CLCL (Geneva) DINN Parser: a Neural Network Dependency Parser Ten Years Later

Christophe Moor
University of Geneva
Christophe.Moor@etu.unige.ch

Paola Merlo
University of Geneva
Paola.Merlo@unige.ch

James Henderson
XRCE / University of Geneva
James.Henderson@unige.ch

Haozhou Wang
University of Geneva
Haozhou.Wang@unige.ch

Abstract

This paper describes the University of Geneva's submission to the CoNLL 2017 shared task Multilingual Parsing from Raw Text to Universal Dependencies (listed as the CLCL (Geneva) entry). Our submitted parsing system is the grandchild of the first transition-based neural network dependency parser, which was the University of Geneva's entry in the CoNLL 2007 multilingual dependency parsing shared task, with some improvements to speed and portability. These results provide a baseline for investigating how far we have come in the past ten years of work on neural network dependency parsing.

1 Introduction

The system described in this paper is the grandchild of the first transition-based neural network dependency parser (Titov and Henderson, 2007b), which was the University of Geneva's entry in the CoNLL 2007 multilingual dependency parsing shared task (Titov and Henderson, 2007a). The system has undergone some developments and modifications, in particular the faster discriminative version introduced by Yazdani and Henderson (2015), but in many respects the design and implementation of this parser is unchanged since 2007. One of our motivations for our submission to this CoNLL 2017 multilingual dependency parsing shared task is to provide a baseline to evaluate to what extent recent advances in neural network models and training do in fact improve performance over "traditional" recurrent neural networks. We are listed in the table of results as the CLCL (Geneva) entry.

As with previous work using the Incremental Neural Network architecture (e.g. Henderson, 2003), the main philosophy of our submission is that we build language universal inductive biases into the model structure of the recurrent neural network, but we do not do any feature engineering. Training the neural network induces language-specific hidden representations automatically. To provide such a baseline, we use UDPipe for all pre-processing (Straka et al., 2016), and Malt Parser for all projectivisation (Nivre et al., 2006). The only exception is our strategy for surprise languages, discussed below.

These goals match well the aim of the 2017 Universal dependencies shared task, described in the introductory overview (Zeman et al., 2017). This task makes true cross-linguistic comparison possible thanks to the universal dependency annotation project, which underlies the data used in this shared task. We train exactly the same parsing model on every language, thereby allowing further comparisons. In addition, the feature induction abilities of the recurrent neural network help minimise any remaining cross-lingual differences due to pre-processing or annotation.

2 Data

We use only the provided treebanks. For large treebanks, we train the model on the UD treebank (Nivre et al., 2017a), with some tuning of meta-parameter using the development set.

For surprise languages, we train on the concatenation of the treebank for the language, no matter how small, and the treebank of an identified source language with a larger treebank. In post-testing experiments, we also apply this same strategy to other small treebanks, resulting in substantial improvements (average 43% better) over the submitted results.

We don't use externally trained word embeddings (we trained our own internally to the parser) or any other data resource.

3 Preprocessing

Tokenisation, word and sentence segmentation is provided by UD pipe (Straka et al., 2016). We do not use the morphological transducers from Apertium/Giellatekno that had been made available for the shared task.

Because our parser can only produce projective dependency trees, we apply the projectivisation transformation of the Malt parser package (Nivre et al., 2006) to all treebanks before training.

4 Parser

We apply a single DINN parser to each language. We do not use any ensemble methods. This makes our results more useful for comparison, and allows our model to be used within an ensemble with other parsers.

We use the parser described in Yazdani and Henderson (2015), the Discriminative Incremental Neural Network parser (DINN). Like the previous version of this parser (Titov and Henderson, 2007b), it uses a recurrent neural network (RNN) to predict the actions for a fast shift-reduce dependency parser. Decoding is done with a beam search where pruning occurs after each shift action. The RNN model has an output-dependent structure that matches locality in the parse structure, making it an "incremental" neural network (INN, previously called SSN). This INN computes hidden vectors that encode the preceding partial parse, and estimates the probabilities of the parser actions given this history. Unlike the previous generative INN parser, DINN is a discriminative parser, using lookahead instead of word prediction. In order to combine beam search with a discriminative model, word predictions are replaced by a binary correctness probability which is trained discriminatively.

