currently existing solutions. The models are also not bounded to existing Hungarian words, which means they can segment words which are properly put together according to language rules. As English is getting involved in every language due to the technological evolution, many Hungarian words have an English

etymon, like "playre", "filet". A rule-based solution may not recognize the morphemes correctly, while our models do not contain such bottlenecks.

During the evolution of these learning algorithms we parallelly developed a static and dynamic pair of them, to show the benefits of dynamic neural networks, against the more widely used static ones. As we experienced, with the help of new machine learning libraries it is getting easier to develop even difficult dynamic architectures, in which we apply weight sharing to save computational resources and avoid unoptimizable parameter spaces. As we were working with these frameworks, we found out it is much harder to write optimized code for a dynamically changing input and output size, which makes it logical to still use the padded static variants. Even the most advanced frameworks lack support on this field, due to its complexity. However, this is part of the cutting edge research and even Google chief scientists approved that¹ dynamic architectures will play a very important role in future natural language processing solutions. It also supposes the same, that Tensorflow has just added an Eager execution module, which will support dynamic computational graphs in the future.

5.1 Stacked bidirectional FC-LSTM with conv1D

This model is a bidirectional fully connected LSTM. The outputs of the two LSTM which are scanning the input sequence forward and backward are concatenated along the time dimension. This means that for every character we accumulate the features of its full environment. This recurrency make it possible to keep the number of parameters low, and make them focus on extracting and transferring the needed features to decide if the given character is the starting point of a new morpheme. This also makes it possible to transfer information from the beginning of the sentence to the end of the sentence and vice versa. With this help the task gets much easier to solve, because a much larger environment is defined, which means more information for the boundary decision.

On the concatenated output sequence we apply 1 dimensional convolutions along the time domain. We applied a kernel of variable sizes ranging from 1 to 9, and set the final output channels to 1. This means that we extracted and transferred the information with the recurrent architectures not only for close morphemes, but sentence wide. Then we applied multiple convolutional mapping on the closest characters with decreasing the kernel size. By making the kernel size smaller with the depth, we focused on the closest characters in the deepest layers, while having

¹At the AI Frontiers conference 2017

the information of the rest of the sentence too. We tried out multiple nonlinear functions, and we found that while there were only minor differences between *ReLU* and *sigmoid*. The best results were provided by *ReLU*, but we experimented with all the nonlinearities. The depth of our architecture made it possible to use sigmoid effectively. With growing number of layers the vanishing gradient problem makes it difficult to use the σ or *tanh* activation function, but it was not a problem in our case, because we constrained the model depth to at most 4 convolutional layers.

The outputs of the last convolutional layer were our logits. Then we used a sigmoid activation function to get the probabilities for a given character in a sequence. σ is often used for multi-label classification, and sentence tagging is analogical with that solution.

This neural network is static in a way that it always needs to have inputs from the same length. We truncated the sentences at 300 characters and words at 30 characters. The label distributions can be found in Table 3.2. After we have truncated the sentences and words, we padded them to the same length, so the size of the input tensor was always the same.

5.2 Dynamic stacked bi-FC-LSTM with conv1D

The model is the very similar to the first one, but in this architecture we used dynamic neural networks. The recurrent cell was folded on the input sequence from both directions. The we concatenated the two folded outputs along the time dimension. The we applied the same convolutional architecture as we did in the static stacked bidirectional LSTM architecture.

The main reason to applied dynamic networks is to avoid the unnecessary padding and to create a natural architecture for our goal. Many deep learning improvements come from simplifying the architectures and applying a more natural models to solve different tasks. By eliminating the padding we save a large amount of computational power, because the large difference between the average length of the sentences is only the 1/3 of the padded length. It means only 1/3 of the calculations are useful for the real output of our model. The network also have to learn how to deal with the paddings too. It is obviously an easy task for such complex models, but it still affects the finite memory of the networks. Other reason is the memory usage, because the large amount of paddings in every batch takes a lot of RAM and VRAM uselessly.

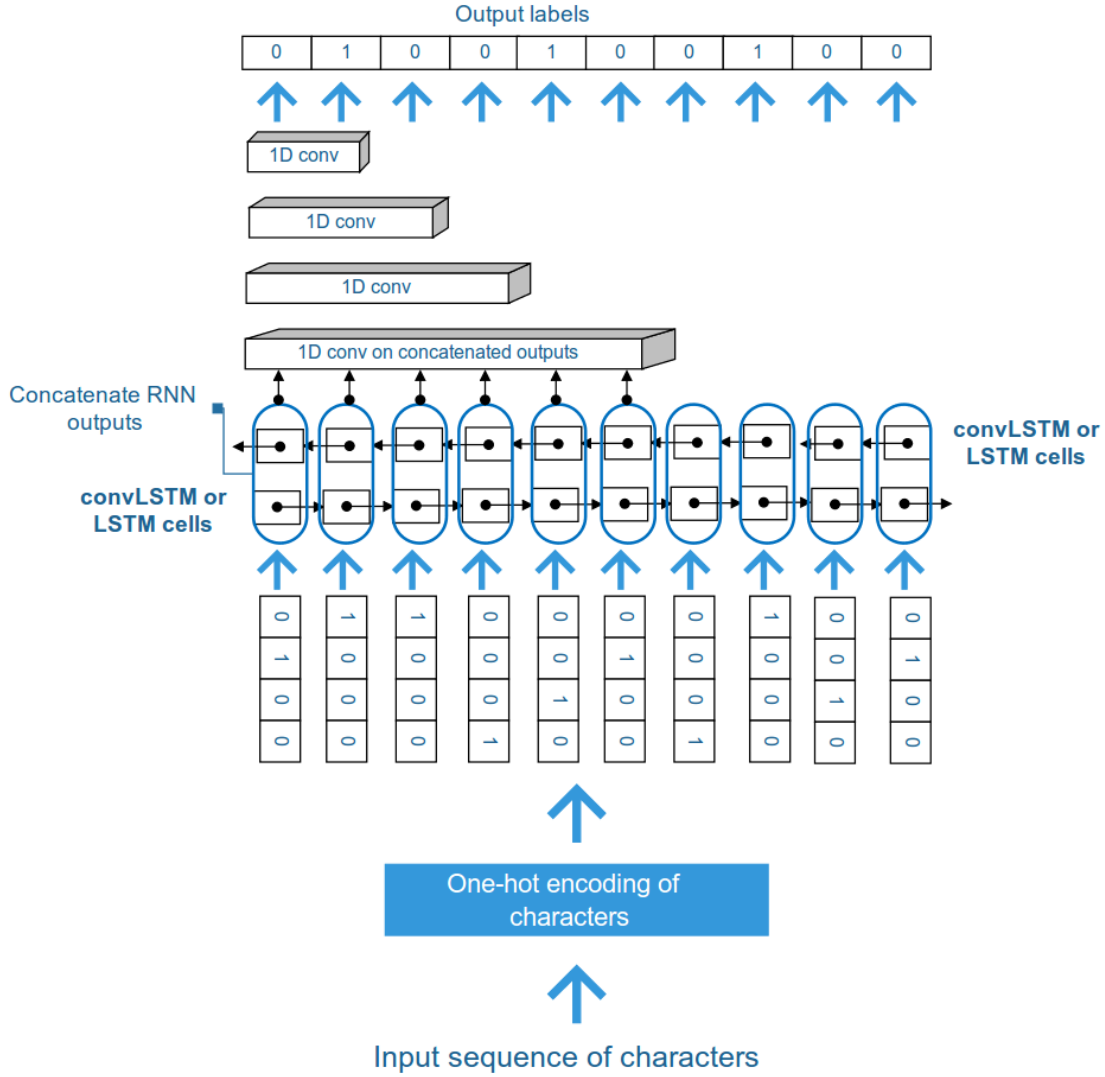


Figure 5.1: Architecture diagram of dynamic and static models: the cell type can be convLSTM and LSTM depending on the experiment type

The usage of dynamic batching also complicates the development, and debugging of the programs. All the dynamic NN frameworks are in experimental, early-beta phase, which means that they contain a lot of bugs, and they are not stable for long hyperparameter optimizations.

