
UNIVERSITY OF CAPE TOWN


DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Project Final Paper 2022

Title: Sequence-to-Sequence Meaning Representation Parsing with BART

Author: Chase Ting Chong

Project Abbreviation: MRP

Supervisor(s): Jan Buys

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	0
Theoretical Analysis	*0*	*25*	5
Experiment Design and Execution	*0*	*20*	20
System Development and Implementation	*0*	*20*	0
Results, Findings and Conclusions	*10*	*20*	20
Aim Formulation and Background Work	*10*	*15*	15
Quality of Paper Writing and Presentation	*10*		10
Quality of Deliverables	*10*		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	*0*	*10*	0
Total marks		**80**	

Sequence-to-Sequence Meaning Representation Parsing with BART

Chase Ting Chong
University of Cape Town
Cape Town, South Africa
tngcha001@myuct.ac.za

ABSTRACT

Meaning Representation Parsing (MRP) is the parsing of a sentence into a semantic graph. In this paper we implement a sequence-to-sequence Elementary Dependency Structures (EDS) parser using the BART pretrained encoder-decoder transformer model. This parser converts English sentences into EDS semantic graphs. The model is trained on gold graphs that are serialized using a modified PENMAN notation. The parser improves upon a previous RNN-based model achieving an EDM score of 91.12 which is 2.44 points lower than the state-of-the-art Hitachi System.

KEYWORDS

Natural Language Processing, Natural Language Understanding, Meaning Representation Parsing, Semantic Graph Parsing, Neural Networks

1 INTRODUCTION

Meaning representation parsing [23] is the parsing of a sentence into a *semantic graph*. A semantic graph represents the meaning of a sentence and consists of nodes that correspond to words in the sentence, and labelled edges that define the semantic interactions between nodes. There are many semantic graph frameworks each with their own properties. In this paper we will focus on the parsing of English sentences into Elementary Dependency Structures (EDS) [25], one of the semantic graph frameworks.

There are two well-established methods of meaning representation parsing in the literature namely Transition-based and Graph-based parsing [23]; however, with the emergence of the Transformer neural network architecture [32], there has been a shift to using pretrained transformer models with a sequence-to-sequence (§3.2) based approach [2, 22, 26]. A pretrained model is a model trained on a large unlabelled corpus with some training objective. The goal of pretraining is to create a model that has a general understanding of a language that can then be used in a more specific downstream task. To use a pretrained model it is further trained on data related to a specific task, this process is called *fine-tuning*.

The aim of this paper is to extend the work of [3], a Recurrent Neural Network (RNN) based sequence-to-sequence parser that makes use of graph linearization techniques to encode graphs into sequences that can be later converted back to graphs. We replace the RNN with the pretrained encoder-decoder transformer model BART [17]. Additionally we aim to answer the following research questions:

- How does the F1 score of the BART model compare to the F1 score of previous RNN based encoder-decoder models?
- How does the F1 score of a pretrained model compare to that of a non-pretrained model?

A similar approach has been taken by [2] using BART to parse English sentences to Abstract Meaning Representation (AMR) [1] graphs. This work achieved state-of-the-art results at its time of publication we therefore hypothesize that applying this approach to EDS parsing will outperform the previous RNN based model.

2 BACKGROUND

2.1 Meaning Representation

In this section we describe two meaning representation frameworks, EDS which is the main representation used in this paper and AMR which is used in all related works, shares similarities with PENMAN and is used to compute the Smatch metric.

2.1.1 *Elementary Dependency Structures.*

EDS [25] is a type of semantic graph derived from Minimal Recursion Semantics (MRS) [6]. MRS is the semantic representation used by the English Resource Grammar (ERG) [9], a general-purpose computational grammar that converts English Text to highly normalized logical-form meaning representations.

An EDS graph is a rooted, labelled, connected, directed graph. Node labels are called *predicates* and edge labels, *arguments*. Each node in the graph corresponds with a token or a continuous set of tokens in the sentence. This correspondence is called the *alignment* or *span* of the node [3]. Figure 1 shows the EDS graph for the sentence "The results were in line with analysts' expectations.", the graph was visualized using the RepGraph tool[1].