4.1 Transition-Based Neural Network Parsing

In DINN, the neural network is used to estimate the conditional probabilities of a transition-based statistical parsing model.

4.1.1 The Probabilistic Parsing Model

In shift-reduce dependency parsing, a parser configuration consists of a stack P of words, the queue Q of words and the partial labelled dependency trees constructed by the previous history of parser actions. The parser starts with an empty stack P and all the input words in the queue Q. It stops when it reaches a configuration with an empty queue Q, with any words left on the stack then being attached to ROOT. We use an arc-eager algorithm, which has 4 actions that all manipulate the word s on top of the stack P and the word q on the front of the queue Q: *Left-Arc*$_r$ adds a dependency arc from q to s labelled r, then popping s from the stack. *Right-Arc*$_r$ adds an arc from s to q labelled r. *Reduce* pops s from the stack. *Shift* shifts q from the queue to the stack. For exact details, see Titov and Henderson (2007b).

To model parse trees, we model the sequences of parser actions which generate them. We take a history based approach to model these sequences of parser actions. So, at each step of the parse sequence, the parser chooses between the set of possible next actions using an estimate of its conditional probability, where T is the parse tree, $D^1 \cdots D^m$ is its equivalent sequence of shift-reduce parser actions and S is the input sentence:

$$\begin{aligned} P(T|S) &= P(D^1 \cdots D^m|S) \\ &= \prod_t P(D^t|D^1 \cdots D^{t-1}, S) \end{aligned}$$

Unlike in previous dependency parser evaluations, the evaluation script for this shared task requires that exactly one word be attached to the ROOT node of the sentence. We implemented this constraint by modifying the calculation of the set of possible next actions. If an action will lead to a parser configuration where all possible ways of finishing the parse result in more than one word being attached to ROOT, then that action is not a possible action.

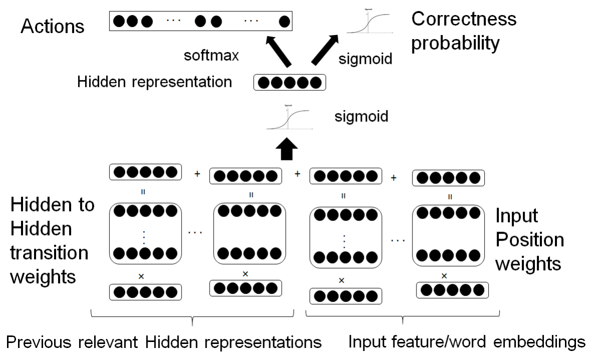
4.1.2 Estimating Action Probabilities

To estimate each $P(D^t|D^1 \cdots D^{t-1}, S)$, we need to condition on the unbounded sequences $D^1 \cdots D^{t-1}$ and S. To condition on the words in the queue, we use a bounded lookahead:

$$P(T|S) \approx \prod_t P(D^t|D^1 \cdots D^{t-1}, w^t_{a_1} \cdots w^t_{a_k})$$

where $w^t_{a_1} \cdots w^t_{a_k}$ is the first k words on the front of the queue at time t. At every *Shift*, one word is moved from the lookahead onto the stack and a new word from the input is added to the lookahead.

To estimate the probability of a decision at time t conditioned on the history of actions $D^1 \cdots D^{t-1}$,



Actions
Correctness probability
softmax
sigmoid
Hidden representation
sigmoid
Hidden to Hidden transition weights
Input Position weights
Previous relevant Hidden representations
Input feature/word embeddings

Figure 1: DINN computations for one decision

we use a recurrent neural network to induce hidden representations of the parse history sequence. The relevant information about the parse history at time t is encoded in the hidden representation vector h^t, of size d.

$$\prod_t P(D^t|D^1 \cdots D^{t-1}, w^t_{a_1} \cdots w^t_{a_k}) = \prod_t P(D^t|h^t)$$

This model is depicted in Figure 1. The hidden representation at time t is computed from selected previous hidden representations, plus pre-defined features. The model defines a set of link types $c \in \mathcal{C}$ which select previous states $t_c < t$ and connect them to the current hidden layer h^t via the hidden-hidden weights W^c_{HH}. The model also defines a set of features $f \in \mathcal{F}$ calculated from the previous decision and the current queue and stack, which are connected to the current hidden layer via the input-hidden weights W_{IH}:

$$h^t = \sigma(\sum_{c \in \mathcal{C}} h^{t_c} W^c_{HH} + \sum_{f \in \mathcal{F}} W_{IH}(f, :))$$

where σ is the sigmoid function and $W(i, :)$ is row i of matrix W.