This model shares the weights similarly to the static network. The recurrent cell always has the same weights in a direction, but these weights are not shared between the forward and backward directions. This way there were no problems from concatenations and there was no need to create new variables in the dynamic computational loop, which would made the architecture much more complex.

5.3 Stacked fully convolutional biLSTM

This model is a dynamic neural network which uses stacked convolutional LSTM cells. The number of stacked layers was a hyperparameter when we were experimenting with the algorithms. We changed the kernel size during our experiments from 1 to the size of character vocabulary. We applied the convolutional LSTM cell, to be able to add more layers into our computational graph, while keeping the number of weights low, so the network would still remain trainable.

We created a static and dynamic version of this architecture, like we did above. We were interested in the convergence differences between the padded and unpadded networks, and we also experimented with the learning capabilities of these different learning algorithms.

In the research phase we found that a convolutional layer can completely replicate a fully-connected layer by making the kernel size as large as the input tensor. This way every output neuron will receive information of the full previous layer. It means that in our case, when we set the conv kernel size to the size of our vocabulary, we created a fully-connected layer, and kept the tensor's first dimension as the same size, and added multiple channels. This method is used in new fully convolutional architectures, where the researchers do not apply any fully-connected layers [23]. The most important benefit of this solution is the spatial size keeping, because this way, we can simply add multiple convolutional layers to this architecture, which would need reshaping if we applied fully-connected layers.

We needed to implement the convLSTM cell, because the Tensorflow version, which was compatible with Fold, did not contain this kind of cell. The cell is implemented similarly to the fully connected LSTM cell, using the recurrent neural network library. Most of the library's high level modules are implemented in python, so we didn't introduce any performance drawbacks compared to other commonly used cells, such as GRU, or basic LSTM. In the dynamic implementation we need to wrap the cell in a computational block, which can be used by the Fold ecosystem. These blocks are then folded from two directions on the same sentences, and their results can be concatenated.

5.4 Losses

We experimented with the loss functions too. In NN development, the choice of the loss function is very important, because it needs to represent the proper loss to us. This loss function also needs to be differentiable, and stable to outliers from our

dataset. In most of the cases, only a subset of the commonly used loss functions works properly, because it's highly dependent from the problem we are trying to solve.

The *labels* vector contains ones and zeros, which are marking the morpheme boundaries. The length of this vector can be the static padding size when we are using static computational graphs. When we are using dynamic computational graphs, this length always equals the input sequence's length in the sentence trainings, and the input word's length in word trainings.

The *logits* vector contains the outputs of our neural network. Our last layer does not have an activation function, instead the activation is applied later, to make it possible to use different loss functions, and nonlinearities for decision.

We always used the sigmoid ($\sigma()$) function to create the label probabilities of the *logits* vector as the following:

$$probabilities = \sigma(logits) \quad (5.1)$$

5.4.1 Least absolute deviations

L1 loss is defined as:

$$L1_loss = \sum_{i=0}^{N-1} |labels_i - probabilities_i| \quad (5.2)$$

N marks the number of elements in the output and labels vector. The loss must be differentiable to backpropagate the errors we calculated. It is not differentiable in 0, so if the loss equals 0, then the derivatives will not be calculated and it will cause an error.

5.4.2 Least square errors

L2 loss is defined as:

$$L2_loss = \sum_{i=0}^{N-1} (labels_i - probabilities_i)^2 \quad (5.3)$$

This loss is differentiable everywhere, but it is less robust against outliers. Small number of outliers can cause large differences when using this loss function, because every error is squared before summation.

5.4.3 Cross Entropy

For multi-label classification it is very commonly used loss function, because it handles the probabilities well. Sequence tagging can be considered as a multi-label classification so we experimented with it too. The logistic loss can be calculated via the following formula:

$$cross_entropy = labels \cdot (-1) \log(\sigma(logits)) + (1 - labels) \cdot (-1) \log(1 - \sigma(logits)) \quad (5.4)$$

As it can be seen on the function above, the loss may be unstable in specific cases. To ensure stability we used the following reformulation of the original function:

$$cross_entropy = \max(logits, 0) - logits \cdot labels + \log(1 + e^{-|logits|}) \quad (5.5)$$

5.5 Optimizers

We experimented with many gradient-based optimizers, here we listed and defined the best performing algorithms related to our highly recurrent neural network optimization task. After a wide research we selected only the suitable algorithms for our needs. The currently used optimizers are collected in [31], with brief descriptions and use cases.

5.5.1 Gradient Descent

Gradient Descent is one of the most commonly used first-order gradient-based iterative optimizing algorithms. The goal of the algorithm is to approximate the global minimum, by searching for local minimums. It means that the algorithm usually get stuck in local minimums. One update of the algorithm is defined as the following:

$$\begin{aligned} \Delta\theta &= -\nabla_{\theta} J(\theta) \\ \theta &= \theta + \eta \cdot \Delta\theta \end{aligned} \quad (5.6)$$

In the equation above θ marks the parameters of the function, η stands for the learning rate hyperparameter. Parameters are meaning the weights in the neural network and J is the cost function we partially differentiate and back-propagate to every weight using the chain rule.

GD has multiple variants considering batch size. When the full training data is fed into the network, and loss is calculated, the optimizer is called GD. When we feed the data in batches, the algorithm is named mini-batch gradient descent (BGD). When only one training example is fed at a time, the method is called stochastic gradient descent (SGD).

In our experiments we used the batch gradient descent algorithm variants, because the full dataset would not fit into memory, but batched calculations usually speed up computations on highly parallelized architectures, such as GPUs [31].

5.5.2 RMSProp

RMSProp is a commonly used optimizer for training recurrent NN models. The following equations describe how we calculate running average for scaling the gradients (E), and how we update each parameter during the training.

$$\begin{aligned} g_{t,i} &= \nabla_{\theta} J(\theta_{t,i}) \\ E[g^2]_t &= 0.9E[g^2]_{t-1} + 0.1g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \end{aligned} \tag{5.7}$$

ϵ is a very small number to ensure numerical stability by avoiding division by zero. The starting learning rate η is a hyperparameter and usually $0.01 > \eta > 0.0001$. θ is marking the weights. This scaling makes the convergence faster and more stable, and this is why this optimizer is widely used for LSTMs and GRUs.

5.5.3 Adam

Adaptive moment estimation [17] is one of the most complicated gradient-based learning algorithms. It stores an exponentially decaying average of the previous

gradients, and squared gradients as the following:

$$\begin{aligned}
g_{t,i} &= \nabla_{\theta} J(\theta_{t,i}) \\
m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1 - \beta_2^t}
\end{aligned} \tag{5.8}$$

So m_t and v_t try to estimate the first and second moment for every variable's gradients. \hat{m}_t and \hat{v}_t are the bias corrected versions of the estimations. β_1^t and β_2^t are usually chosen to be close to 1, a common setting for them is $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

The weight update is defined as follows:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{5.9}$$

ϵ is only for numerical stability and is usually very small, in the magnitude of 10^{-7} .

Adam is the most widely used optimizer because of its very good convergence capabilities. The calculations are more expensive then in the case of a simple SGD, but the added value of these exponential averages makes it logical to use this optimizer.

Chapter 6

Experimental Setup

The development speed of neural network systems heavily relies on the supporting framework and computational power. Recently most AI researchers are developing their new architectures using a huge amount of computational resources. In the beginning of the new deep learning era scientist used servers with thousands of CPUs to train their networks for tasks like classification and regression. The recognition of GPUs as compute nodes for deep learning boosted the whole research field, because anyone with a high-end GPU could teach artificial neural networks at home. Large companies tend to create specific hardware for only deep learning applications. Apple already introduced neural co-processors in the iPhone to support their personal assistant's networks. Google created a custom tensor processing unit (TPU), which is designed to handle enormous training and inference loads. Nvidia is also working on large server and mobile solutions too. The supporting framework, including the operating system, machine learning framework and programming environment are also crucial to reach the goal of the research.