2.1.2 *Abstract Meaning Representation.*

Abstract Meaning Representations (AMRs) [1] are rooted, directed, edge-labeled, leaf-labeled graphs. Nodes are represented by *variables* that correspond to entities, events, properties, and states. Leaf nodes are labelled by concepts which can be either English words, PropBank [16] framesets, or special AMR keywords. A PropBank frameset can be thought of as a unique verb label that corresponds to a precise definition of how other words interact with the verb. A *relation* or labelled edge defines the relationship between entities (nodes), AMR uses approximately 100 relations. For example in Figure 2 the root node is an instance of the want concept labelled by the PropBank frameset `want-01`.

The main difference between EDS and AMR is that AMR does not have an alignment between nodes in the graph and the tokens of the surface sentence.

[1] https://repgraph.vercel.app/

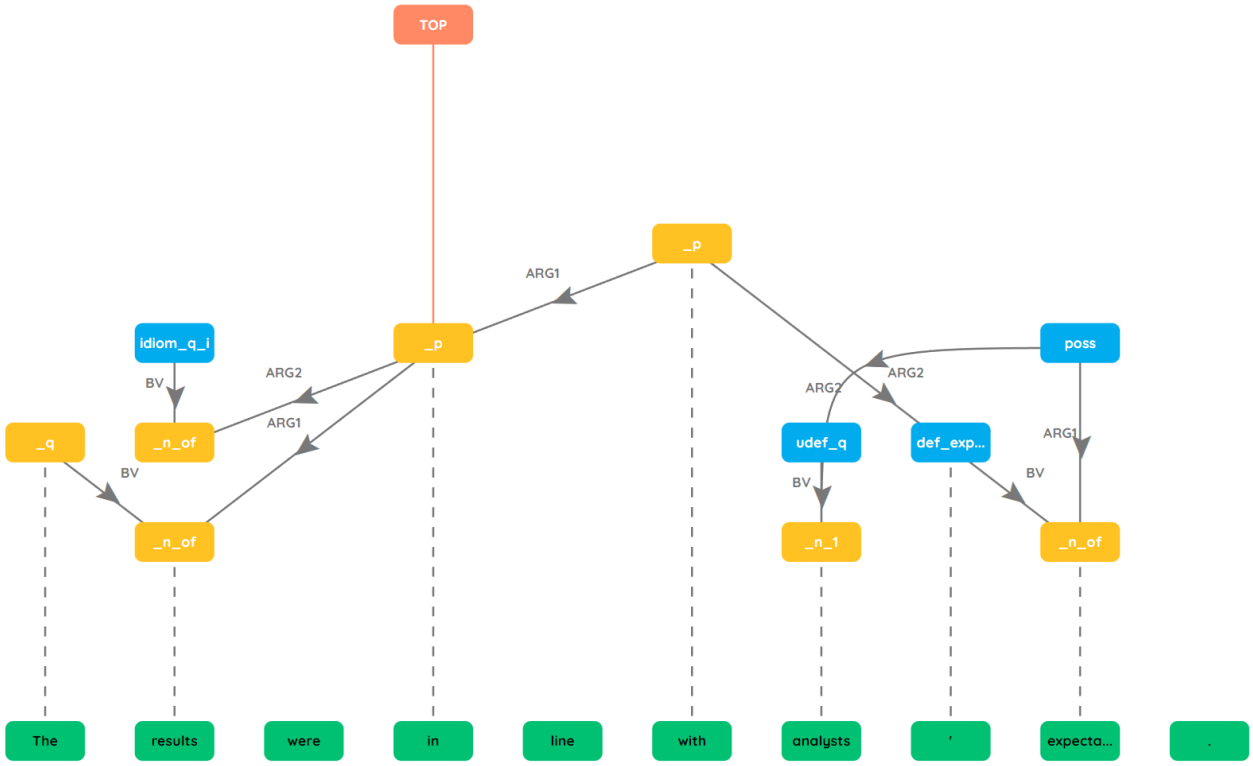

Figure 1: EDS Graph for the sentence "The results were in line with analysts' expectations."