The probability of each decision is estimated with a softmax layer (a normalised exponential) with outputs for all decisions that are possible at this step, conditioned on the hidden representation.

$$P(D^t{=}d|h^t) = \frac{e^{h^t W_{HO}(:,d)}}{\sum_{d'} e^{h^t W_{HO}(:,d')}}$$

where W_{HO} is the weight matrix from hidden representations to the outputs.

4.1.3 Hidden and Input Features

$\mathcal{C}$ and $\mathcal{F}$ are the only hand-coded parts of the model. Because $\mathcal{C}$ defines the recurrent connections in the neural network, it is responsible for passing information about the unbounded parse history to the current decision. Because RNNs are biased towards learning correlations which are close together in the connected sequence of hidden layers, we exploit this bias by making the structure of the neural network match the structure of the output parse. This is achieved by including previous states in $\mathcal{C}$ if they had a word on the top of the stack or front of the queue which are also relevant to the current decision. In the version we use in this experiment, we use a minimal set of these link types, specified in section 5.

The input features $\mathcal{F}$ are typical of any statistical model. But in the case of neural networks, it is common to decompose the parametrisation of these features into a matrix for the feature role (e.g. front-of-the-queue) and a vector for the feature value (e.g. a word). This decomposition of features overcomes feature sparsity, because the same value vector can be shared across multiple roles. Word embedding vectors are the most common example of this decomposition.

Unlike in the previous versions of the parser, Yazdani and Henderson (2015) added feature decompositions in the definition of the input-to-hidden weights W_{IH}.

$$W_{IH}(f, :) = W_{emb.}(val(f), :) W^{role(f)}_{HH}$$

Every row in $W_{emb.}$ is an embedding for a feature value, which may be a word, lemma, POS tag, or dependency relation. $val(f)$ is the index of the value for feature role $role(f)$, for example the particular word that is at the front of the queue. The matrix $W^{role(f)}_{HH}$ is the feature role matrix, which maps the feature value embedding to the role-value feature vector for the given feature role $role(f)$. For simplicity, we assume here that the size of the embeddings and the size of the hidden representations of the DINN are the same. In this way, the parameters of the embedding matrix $W_{emb.}$ is shared among various feature input link types $role(f)$, which can improve the model in the case of sparse features f.

We train our own word embeddings within the parsing model, using only the parsed training data. We tried initialising with Facebook embeddings on a sample of languages, but random initialisation worked better.

Unlike Yazdani and Henderson (2015), we did not cache any features, either in testing or in training. Caching can have a big impact on speed, but it has not been shown to improve accuracies.

4.1.4 Discrimination of Correct Partial Parses

Unlike in the previous generative models, the above formulas for computing the probability of a parse make independence assumptions, in that words to the right of $w^t_{a_k}$ are assumed to be independent of D^t. And even for words in the lookahead, it can be difficult to learn correlations with the unstructured lookahead string. If a discriminative model uses normalised estimates for decisions, then once a wrong decision is made there is no way for the estimates to express that this decision has lead to a structure that is incompatible with the current or future lookahead string (see Lafferty et al. (2001) for more discussion). For this reason, there is no obvious way to make effective use of beam search for a normalised discriminative model.

To overcome this problem, DINN estimates a correctness probability after every *Shift* action. This output is trained to discriminate correct from incorrect parse prefixes, using the same hidden representation as used to predict parser actions, as depicted in Figure 1. A beam search is then used to consider multiple possible partial parses so that the correctness probability can be used to select between them. The total score of a parse is the multiplication of the probabilities of all its actions with the correctness probabilities at the shift of each word. For more details on this technique, see Yazdani and Henderson (2015).