6.1 Computers

Because of the large number of experiments we needed to perform, we utilized more GPUs and more computers at the same time. Google researchers did the same in 2016 for using reinforcement learning to create neural network architectures by itself [43]. The group used a cluster of 800 GPUs, which were parallelly training different neural network architectures. With this solution they achieved state-of-the-art performance and their cell (NAS cell) outperformed LSTM in many use cases. Andrew Ng also stated that in the modern deep learning era, the more computational

power you have, the faster your iteration will be, which is especially helpful in deep learning.

Considering the amount of our training data, and the problem's complexity, we trained the network on 2xNvidia Titan Xp, and 4xGTX 980 GPUs. The provided peak performance was 42 Tflops/sec, which was enough to experiment with all the upcoming ideas. The data server was connected with them directly, and to avoid bandwidth problems, the full database was loaded into RAM before we started training the networks. With this speed gain the algorithms utilized the GPU to 100%.

6.2 Software Stack

As mentioned in Chapter 3, the data type needs architectures which are able to adapt to sentences of indeterminate length. We compared the different machine learning libraries below in order to show their capabilities and drawbacks. Our final chosen framework was Tensorflow and Fold, but we experimented with other libraries too.

6.2.1 Tensorflow

As of 2017/Q4 Tensorflow is the leading machine learning framework for research and production. The library is maintained and supported by Google, it is also the most popular machine learning project on Github. It supports multi GPU training in distributed systems and most of the neural network models. The library is currently under heavy development, updates are surfacing every day.

Tensorflow is working with static computation directed acyclic graphs (DAG). The first step is to define the graph, which will defines the computation. After designing the forward pass of the model, the framework compiles it, and builds up the graph. It automatically calculates the gradients and contains many gradient based optimizers out of the box.

The drawback of the static computational approach is making it difficult to define sequence-based models, especially for sequences with a length of high standard deviance. For the static graph we need to pad the inputs to a predefined length, and truncate all the longer sequences to this length. It means the network will always receive a tensor from the same shape. Considering our data which has a varying length, it would make our network to iterate mostly on pad characters, and the

memory usage would be unnecessarily high, because the input tensor shape would be always `[batch_size, max_len, vocabulary_size]`. It is also an unnatural view of the data to always pad it to the same length, while the recurrent neural networks would allow us to work on arbitrary length sentences.

To overcome this error, researchers use a method called bucketing [16]. With this solution they create more static graphs, with predefined sentence lengths, for example `[5,10,20,30,50]` and in run-time the interpreter decides which static graph to use. This still contains memory, time and computational overhead, but still much more effective than the basic solution.

Despite all the mentioned problems related to static computational graphs, they are the most common these days. The main reason for that is related to the hardware behind deep learning. Learning algorithms usually run on highly parallelized hardware, such as GPUs. These have dedicated memory and work as a streaming multiprocessor. The newest hardware, such as the Nvidia Tesla V100 [22] is already using tensor-cores to speed up deep learning computations. In dynamic computational graph it's more difficult to utilize, and there will be always an overhead because of the conditionals, concatenations and inner loops.

We chose Tensorflow for implementing the static networks in this thesis.

6.2.2 PyTorch

PyTorch is the machine learning framework which is the closest to pure Python code. It contains automatically calculated gradients (autograd), and supports dynamic neural networks out of the box. Torch is originally implemented in Lua, but Python is more known these days, so they ported it to Python 2 and 3 too. It's one of the easiest framework to use, because the full training pipeline happens on Python-level. So it's not building up the computational graph and feeds the data, but the Python interpreter which goes in loops, and can even take conditions too. The tensor operations and gradient calculations are supported to run on the GPU, but it does not support distributed computations yet.

PyTorch as described above is mainly used for experimenting with new architectures, and researching neural network models in quick iterations. Implementing in this framework is fast compared to other frameworks, and handles the tensor operations naturally on the GPU or CPU depending on your choice.

While it supports dynamic neural networks, usually machine learning engineers still apply padding, because with mini-batch gradient descent [18], the resources can be

utilized to 100%, and also the convergence could be more stable using multiple input samples at the same time. Currently PyTorch does not support dynamic batching, so these sequences need to be truncated and padded to the same length for the network. This still contains unnecessary computations, but with clever learning techniques these could be reduced. One of the most simple ideas is to always batch together sequences of similar length, so as to minimize the padding.

6.2.3 Tensorflow Fold

Tensorflow Fold is part of the Tensorflow repository¹, and is in highly experimental phase in 2017 Q4. It is the first library to support dynamic batching [24], which means the sequences in the batch do not need to have the same length. This totally eliminates the padding during training and inference. It reduces the computational cost, and also speeds up the training and preprocessing part. This library is supported by Google developers who were very helpful when we were experimenting with the framework.

The library uses a low-level C++ API called loom. With the help of the added features, it can transform arbitrarily shaped DAGs to create a static computational graph. It mainly uses Tensorflow's while loop, gather and concat functions during the computations.

Dynamic batching also creates an overhead, because it needs to concatenate and gather the results from the computations. Using recurrent or tree-LSTM variants make it difficult to parallelize the computational graph. These networks are usually very long, like in our case the longest sequence contained 500 items. Because every calculation need to know the previous results, the full computation will contain 500 folds in this case for a single sequence, where only the cells can be parallelized, not the architecture. When we applied a batch size of 20, the length of the longest sequence in our batch determined the length of the training and inference time. The concat and gather ops were not fully prepared and optimized for dynamic computations. Where the given sentence was already calculated, the but there was a longer sentence in the batch, the computations and gathers continued with a batch size of zero. Zero batch size means that there should be no element in the input tensor, but not all the computational nodes support this kind of interpretation. We needed to modify the convolutional operations of Tensorflow in order to use them with dynamic graphs. None of the convolution operators supported zero batch sizes, so we created a condition to bypass these calculations and provide a zero tensor with

¹<https://github.com/tensorflow/fold>

a batch size of zero. This solution solved the problem and made it possible to use not only fully-connected layers, but it clearly introduced a computational overhead, because creating and allocating new tensors was an important cost. It also took time in the graph to use the tf conditional operators, because they do not apply lazy-semantics, so both of the branches are precalculated by default, and only the proper one is forwarded.

We used Tensorflow Fold library to implement the dynamic neural networks in this thesis.

6.2.4 Other libraries

Other libraries like Keras, Theano, Torch, etc. were out of the scope, because in this thesis we focused on the most popular frameworks in deep learning. Keras and Theano does not support dynamic neural networks, so their relevance is low for this task, and Keras would not allow us to reach deeper level functions in the backend.

For other dynamic approaches the DyNet[25] neural network toolkit is a commonly used for research and new architecture development. For further improvements and development it seems to be a good direction, because of its research oriented and continuously developed NLP oriented framework.

6.3 Environment, Train Helpers

The software environment we used is very complex, because Fold is currently in experimental phase, and it needs to be built from sources. For this we needed to use Google's Bazel² build system. We used Python 3.4 and 3.5 on different computers and there were no compatibility problems. For using multiple versions of Python related packets we used virtual environments, to separate us from the core interpreters. All the experiments ran on Ubuntu linux version 16.04.

The neural networks were trained on GPUs using Nvidia CUDA 8 and cudNN v6. These libraries helped speeding up the training procedure. These libraries also speed up the development of NN architectures, because the basic layer prototypes, nonlinearities and memory copies are implemented and optimized for most of the modern GPU architectures.