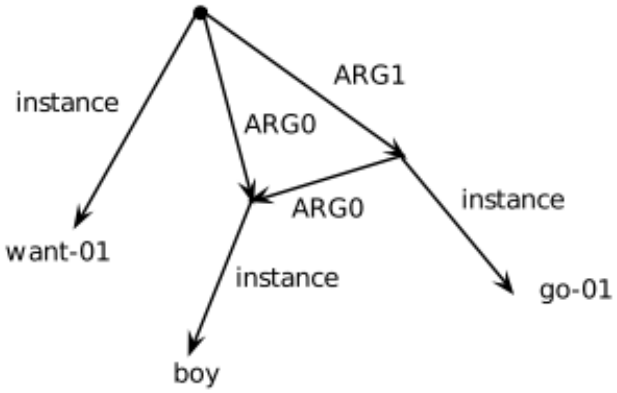

Figure 2: AMR Graph for the sentence "The boy wants to go" [1]

2.2 Sequence-to-Sequence Neural Networks

Sequence-to-Sequence models are end-to-end models that when given an input sequence predict an output sequence. Sequence-to-Sequence models are implemented using an encoder-decoder neural network architecture. Two commonly used neural network architectures for sequence modelling will be discussed in this section.

2.2.1 *Recurrent Neural Networks.*

A Recurrent Neural Network (RNN) processes sequences in time steps, at each time step taking a new token from the input sequence. RNNs contain cycles within its network connections allowing RNNs to receive hidden state values from previous time steps. This information acts as a form of context or memory that is taken into account when calculating the new hidden value at the current time step [14]. This simple RNN model suffers from two problems. Firstly, when processing a sequence, the information encoded in the hidden states are more relevant to the most recently processed tokens therefore tokens processed long before the current time step do not have a strong influence on current decisions. However, the ability to remember distant information is an important feature to have in many language applications [14]. Secondly, RNNs suffer from the vanishing gradients problem [27], since the value of hidden states depends on the values of previous time steps, there are many repeated multiplications during backpropagation leading to the gradients being driven to zero [14].

Long Short Term Memory (LSTM) [13] is an extension to RNNs that mitigates the above problems. LSTMS add a memory cell that stores the context. This context is sent along with the hidden state at each time step. LSTMs make use of three gates to manage the context [14]:

- The **forget** gate deletes information from the context that is no longer needed.

- The **add** gate decides what information should be added to the current context.
- The **output** gate decides which information is relevant for the current hidden node.

2.2.2 *Transformers.*

A Transformer [32] is a type of neural network that maps sequences of input vectors to sequences of output vectors. What differentiates Transformers from other neural network models is that it relies only on *attention*. The idea behind attention is to compare a collection of items based on their relevance to the current context. Transformers make use of two kinds of attention, *self-attention* and *multihead attention*. Self-attention is simply the comparison of an item in a collection to other items in the same collection. However, there may be various ways to classify something as relevant to a context, this is solved by multihead attention. Multihead attention is a set of self-attention layers, where each layer is called a *head*. Each head of these self-attention layers can now attend to a different representation of relevance [14].

Since Transformers do not use recurrence they do not encounter the problems that RNNs face. They are superior in quality, easily parallelizable and require significantly less time to train [32].

3 RELATED WORK

3.1 Transition-based Parsing

Transition-based parsing [20] has its roots in dependency parsing, which is the parsing of a sentence into a dependency graph. A dependency graph treats each word in a sentence as a node and labelled arcs (edges) between them represent syntactic modifiers. For an arc from node i to node j, i is called the *head* and j the *dependant*.

Transition-based dependency parsers construct dependency graphs incrementally by predicting a sequence of actions that when executed by a *stack-based transition system* create the graph. Actions are predicted by a neural network based classifier that is trained by an *oracle*. Transition systems follow an algorithm which defines the transition set. We describe the *arc-eager* algorithm, one of many algorithms, below. Many algorithms can be extended to parse semantic graphs as well as general graphs by adding new transitions to the transition set [21, 30].