5 Experimental Settings

The implementation of DINN uses a parameter file to define the hidden-hidden connections, the input-hidden features, training meta-parameters, and various other parameters of the parser. We use the same settings for all languages. For the official submission, we used the following settings.

- We used a frequency cutoff for words/lemmas of 3.
- We did not normalise the input string to lowercase.
- We always used all the available training and development set.
- Search beam size is 10.
- Hidden layer size is 80.
- The size of the internally calculated embeddings is 50.
- Word embeddings are initialised randomly.
- We do not apply any feature caching.
- Validation occurs at every iteration.
- The configurations of the Input-to-Hidden layer connections are as follows:
    + Look at 4 last elements in the stack and 4 next elements from the input (Except the treebanks fr, ko, it_partut, grc_proiel, cu, where we look at 5 last elements from the stack.)
    + For each element, use all possible features from UDPipe (except UPoS if XPoS exists).
- The configurations of the Hidden-to-Hidden layer connections are as follows:

Closest	Current	H-to-H
Queue	Queue	+
Top	Top	+
Queue	Top	+

In this specification of the hidden-to-hidden connections, Queue refers to the front of the input queue and Top refers to the top of the stack in the parser configuration. This specification uses the same simplified set of connections between hidden states used in Yazdani and Henderson (2015). We assume that the induced hidden features primarily relate to the word on the top of the syntactic stack and the word at the front of the queue, since these are the words used in any action. To decide which previous state's hidden features are most relevant to the current decision, we look at these words in the current parser configuration. For each such word, we look for previous states where the top of the stack or the front of the queue was the same word. If more than one previous state matches, then the hidden vector of the most recent one is used. If no state matches, then no connection is made.

5.1 Training

One aspect of the current implementation which is basically unchanged from ten years ago is the training protocol. Learning rates and weight decay regularisation rates are reduced during training whenever there is a decrease in accuracy on the development set, and early stopping is used to prevent overtraining. Training and development splits are those provided by the shared task. The development set is also used to select which iteration's

Language	abbr.	dev. LAS	test LAS	Rank /33
Ancient Greek	grc	54.98	54.56	15
Ancient Greek-PROIEL	grc_proiel	63.30	62.83	18
Arabic	ar	63.37	64.17	23
Basque	eu	62.73	62.47	26
Bulgarian	bg	82.46	83.50	18
Catalan	ca	82.88	82.83	24
Chinese	zh	52.51	54.89	25
Croatian	hr	72.80	73.78	27
Czech	cs	83.36	82.52	18
Czech-CAC	cs_cac	82.07	81.35	23
Czech-CLTT	cs_cltt	63.98	69.16	22
Danish	da	69.94	69.43	27
Dutch	nl	73.75	67.70	22
Dutch-LassySmall	nl_lassysmall	68.84	73.97	22
English	en	75.69	75.09	22
English-LinES	en_lines	73.03	72.68	21
English-ParTUT	en_partut	71.97	71.78	25
Estonian	et	53.32	52.67	26
Finnish	fi	64.38	63.93	27
Finnish-FTB	fi_ftb	75.69	76.26	8
French	fr	83.74	79.85	21
French-ParTUT	fr_partut		17.85	30
French-Sequoia	fr_sequoia	76.35	76.36	25
Galician	gl	76.04	75.93	23
Galician-TreeGal	gl_treegal		2.76	30
German	de	72.76	69.59	16
Gothic	got	57.47	57.72	21
Greek	el	76.93	77.80	23
Hebrew	he	58.93	55.36	24
Hindi	hi	87.20	86.80	17
Hungarian	hu	53.81	50.95	27
Indonesian	id	68.33	69.45	26
Irish	ga		4.30	30
Italian	it	84.01	85.05	19
Japanese	ja	73.22	71.85	23
Kazakh	kk		1.00	29
Korean	ko	57.39	61.08	18
Latin	la		5.72	31
Latin-ITTB	la_ittb	69.00	75.81	18
Latin-PROIEL	la_proiel	55.28	54.07	22
Latvian	lv	60.06	59.28	22
Norwegian-Bokmaal	no_bokmaal	82.37	82.44	20
Norwegian-Nynorsk	no_nynorsk	80.73	79.34	21
Old Church Slavonic	cu	62.80	62.45	20
Persian	fa	75.74	75.86	23
Polish	pl	80.32	79.83	14
Portuguese	pt	81.83	79.74	22
Portuguese-BR	pt_br	84.75	84.00	21
Romanian	ro	77.27	77.34	23
Russian	ru	73.03	72.03	22
Russian-SynTagRus	ru_syntagrus	83.78	83.89	23
Slovak	sk	73.70	73.30	18
Slovenian	sl	80.31	81.32	14
Slovenian-SST	sl_sst		4.37	30
Spanish	es	81.92	79.96	22
Spanish-AnCora	es_ancora	81.63	81.26	23
Swedish	sv	71.86	76.06	20
Swedish-LinES	sv_lines	73.07	73.82	18
Turkish	tr	48.42	47.91	18
Ukrainian	uk		7.87	30
Urdu	ur	75.37	76.01	20
Uyghur	ug		9.29	27
Vietnamese	vi	39.14	35.77	25
MEAN		71.17	62.83	