²<https://bazel.build>

6.3.1 Hyperparameter optimization

During the training phase we applied an early stopping [28] system, which monitored the validation dataset loss. If this loss started to go up, the daemon started to log that there may be overfitting. After a patience period, this daemon stopped the training. With this solution we saved computational resources, and this way we were able to run more experiments. It was very important to avoid overfitting [42] the training dataset to 100% accuracy, while the cross validation results are getting worse and worse.

To find the optimal hyperparameters we created a Python orchestrator, which runs multiple experiments on multiple computers at the same time.

To run multiple Tensorflow experiments at the same time we needed to use separate threads. The currently existing hyperparameter optimization solutions use bash scripts or Python scripts to execute numerous experiments with different run configurations. In Python the global interpreter lock (GIL) makes it difficult to utilize multi-threaded applications. The solution for this is usually multiprocessing. When we are using the multiprocessing module, the GIL is not a problem anymore, it is only blocking th threads. Because our problem did not need any communication between these different processes, there was no overhead because of this solution. By default we were running the experiments on GPUs, but this also gave us a constraint when using multiple workers on the same computer at the same time. The VRAM of the GPUs is usually a bottleneck, and we needed to reduce the batch size to fit multiple models at the same time on the video cards. We discover another problem, which was caused by Nvidia’s scheduler. The newer Pascal architecture supports preemptive scheduling, so on the Titan X cards it was faster to train multiple train at the same time. On the other hand, the older Maxwell architecture (GTX980) does not support preemptive scheduling, and this scheduler is much slower for multiple tasks at the same time. This was the reason that learning algorithms were much slower when researchers were using the same computer for training and running their operating system’s UI. The newer architectures benefit from the new scheduler, but on the older GPUs we were running only one experiment at a time, to avoid long waiting times.

During the last semester of university studies, we developed a solution³ for training neural networks with different hyperparameters. The models were different from this one, but the optimization algorithm can be used for different problems, but needs a proper description of the hyperparameter space. The created optimizer was

³Code and documentation is available at: <https://github.com/evelkey/vahun>

a genetic algorithm (evolutionary algorithm), which created a population of neural networks. After the population was created, all of these networks were trained, and evaluated on the test and cross validation dataset. From these evaluations we extracted a single score to define the performance of a network. Then we selected some of the best performing architectures and kept them for the next population. We also added new random elements from the hyperparameter space. Other fraction of the new population consisted of descendants of the best performing ones and mutations from the previous population’s individuals. For inheritance we created random couples in the population, and created new items by combining their layers and hyperparameters following predefined rules. the mutation affected only a small part of the population and was done in every generation in order to increase the diversity of the model parameters in the population. The algorithm converged quickly for the best performing hyperparameters, and needed much less calculations than exhaustive search on the parameters.

Optimizing the hyperparameters of recurrent networks is a very difficult task, because the complexity of the problem is much larger. Considering a simple fully connected layer, the descriptor of the architecture is very easy, can be even a single list of numbers, which show the number of neurons in each layer if we apply the same nonlinearity. When we apply convolutions, the hyperparameter space grows too, we need to define the kernel size, nonlinearity and output channel count for each layer. With the introduction of LSTM cells, we need to define more attributes, like hidden state size, interconnections, etc.

Our architectures are much more difficult then the simple building blocks we described above. We apply convolutions on the output of bidirectionally stacked recurrent cells, and even the cell types and sizes are different in every model. This make it much more difficult to find the best settings for a given architecture. Most of the recurrent architectures are even difficult to optimize, and their behaviour is more dependent on their weight initialization. The dependency is logical, because we apply the same weights a few hundred times in a row. While LSTMs successfully solve the problem of vanishing gradient, the convergence and training is still more difficult then in case of a feed-forward simple network.

We tried out 3 methods to optimize the hyperparameters: exhaustive search, random search, and genetic algorithm. Exhaustive search is the most stable solution, because it is not dependent of any random sampling on the hyperparameter side. The huge reason against it is the evaluation time of the search. Considering our case using exhaustive search for only a subset of the chosen hyperparameters would have taken months to evaluate. The experiments with the genetic algorithms seemed

very promising, but we found out that the algorithm fails badly in many cases, depending on the first population. The first population's individuals are sampled from the hyperparameter options assuming uniform distribution of them. It is very difficult to converge for the genetic algorithm, because the number of stacked LSTMs is also a parameter along with the number of neurons in each layer. After a few experiments we decided to not use this optimal parameter search solution, because of its instability.

The final hyperparameter optimizer used random search as its core parameter selection method, and showed very good results by mapping a large subset of the parameter space. The module is capable of creating arbitrary number of LSTM cells stacked upon each other and apply arbitrary number of convolutional layers on their concatenated output. The same method is used for sentence and word problems, but with different parameter options.

We created logs for every run by the logger and separately in a hyperopt logfile, which is a tsv for all the different experiments. This log contains the automatically evaluated metrics on the test dataset. By only reading this log we can select which model is the best performing.

6.3.2 Data pipeline

Data pipeline is one of the most difficult parts during the development phase. It needs to be as fast as possible, while it needs to support the different input needs of the architectures. Usually when researchers are training on large datasets, when the memory is too large to fit into memory, they are using pipelines to read, convert and preprocess their dataset. As we are moving to larger computational power these steps needs to be faster and faster. The current best practice in this field is to use queues and multiprocessing. This way the data processing and neural network training is asynchronous, and this way much faster too. The data pipeline can run on multiple cores, or even on multiple computers in large clusters. The same applies for the training, it can also run on different worker nodes, speeding up the whole system.

For Tensorflow there are many different ways to feed the data. The most simple one is to use feed dictionaries, which feed data to the already compiled TF model's nodes called placeholders. The placeholders have predefined shape and data type, so this feeding is typesafe. The drawback of this solution is it's speed. We need to serialize the feed dictionary and pass it to the graph.

The current best practice does not use feed dictionary, it is more for experimenting with new architectures. The best way currently is to use the already mentioned asynchronous data processing mechanism. The newest Tensorflow is already able to cache the processed data to the GPU, so the communication speed between the CPU and the graphics card is not a bottleneck with this solution.

The Tensorflow Fold library has a different feeding mechanism, it also support the queues, but as there may be sentences of very different length in our dataset, the shapes cannot be the same. That's why the developers introduced a new feeding mechanism called compiled feed dictionary. It is very similar to the original one, but it supports the different shapes and can be compiled on different threads, so it can also speed up the computations by eliminating idle time.

For our experiments we were using feed dictionaries and precompiled feeds, because our data fitted into memory, and there was no major speed difference considering other solutions.

6.3.3 Saving and logging

Saving and logging are very important features during the NN development. Researchers study the learning curves of the models and based on that create better and better architectures for machine learning tasks. We logged all the evaluation metrics for the train, test and validation datasets. With the help of these logs we were able to find the best performing models and iteratively develop new ones based on others performance.

During the training phase we log every 10th batch run's results and automatically evaluate our predefined metrics on them. All these metrics are saved to the log file too. We run cross-validation checks on our validation set twice per epoch, and save the evaluated metrics to the log.

When we start the training we save the meta graph, which describes to full DAG of the NN architecture, which contains the full graph definition. We save the trained variables at every cross validation step in order to select best performing snapshot of the weights in the future. After the training is stopped by the early stopper, or run out of training samples we save all of the weights for the last time. We can reconstruct these models anytime if we have these files, which are stored in the same folder as the logs. This way we can easily select the best models and load them for further training or inference. The meta graph and weights are saved as protocol buffers. These were introduced by Google to serialize large amount of data

for gRPC calls and to save them to files. It is very fast to save, load and interpret these buffers.

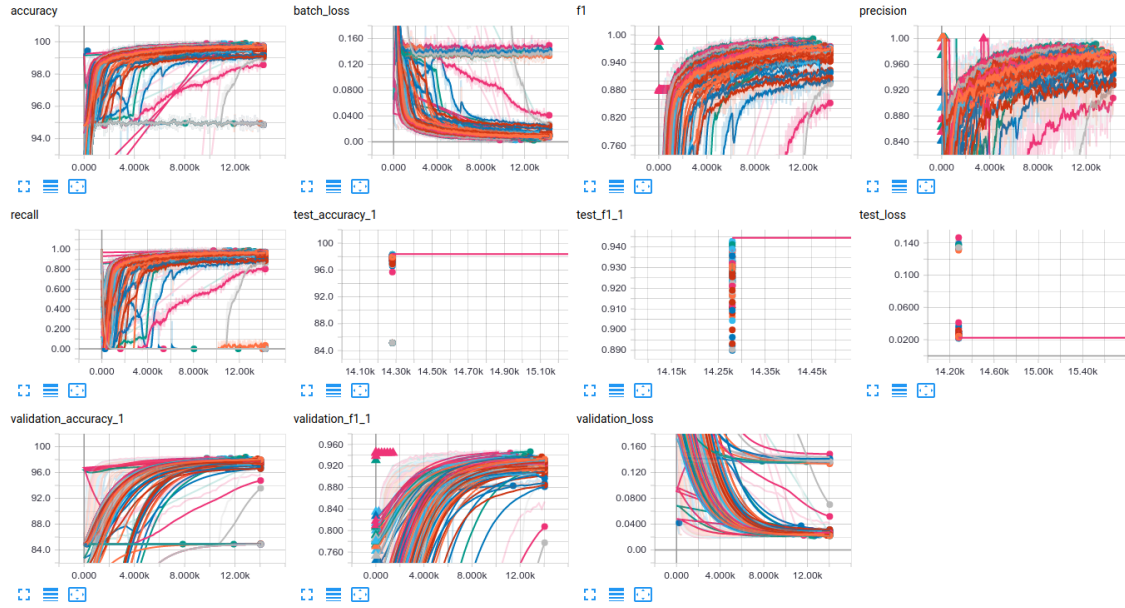


Figure 6.1: Tensorboard visualization tool for analysis

We used Tensorboard to visualize the logs we created while training the models. This tool can visualize not only the logs but also the neural network architecture and weight histograms too. These figures are extremely useful in the development phase for checking the computational graphs visually, and validating if the architecture does what we expect from it.

6.4 Evaluation metrics

To evaluate the performance of the different models and architectures, it was compulsory to define metrics which describe the differences of our models. The following metrics were automatically evaluated during training for the train batches. These metrics were also run on the separated cross-validation twice per epochs and also on the test dataset on the end of the training.

- Loss: We evaluate the loss function we use for training the model.
- Character-level accuracy: $\frac{\text{matched_labels}}{\text{number_of_labels}} \cdot 100$ in percent.
- Precision: $\frac{\text{true_pos}}{\text{true_pos} + \text{false_pos}}$
- Recall: $\frac{\text{true_pos}}{\text{true_pos} + \text{false_negative}}$

- F1 score: $2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$
- Word accuracy: $\frac{\text{correct_words}}{\text{number_of_words}} \cdot 100$ in percent. A word is correct only, if all the labels are correct in the word.
- Sentence accuracy: $\frac{\text{correct_sentences}}{\text{number_of_sentences}} \cdot 100$ in percent. A sentence is matched only, if all the labels are correct for that sentence.

The sentence accuracy is calculated only for sentence trainings, the others are evaluated during all the trainings. The logs contain these values and these can be read out for any given time-step from Tensorboard as it can be seen on Figure 6.2. With the help of these convergence characteristics we can define when the model started to overfit the training database.

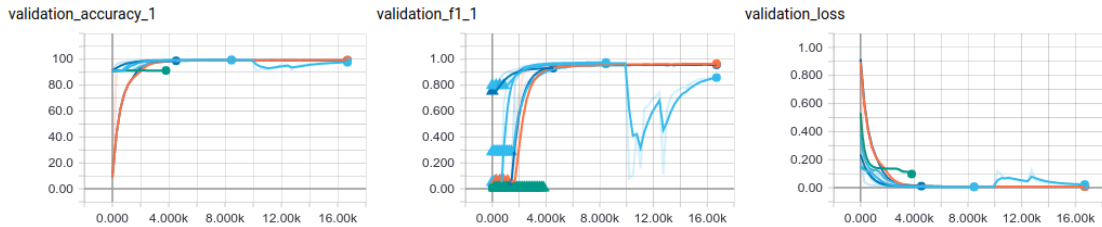


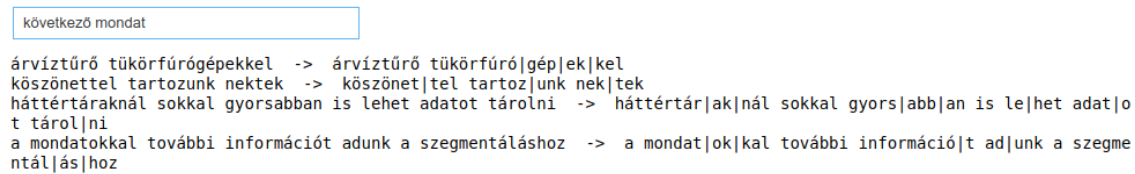
Figure 6.2: Automatically evaluated metrics

When we are developing NN solutions, it is very important to have a main metric, which is only one number and describes the model’s performance for our task. In our case, we decided to use word accuracy as our main metric. It means that when we were comparing two models the first and most important point was their accuracy on the previously unseen test dataset. We chose word accuracy, because words usually contain multiple morphemes, and this is a much more difficult problem to completely match a full word than to match a single character label.

6.5 Inference

To manually evaluate the segmentation results we created interactive inference IPython notebooks for word and sentence models⁴. In these notebooks one can load the already trained models and inference them with custom words and sentences. The models are loaded from the train log folders, which contain the protocol buffers of the meta graph and weights too at different checkpoints. After loading and interpreting the protocol buffer we restore the exact same DAG as we used for training.

⁴Notebooks can be found in the root of the Python module



```

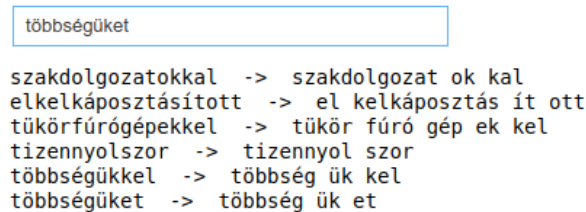
következő mondat

árvíztűrő tükörfúrógépekkel -> árvíztűrő tükörfúró|gép|ek|kel
köszönettel tartozunk nektek -> köszönet|tel tartozunk nek|tek
háttértáraknál sokkal gyorsabban is lehet adatot tárolni -> háttértár|ak|nál sokkal gyors|abban is le|het adat|o
t tárol|ni
a mondatokkal további információt adunk a szegmentáláshoz -> a mondat|ok|kal további információ|t ad|unk a szegme
ntál|ás|hoz

```

Figure 6.3: Sentence model inference interface

With this evaluation method we can find the models, which suit human sense more. Sometimes a model with less accuracy is better in human sense. Accuracy and loss give high penalty for false positives, but for humans it is often more logical to have more boundaries than less. We found that some models, which were not even among the best performing ones learned how to split compound words too.



```

többségüket

szakdolgozatokkal -> szakdolgozat ok kal
elkelkáposztásított -> el kelkáposztás ít ott
tükörfúrógépekkel -> tükör fúró gép ek kel
tizennyolcszor -> tizennyol szor
többségükkel -> többség ük kel
többségüket -> többség ük et

```

Figure 6.4: Word model inference interface

We created separate inference interfaces, because the models are different for these solutions and the data requires different preprocessing, and even the vocabularies of these models are different. In the repository we supplied our training data and vocabularies and models⁵ for training and inferencing new models. This inference notebook can be converted into scripts which use the command line interface with stdin and stdout. We created notebooks because of their easy usage and interactive interface with IPython widgets.