Table 1: DINN and Universal Dependencies treebanks - official results.

model to use in testing. Recent advances in optimisation methods for neural networks — such as AdaGrad and mini-batch – are obvious modifications to compare against the reported results.

To deal with small treebanks without development sets, we use a fixed training protocol developed by looking at the training of models with other small training sets. We run for a total of 8 iterations and changed the learning rate and weight decay values every other iteration.

5.2 Dealing with surprise languages and other small datasets

To build a model for the surprise languages, we use simple cross-lingual techniques. For the official test phase, we identified the most similar languages to the surprise language with a string-based technique, concatenated the treebanks, trained and tested on the surprise languages.

The string-based technique constructs a list of words for each language. We used the sample data for the surprise language and the training data for the languages for which we have enough resources. Call these languages with big data sets $\mathcal{B}$. We denote $\mathcal{T}$ as the set of lists of words of $\mathcal{B}$, and t is a word in $\mathcal{T}$. For a given surprise language, we calculate the similarity score S for each t. We treat two words as similar if and only if the first three characters of these two words are identical and the edit distance between these two words is less than or equal to 1. We choose the language that has the best S for training our model for the surprise language. This procedure yields the following similar languages for training. We call them source languages.

Buryat:	Russian (rus_syntagrus), Turkish (tr)
Upper Sorbian:	Czeck (cs), Norwegian (no_bokmaal)
Kurmanji:	Spanish (es), Turkish (tr)
North Sami:	Czech (cs), Finnish (fi_ftb)

To train a parser for the surprise language, we concatenate the datasets for the source languages with three copies of the dataset for the target language. Because our frequency threshold is three, this means that all words in the target language dataset are included in the vocabulary. Then we trained a parser on this concatenated dataset, using the surprise language corpus also as a development set.

In addition to the surprise languages, there are other languages whose available data is just enough for a small training set without any development set. For the submitted test run, we did not do anything special for these datasets (other than the training schedule discussed above), training parsing models on the individual datasets. But in subsequent experiments we tried treating them in the same way as surprise languages, with much improved results, discussed below.

Language	Abbrev.	LAS	rank
Buryat	bxr	22.59	16
Kurmanji	kmr	22.20	22
North Sami	sme	23.99	21
Upper Sorbian	hsb	48.50	18
Mean on surprise languages		29.32	

Table 2: DINN and Universal Dependencies treebanks - official results on surprise languages.

Language	Abbrev.	LAS	rank
Arabic-PUD	ar_pud	42.61	25
Czech-PUD	cs_pud	79.17	18
English-PUD	en_pud	78.22	18
Finnish-PUD	fi_pud	64.91	24
French-PUD	fr_pud	74.93	9
German-PUD	de_pud	67.76	15
Hindi-PUD	hi_pud	51.31	10
Italian-PUD	it_pud	83.28	21
Japanese-PUD	ja_pud	76.21	16
Portuguese-PUD	pt_pud	73.01	17
Russian-PUD	ru_pud	67.22	16
Spanish-PUD	es_pud	75.90	22
Swedish-PUD	sv_pud	68.92	21
Turkish-PUD	tr_pud	29.01	25
Mean on PUD treebanks		66.60	

Table 3: DINN and Universal Dependencies treebanks - official results on PUD Treebanks.