⁵The models can be found at <https://github.com/evelkey/dynamic-segmentation/tree/master/model>

Chapter 7

Experiments

In this chapter we present the difference between the defined neural networks and libraries. First of all we experimented with the losses to select the most suitable function for this sentence tagging task. The next experiments are comparing the dynamic and static neural networks, loss functions, optimizers and architectures we used to address morphological segmentation. The sequence of the experiments is important, because some of the experiments rely on others. We added everything ordered by execution sequence, so in this order the research is fully reproducible and straightforward.

We created an IPython notebook which automatically selects the best models with given datasets and architecture. The notebook can be found in the repository among the models and optimization logs.

7.1 Loss function selection

We defined the following losses: L1, L2 and sigmoid cross entropy. As it can be seen on Figure 7.1, the validation loss is decreasing when we train the model, but our precision and recall are very close to zero. The F1 score shows that the cross entropy loss models this problem much better than the other two.

We tested the losses with two different architectures to avoid being too model-specific. One model was a static fully connected bidirectional LSTM with conv-mapping for prediction, the other was a fully convolutional recurrent network. The figures show the performance on the first model, the fully convolutional had the exact same convergence characteristics.

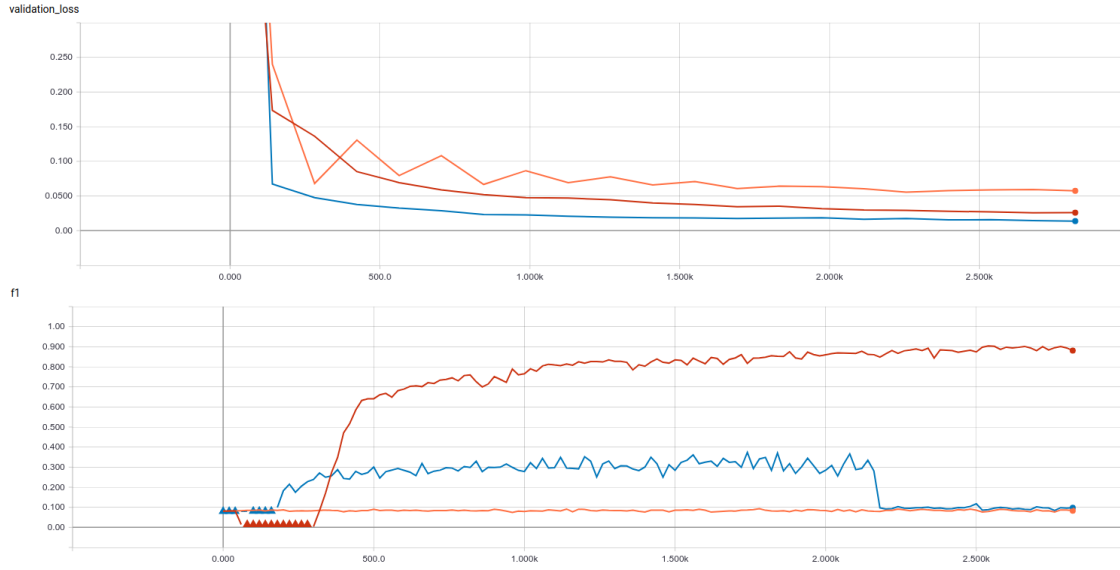


Figure 7.1: Loss function convergence analysis: (orange: l1_loss, blue: l2_loss, red:cross_entropy)

The loss function for further experiments and new models was cross entropy, because of it's superior performance compared to the other two.

7.2 Dynamic and static network comparison

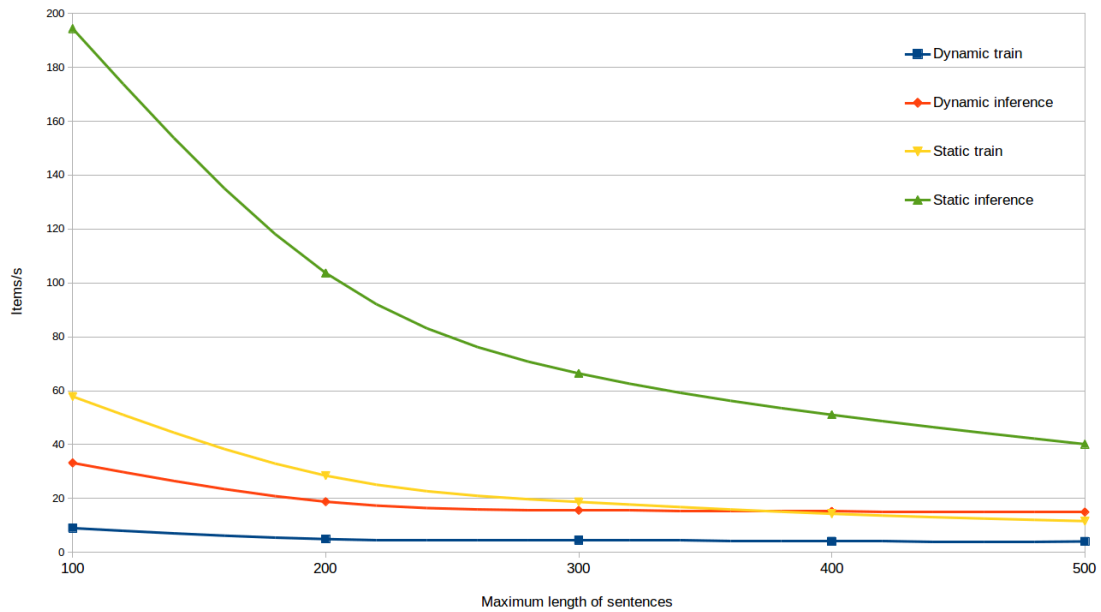


Figure 7.2: Train and inference speed analysis comparing dynamic and static environments

Table 7.1: Speed of static and dynamic network, measured in items/s

Max len(chars)	Dynamic train	Dynamic inference	Static train	Static inference
100	8.99	33.2	57.84	194.4
200	4.9382716049	18.8	28.48	103.68
300	4.469273743	15.6	18.72	66.4
400	4.1025641026	15.2	14.32	51.04
500	4.06	14.96	11.6	40.16

We compared the training and inferencing speed of the same architecture implemented with a static graph in Tensorflow and a dynamic graph in Tensorflow fold. The dynamic one has much less computations and should be faster in training and inference too. The experiments were run on a single Titan X GPU, and both solutions used the same batch size and hyperparameters. The model consisted of bidirectional fully connected LSTMs, with one 1D convolutional layer for predicting the labels.

The results are surprisingly showing that the static computational graph is much faster, even on very long sentences. All the sentences being truncated to 500 characters, the average padding we introduce is 390.1 chars per sentence. It means that 78% of the computations are just computing the paddings. As it can be seen on Table 7.1, the static graph utilizes the GPU much better. The reason of this is the highly vectorized architecture of the GPU. On a single CPU the dynamic graph should outperform the static one, while on the GPU we need to create parallellized programs. The many branches of calculations in the dynamic approach makes it slower on this architecture.

The convergence characteristics of the two architectures were very similar, so the padding did not introduce convergence difficulties to the static networks. This also means that the dynamic architecture does not profit from this benefit.

Because of the large differences, the static solution seems to be the most suitable for this task. We provided implementation in the code repository for the static and dynamic solutions as well. We used the dynamic networks only when it was needed because of the neural network architecture could not be implemented in a static DAG.