6 Test Phase Results

Evaluation was run on the provided TIRA platform (Potthast et al., 2014) using the data provided by the organisers (Nivre et al., 2017b), but blind to us, as described in the introduction. The results of our submission are shown in the next three tables. Accuracy by LAS is shown in Table 1. Accuracy on surprise languages is shown in Table 2. Accuracy on parallel UD data is shown in Table 3.

6.1 Analysis of results

Our results are 25th over the 33 participants globally, 22nd on the large treebanks only, 19th on the PUD treebanks only, 30th on the small treebanks with only 6% accuracy (see below), 20th on sur-

		Training runtimes			Testing runtimes	
Language	abbr.	T/s	NUI	UTH	DPT	W/s
Ancient Greek	grc	0.0714	14	3.18	177	125.056
Ancient Greek-PROIEL	grc_proiel	0.0663	25	6.84	99	137.899
Arabic	ar	0.2698	144	65.56	259	116.753
Basque	eu	0.1001	66	9.90	175	137.686
Bulgarian	bg	0.1360	57	19.17	84	191.536
Catalan	ca	0.1998	23	16.75	404	139.807
Chinese	zh	0.3020	13	4.36	71	178.352
Croatian	hr	0.0990	56	11.84	93	156.269
Czech	cs	0.1256	26	62.14	1130	140.959
Czech-CAC	cs_cac	0.1615	43	45.29	87	125.425
Czech-CLTT	cs_cltt	0.2688	10	0.35	110	94.373
Danish	da	0.0835	58	5.90	69	149.739
Dutch	nl	0.3163	16	17.33	80	143.225
Dutch-LassySmall	nl_lassysmall	0.2697	14	6.33	75	140.667
English	en	0.0829	17	4.91	163	154.282
English-LinES	en_lines	0.0931	12	0.85	127	134.661
English-ParTUT	en_partut	0.1037	105	3.30	79	154.975
Estonian	et	0.1613	129	13.08	73	165.795
Finnish	fi	0.0842	137	39.16	118	155.000
Finnish-FTB	fi_ftb	0.1797	20	14.96	86	182.837
French	fr	0.1297	19	9.95	289	123.758
French-ParTUT	fr_partut	0.1226	8	0.17		
French-Sequoia	fr_sequoia	0.0829	429	22.05	63	158.937
Galician	gl	0.1402	15	1.33	177	168.232
Galician-TreeGal	gl_treegal	0.1183	8	0.16		
German	de	0.0853	124	41.47	80	154.350
Gothic	got	0.0484	32	1.46	60	168.567
Greek	el	0.1107	14	0.72	63	160.937
Hebrew	he	0.1074	132	20.64	68	167.765
Hindi	hi	0.0929	9	3.09	213	165.338
Hungarian	hu	0.1308	168	5.55	69	165.478
Indonesian	id	0.1072	127	16.93		
Irish	ga	0.1166	8	0.15		
Italian	it	0.1049	16	5.99	79	150.734
Italian-ParTUT	it_partut	0.1165	22	0.78	83	166.349
Japanese	ja	0.1336	242	64.33		
Kazakh	kk	0.0968	8	0.01		
Korean	ko	0.1734	35	7.42	58	191.345
Latin	la	0.2819	8	0.84		
Latin-ITTB	la_ittb	0.1741	146	111.61	80	129.138
Latin-PROIEL	la_proiel	0.1206	30	14.27	86	143.756
Latvian	lv	0.2741	18	3.17	60	168.800
Norwegian-Bokmaal	no_bokmaal	0.1099	174	83.38	222	163.824
Norwegian-Nynorsk	no_nynorsk	0.1375	83	44.94	190	164.474
Old Church Slavonic	cu	0.0403	17	0.78	54	187.037
Persian	fa	0.4894	18	11.74	112	141.357
Polish	pl	0.1520	46	11.84	53	193.623
Portuguese	pt	0.6852	10	15.86	83	130.735
Portuguese-BR	pt_br	0.5102	13	17.81	191	168.215
Romanian	ro	0.6079	15	20.37	126	135.508
Russian	ru	0.2192	14	3.28	81	146.630
Russian-SynTagRus	ru_syntagrus	0.1073	19	27.66	804	147.297
Slovak	sk	0.0888	16	3.35	72	172.778
Slovenian	sl	0.1882	13	4.40	86	163.523
Slovenian-SST	sl_sst	0.0917	8	0.44		
Spanish	es	0.2132	15	12.60	271	137.089
Spanish-AnCora	es_ancora	0.2073	99	81.54	367	142.605
Swedish	sv	0.1406	77	12.94	63	155.508
Swedish-LinES	sv_lines	0.2144	15	2.45	120	137.183
Turkish	tr	0.1115	148	16.90	79	126.722
Ukrainian	uk	0.1356	8	0.26		
Urdu	ur	0.3030	7	2.38	101	144.366
Uyghur	ug	0.1000	8	0.02		
Vietnamese	vi	0.1536	317	18.93	67	171.851
MAX			429	111.61	1130.000	193.623
MEAN		0.1727		16.83	152.389	152.576

Table 4: Training and testing runtimes. T/s: Training time per sentence; NUI: Number of useful iterations; UTH: Useful training time (hours); DPT: Development parsing time (seconds); W/s: Words/sec.

Model	Training				Testing		
	Training set	Dev. set	LAS	LAS deproj	LAS	LAS deproj	DINN Official LAS
FrenchParTUT	fr+ fr_partut	fr	85.08	85.09	78.14	78.14	17.85
GalicianTreeGal	gl+gl_treegal	gl	75.88	75.88	60.94	60.94	2.76
Irish	ga+it_partut	it_partut	75.99	75.99	59.44	59.44	4.30
Kazakh	kk+ja	ja	93.10	93.10	18.03	18.03	1.00
Latin	la+grc	grc	55.64	56.86	43.22	43.59	5.72
SlovenianSST	sl+sl_sst	sl	81.09	81.52	47.91	47.99	4.37
Ukrainian	lv+uk	lv	62.84	62.94	59.64	59.64	7.87
Uyghur	ug+bg	bg	83.40	83.49	29.66	29.66	9.29
Mean on small treebanks			76.63	76.86	49.62	49.68	6.64

Table 5: Small treebanks as surprise language, Run 5.1 UD PMor (20/05).

prise languages. They are rather firmly in the bottom third, around 22nd-25th place. They rarely beat the baseline. They are well above the baseline or close to it (above or below) for twelve treebanks (Fi_ftp, 8th, well above; fr_pud, 9th/33, well above; grc, 15th, just under; hi, 17th, just above; hi_pud, 10th, well above; it, 19th; ja_pud, 16th just below; ko, 18th, well above; la_ittb, 18th, same; pl, 14th, above; sk, 18th, above; sl, 14th a little above.)

There are a number of treebanks where the submitted parser does very poorly (fr_partut, 17%; ga, 4%; gl_treegal, 2.75%; kk, 1%; la, 6%; sl_sst, 4%; ug, 9%; uk, 8%). These are all small treebanks with no development set, which we treated in the same way as all other treebanks. As discussed in the post-test results section, treating these treebanks with the same approach that we used for surprise languages yielded instead results on-average 43% LAS better.

6.2 Resources used

Table 4 shows the training and parsing times, calculated on the training and development sets, respectively. Our shared task submission was prepared primarily by one computer science MSc student.

7 Post-Test Results

In the post-test results, we aim to increase the performance on the small treebanks, and correct errors in the submitted system.

7.1 Postprocessing, if any

In the submitted parser, we overlooked the need to deprojectivise the output of the parser. In the post-test results, we run the Malt parser deprojectivisation routine on the output of the DINN parser before doing evaluation. Deprojectivisation makes no or little difference for most languages, but there is an improvement on some. Improvements range from zero to 1.3% LAS score, with an average improvement of 0.16%. We report some deprojectivisation results on small treebanks in Table 5.