7.3 Optimizers and learning rate

As we defined in Section 5.5, we experimented with three different gradient-based optimizers. On Figure 5.5 the convergence characteristics are plotted for the dif-

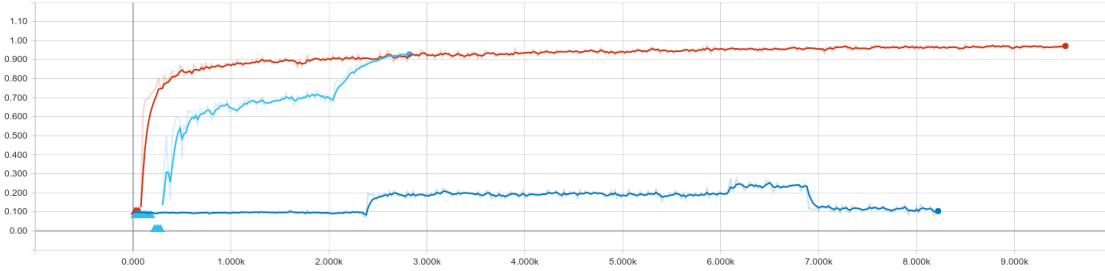


Figure 7.3: Optimizer convergence on F1 scores of the function of train steps (orange: Adam, light blue: RMSProp, dark blue: mini-batch GD)

Table 7.2: Hyperparameter options

parameter	options
recurrent cell	LSTM, convLSTM
stacked cell count	1, 2, 3, 4, 5
convLSTM channels	4, 8, 16, 32, 64
convLSTM kernel size	20, 30, 60, 90
LSTM unit size	64, 128, 256
conv kernel size	1, 3, 5, 7, 9
conv nonlinearity	ReLU, sigmoid at last layer
conv output channels	20, 80, 120, 200

ferent optimizers. All of these can be selected via the command line arguments in the training Python files for the experiments. The Adam showed the best performance based on multiple experiments on the same dataset with the same convLSTM models.

After defining the best optimizer solution for our task, the starting learning rate also needed to be selected. We applied exhaustive search for different models to define the best learning rate. The smoothest learning curve was provided with $\eta = 0.005$, while the learning diverged in most of the cases when we used $\eta > 0.03$.

For further experiments we applied Adam optimizer with $\eta = 0.005$, $\beta_1 = 0.9$, $\beta_2 = 0.999$.

7.4 Sentence experiments

The sentences were truncated to the length of 300 characters for this experiment. Padding was applied to this length for static architectures, while dynamic ones were fed with elements of arbitrary length. We run a hyperparameter optimization using the ranges defined in Table 7.2. Altogether 110 bidirectional LSTM with convolutional output experiments were run in the parameter search phase. We also ran 30

Table 7.3: Best sentence models with convolutional LSTMs

Type	cell sizes	c kernels	c channels	word%	char%	char F1
LSTM+conv	128,64,64	7,7,7,1	200,80,80,1	97.92	99.64	0.98
convLSTM	128,128,64	9,3,3	64,32,1	96.19	99.36	0.963
Baseline	256,256	-	-	82.73	-	0.927

conv-LSTM models with convolutional mapping. The best models can be found in Table 7.3. These experiments show much better results than the word experiments. The main reason for this is the much larger context we give for segmentation, and the different data distribution. The train time for this architecture was slightly longer than with the word-based models. These sequences are 10 times longer than the words, which means 10 times longer calculations in every batch. These experiments were run for 40 epochs on the dataset, but the early stopper usually stopped the training before it has finished all the epochs. The average training for one model took about 2 hours. This is the main reason we trained slightly fewer sentence models than word models.

In the experimenting phase we learned that the much faster word-based model architectures are performing similarly well with minor transformations on the sentence dataset too. We learned the behaviour of the cells on the fast word models, and it was not needed to try out many similar architectures on this sentence dataset too.

After training all the models during the hyperparameter optimization, we continued to improve the models by hand. These handcrafted models slightly outperformed the previous solutions, because we allowed more layers and more output channels for the convolutional layers. This way we made it possible for the model to learn more difficult features, and learn more connections in the dataset.

We already pointed out in Chapter 3, that the sentence data is unique on sentence level, while it isn't on word level. It means there is a major overlap between the test, train, validation dataset's words. It also introduces that the more frequent words, which are usually shorter and do not contain many morphemes. So a more biased predictor (to zero labels) is performing better in this case, and the infrequency of real morpheme boundaries is much lower.

Taking into account the mentioned problems, the results should be interpreted differently than in the case of unique word experiments. These results show that the network was able to learn the morphemes in sentence level, and we also evaluated the solution using unseen sentences with our inference notebook. The model performance is very promising, and using a much larger corpus could help us to create more independent evaluation datasets.

All the experiment logs have been added to the repository for sentence models, and the best model is available as a Tensorflow checkpoint, which can be loaded with the prepared inference notebooks.

7.5 Word experiments

The words were truncated to 30 characters for all the word experiments. The shorter words were padded to this length with zero-padding. During the hyperparameter optimization we trained and evaluated 120 fully convolutional bidirectional LSTMs, and 149 bidirectional LSTMs with convolutional mapping at their end. We added the best performing models to Table 7.4. The baseline is the first result for Hungarian segmentation using bidirectional GRU architectures [1]. The best hyperparameters are added to the tables and the models can be found in the repository. The train time was depending on the epoch count. A train cycle on one epoch took approximately *1 minute*. It was fast enough to run a wide hyperparameter search, with the options provided in Table 7.2. The experiments were run for at most 100 epochs, but in most of the cases the early stopper killed the trainings at about 40 epochs. An average training took about an hour to finish.

After the hyperparameter search finished, the results were evaluated using the inference notebooks on unseen words. We also experimented with much deeper networks, because the results were suggesting their success from the discovered hyperparameters. We created networks up to 10 layers of convolution and 5 stacked LSTMs. Too deep networks did not improve the performance on the test set, while completely fitting the train dataset to 99.99% on label-level accuracy. Using the ReLU nonlinearity before the last layer in the convolutional layers improved the test word accuracy with an average 2%.

When we were fine-tuning the networks we found that too much stacked LSTM cells decreased performance. The optimal number for stacked cells was 2 or 3 depending on the following layers. When we used too much LSTMs, the information passed through too many filters and some of the information got lost.

Interestingly the validation loss was going up while our model got better and better F1 scores on the validation dataset. It was because the cross entropy losses penalty for mislabeled items. As we introduced more and more positive labels, the probability for false positive predictions grew as well.

Table 7.4: Best word models with convolutional LSTMs

Type	cell sizes	c kernels	c channels	word%	char%	char F1
LSTM+conv	256,256	9,5,3, 3,3,3,1	120,64,64, 32,32,32,1	87.48	98.63	0.954
convLSTM	256,128,64,64	9,7,3,1	64,64,8,1	76.74	97.38	0.91
Baseline	256,256	-	-	82.73	-	0.927

The results suggest that more training data could help the model to generalize better. The best model’s word-level accuracy was 87.5%, which means that there is place for future improvements, which could be boosted using more data.

The training logs are available in the repository along with the best performing models. These can be loaded with the inference framework, and they can be used for real Hungarian word segmentation.

Chapter 8

Conclusion

After evaluating the experiments it was a surprise that dynamic networks with Tensorflow Fold are slow and there is no direct benefit from making recurrent networks dynamic. This suggests that we have to wait for proper dynamic algorithms until Tensorflow fully releases its new module called Eager to get rid of graph-bound computations with Google's framework. On the other hand the developers of the framework were very helpful, and they helped with the problems quickly. PyTorch seems to be a better solution for dynamic NN research and much faster for the iterative development too, because of its imperative execution system.

Our bidirectional LSTM with 1D convolution models outperformed the baseline by almost 5% on word accuracy. It means the research direction was proper we set up. To make the model even better it would be useful to have more training data to cover most of the morphemes during the training. As we saw in the best training results, the train F1 score almost reached one, while our test word accuracy remained lower than 90%. More data would also help our model to generalize better. We did not apply regularization in this task, because the results were showing no direct need for it, but with larger models it will be a necessary to make the models better too.