7.2 Dealing with small treebank languages

In the test phase, we train on small treebanks. Given that our results were particularly unsatisfactory on small treebanks, in the post-test phase, we tried a different technique: we treated small treebanks like surprise languages.

For small treebanks, we identified the best source language by exhaustively searching all the possible languages. As with surprise languages, we then concatenated three copies of the small treebank to the larger treebank and trained a parser on this combined dataset. Table 5 shows the treebank configurations and results on the development set and test set. This new method raises the total average score of our parser by 4.20% LAS.

8 Conclusions and Future Work

With this submission, we have shown how a neural network dependency parser whose main architecture is largely unchanged from ten years ago performs with respect to the state of the art. These results can serve as a baseline for future work evaluating to what extent recently proposed methods have a measurable impact on neural network dependency parser accuracy.

Acknowledgements

We would like to thank Corentin Ribeyre for his help in getting the endeavour started and the Language and Communication Network at the university of Geneva for support. We also thank Majid Yazdani for his help understanding the original code base.

References

James Henderson. 2003. Inducing history representations for broad coverage statistical parsing. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*. Association for Computational Linguistics, Stroudsburg, PA, USA, NAACL '03, pages 24–31.

John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, ICML '01, pages 282–289. http://dl.acm.org/citation.cfm?id=645530.655813.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2006. Maltparser: A data-driven parser-generator for dependency parsing. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC-2006)*. pages 2216–2219.

Joakim Nivre et al. 2017a. Universal Dependencies 2.0. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University, Prague, `http://hdl.handle.net/11234/1-1983`. http://hdl.handle.net/11234/1-1983.

Joakim Nivre et al. 2017b. Universal Dependencies 2.0 CoNLL 2017 shared task development and test data. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University, Prague, `http://hdl.handle.net/11234/1-2184`. http://hdl.handle.net/11234/1-2184.

Martin Potthast, Tim Gollub, Francisco Rangel, Paolo Rosso, Efstathios Stamatatos, and Benno Stein. 2014. Improving the reproducibility of PAN's shared tasks: Plagiarism detection, author identification, and author profiling. In Evangelos Kanoulas, Mihai Lupu, Paul Clough, Mark Sanderson, Mark Hall, Allan Hanbury, and Elaine Toms, editors, *Information Access Evaluation meets Multilinguality, Multimodality, and Visualization. 5th International Conference of the CLEF Initiative (CLEF 14)*. Springer, Berlin Heidelberg New York, pages 268–299. https://doi.org/10.1007/978-3-319-11382-1_22.

Milan Straka, Jan Hajič, and Jana Straková. 2016. UDPipe: trainable pipeline for processing CoNLL-U files performing tokenization, morphological analysis, POS tagging and parsing. In *Proceedings of the 10th International Conference on Language Resources and Evaluation (LREC 2016)*. Portorož, Slovenia, publisher = European Language Resources Association, isbn = 978-2-9517408-9-1.

Ivan Titov and James Henderson. 2007a. Fast and robust multilingual dependency parsing with a generative latent variable model. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*. Association for Computational Linguistics, Prague, Czech Republic, pages 947–951. http://www.aclweb.org/anthology/D/D07/D07-1099.

Ivan Titov and James Henderson. 2007b. A latent variable model for generative dependency parsing. In *Proceedings of the 10th International Conference on Parsing Technologies*. Association for Computational Linguistics, Stroudsburg, PA, USA, IWPT '07, pages 144–155.

Majid Yazdani and James Henderson. 2015. Incremental recurrent neural network dependency parser with search-based discriminative training. In *Proceedings of the Nineteenth Conference on Computational Natural Language Learning*. Association for Computational Linguistics, Beijing, China, pages 142–152. http://www.aclweb.org/anthology/K15-1015.

Daniel Zeman, Filip Ginter, Jan Hajič, Joakim Nivre, Martin Popel, Milan Straka, and et al. 2017. CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*. Association for Computational Linguistics, pages 1–20.