The convLSTM models, which were end-to-end convolutional, slightly underperformed the current baseline on word level predictions. This suggests that the fully connected LSTM variant is more suitable for this kind of task despite the much deeper architecture due to the multiple stacked layers. The results show that the other solution is the better for both sentence and word models.

The difference between sentence and word models is difficult to define based on the automatic metrics. The sentence models already seen many of the independent test words in the train dataset, because it's very difficult to create sentences without repeating words. On the direct evaluations the sentence models had much higher

accuracy, but these numbers are not directly comparable. We evaluated the results of the different models by hand, and found that their behaviour is very similar, and they have almost the same accuracy on unseen words. However, the sentence models contain much more information, and they can provide more accurate predictions because they can consider the environment of the individual words.

For future improvements in this field we would apply a semi-supervised training setup for the task. The first module would be an autoencoder to learn the most important features of our data (complete principal component analysis), and then we would use that as an embedding for our input. Using this solution this module could learn on any open web corpus, which may have billions of words to learn proper component analysis in an unsupervised manner. Then we could train a similar architecture to this one to label our input sentences and words morpheme boundaries. This is a much larger training scenario, called transfer learning [29]. The prediction part would be a stacked bidirectional fully-connected LSTM architecture with 1D convolutional output features, which is the introduced best solution.

Bibliography

- [1] Judit Ács. Hungarian morphological segmentation using recurrent and convolutional neural networks. 2017.
- [2] Sally Andrews, Brett Miller, and Keith Rayner. Eye movements and morphological segmentation of compound words: There is a mouse in mousetrap. *European Journal of Cognitive Psychology*, 16(1-2):285–311, 2004.
- [3] David Ascher, Paul F. Dubois, Konrad Hinsén, James Hugunin, and Travis Oliphant. *Numerical Python*. Lawrence Livermore National Laboratory, Livermore, CA, ucrl-ma-128569 edition, 1999.
- [4] Delphine Bernhard. Unsupervised morphological segmentation based on segment predictability and word segments alignment. In *Proceedings of 2nd Pascal Challenges Workshop*, pages 19–24, 2006.
- [5] Léon Bottou. *Large-Scale Machine Learning with Stochastic Gradient Descent*, pages 177–186. Physica-Verlag HD, Heidelberg, 2010.
- [6] Junyoung Chung, Çağlar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.
- [7] Mathias Creutz and Krista Lagus. *Unsupervised morpheme segmentation and morphology induction from text corpora using Morfessor 1.0*. Helsinki University of Technology, 2005.
- [8] Sajib Dasgupta and Vincent Ng. High-performance, language-independent morphological segmentation. In *HLT-NAACL*, pages 155–163, 2007.
- [9] Miryam de Lhoneux, Yan Shao, Ali Basirat, Eliyahu Kiperwasser, Sara Stymne, Yoav Goldberg, and Joakim Nivre. From raw text to universal dependencies-look, no tags! *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 207–217, 2017.

- [10] Vera Demberg. A language-independent unsupervised model for morphological segmentation. In *ANNUAL MEETING-ASSOCIATION FOR COMPUTATIONAL LINGUISTICS*, volume 45, page 920, 2007.
- [11] Greg Durrett and John DeNero. Supervised learning of complete morphological paradigms. In *HLT-NAACL*, pages 1185–1195, 2013.
- [12] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.
- [13] Alex Graves, Navdeep Jaitly, and Abdel-rahman Mohamed. Hybrid speech recognition with deep bidirectional lstm. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, pages 273–278. IEEE, 2013.
- [14] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 06(02):107–116, 1998.
- [15] Nal Kalchbrenner, Ivo Danihelka, and Alex Graves. Grid long short-term memory. *CoRR*, abs/1507.01526, 2015.
- [16] V. Khomenko, O. Shyshkov, O. Radyvonenko, and K. Bokhan. Accelerating recurrent neural network training using sequence bucketing and multi-gpu data parallelization. In *2016 IEEE First International Conference on Data Stream Mining Processing (DSMP)*, pages 100–103, Aug 2016.
- [17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.
- [18] Jakub Konečný, Jie Liu, Peter Richtárik, and Martin Takáč. Mini-batch semi-stochastic gradient descent in the proximal setting. *IEEE Journal of Selected Topics in Signal Processing*, 10(2):242–255, 2016.
- [19] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.
- [20] Young-Suk Lee, Kishore Papineni, Salim Roukos, Ossama Emam, and Hany Hassan. Language model based arabic word segmentation. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 399–406. Association for Computational Linguistics, 2003.
- [21] Krister Lindén, Miikka Silfverberg, and Tommi Pirinen. Hfst tools for morphology—an efficient open-source package for construction of morphological

- analyzers. In *International Workshop on Systems and Frameworks for Computational Morphology*, pages 28–47. Springer, 2009.
- [22] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2), 2008.
 - [23] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.
 - [24] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.
 - [25] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.
 - [26] Attila Novák, Katalin Gugán, Mónika Varga, and Adrienne Dömötör. Creation of an annotated corpus of old and middle hungarian court records and private correspondence. *Language Resources and Evaluation*, pages 1–28.
 - [27] Attila Novák, Borbála Siklósi, and Csaba Oravecz. A new integrated open-source morphological analyzer for hungarian. In *LREC*, 2016.
 - [28] Lutz Prechelt. Automatic early stopping using cross validation: quantifying the criteria. *Neural Networks*, 11(4):761–767, 1998.
 - [29] Rajat Raina, Alexis Battle, Honglak Lee, Benjamin Packer, and Andrew Y Ng. Self-taught learning: transfer learning from unlabeled data. In *Proceedings of the 24th international conference on Machine learning*, pages 759–766. ACM, 2007.
 - [30] Jason Riesa and David Yarowsky. Minimally supervised morphological segmentation with applications to machine translation. In *Proceedings of the 7th Conference of the Association for Machine Translation in the Americas (AMTA06)*, pages 185–192, 2006.
 - [31] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

- [32] Teemu Ruokolainen, Oskar Kohonen, Sami Virpioja, and Mikko Kurimo. Supervised morphological segmentation in a low-resource learning setting using conditional random fields. In *CoNLL*, pages 29–37, 2013.
- [33] Xingjian SHI, Zhouong Chen, Hao Wang, Dit-Yan Yeung, Wai-kin Wong, and Wang-chun WOO. Convolutional lstm network: A machine learning approach for precipitation nowcasting. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 802–810. Curran Associates, Inc., 2015.
- [34] Kairit Sirts and Sharon Goldwater. Minimally-supervised morphological segmentation using adaptor grammars. *Transactions of the Association for Computational Linguistics*, 1:255–266, 2013.
- [35] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Thirteenth Annual Conference of the International Speech Communication Association*, 2012.
- [36] Antal Van den Bosch and Walter Daelemans. Memory-based morphological analysis. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, pages 285–292. Association for Computational Linguistics, 1999.
- [37] Veronika Vincze, Viktor Varga, Katalin Ilona Simkó, János Zsibrita, Ágoston Nagy, Richárd Farkas, and János Csirik. Szeged corpus 2.5: Morphological modifications in a manually pos-tagged hungarian corpus. 2014.
- [38] Sami Virpioja, Peter Smit, Stig-Arne Grönroos, Mikko Kurimo, et al. Morfessor 2.0: Python implementation and extensions for morfessor baseline. 2013.
- [39] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.
- [40] Nianwen Xue et al. Chinese word segmentation as character tagging. *Computational Linguistics and Chinese Language Processing*, 8(1):29–48, 2003.
- [41] Daniel Zeman, Martin Popel, Milan Straka, Jan Hajic, Joakim Nivre, Filip Ginter, Juhani Luotolahti, Sampo Pyysalo, Slav Petrov, Martin Potthast, et al. Conll 2017 shared task: multilingual parsing from raw text to universal dependencies. *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 1–19, 2017.

- [42] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *CoRR*, abs/1611.03530, 2016.
- [43] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.