Transition-based parsers are attractive because of their efficient implementation and linear run time [21]. However, because of their local greedy nature they are prone to search errors in sentences that require long transition sequences [19].

3.1.1 *Stack-Based Transition System.*

A stack-based transition system consists of parser configurations, a transition set, a way to map an input sentence to a starting configuration and set of configurations that when transitioned to terminates the algorithm [20].

- A parser configuration consists of a stack that holds nodes currently being processed, a buffer holding tokens (words in the sentence) that still need to be processed and a set of arcs.
- A transition set is a set of actions that perform a transformation to a parser configuration resulting in a new configuration.

Transition	Configuration			
	([0],	[1, . . . , 9],	∅	)
SHIFT ⟹	([0, 1],	[2, . . . , 9],	∅	)
LEFT-ARC$_{\text{NMOD}}$ ⟹	([0],	[2, . . . , 9],	$A_1 = \{(2, \text{NMOD}, 1)\}$	)
SHIFT ⟹	([0, 2],	[3, . . . , 9],	A_1	)
LEFT-ARC$_{\text{SBJ}}$ ⟹	([0],	[3, . . . , 9],	$A_2 = A_1 \cup \{(3, \text{SBJ}, 2)\}$	)
RIGHT-ARC$^e_{\text{ROOT}}$ ⟹	([0, 3],	[4, . . . , 9],	$A_3 = A_2 \cup \{(0, \text{ROOT}, 3)\}$	)
SHIFT ⟹	([0, 3, 4],	[5, . . . , 9],	A_3	)
LEFT-ARC$_{\text{NMOD}}$ ⟹	([0, 3],	[5, . . . , 9],	$A_4 = A_3 \cup \{(5, \text{NMOD}, 4)\}$	)
RIGHT-ARC$^e_{\text{OBJ}}$ ⟹	([0, 3, 5],	[6, . . . , 9],	$A_5 = A_4 \cup \{(3, \text{OBJ}, 5)\}$	)
RIGHT-ARC$^e_{\text{NMOD}}$ ⟹	([0, . . . , 6],	[7, 8, 9],	$A_6 = A_5 \cup \{(5, \text{NMOD}, 6)\}$	)
SHIFT ⟹	([0, . . . , 7],	[8, 9],	A_6	)
LEFT-ARC$_{\text{NMOD}}$ ⟹	([0, . . . 6],	[8, 9],	$A_7 = A_6 \cup \{(8, \text{NMOD}, 7)\}$	)
RIGHT-ARC$^e_{\text{PMOD}}$ ⟹	([0, . . . , 8],	[9],	$A_8 = A_7 \cup \{(6, \text{PMOD}, 8)\}$	)
REDUCE ⟹	([0, . . . , 6],	[9],	A_8	)
REDUCE ⟹	([0, 3, 5],	[9],	A_8	)
REDUCE ⟹	([0, 3],	[9],	A_8	)
RIGHT-ARC$^e_{\text{P}}$ ⟹	([0, 3, 9],	[ ],	$A_9 = A_8 \cup \{(3, \text{P}, 9)\}$	)

Figure 3: Arc Eager transition sequence for the sentence "Economic news had little effect on financial markets". Configurations are represented by a triple (stack, buffer, arcs) [20]

- The initial configuration starts with an artificial root node 0 on the stack, nodes i, for every word w_i in the input sentence, on the buffer and the set of arcs is empty.
- A terminal configuration is any configuration where the buffer is empty.

3.1.2 *Arc Eager Algorithm.*

The arc eager algorithm's transition set consists of four types of transitions [20]:

- **Left-Arc**$_l$ adds a dependency arc with label l from the first node in the buffer j to node i at the top of the stack and then pops i off the stack, given that i is not the root node and does not have a head.
- **Right-Arc**$_l$ adds a dependency arc with label l from the node i at the top of the stack to the first node in the buffer j and then shifts j onto the stack, given that j does not already have a head.
- **Reduce** Pops the top node i from the stack, given that i has a head.
- **Shift** removes the first node i in buffer and pushes it to the top of the stack.

The arc eager algorithm tries to attach right dependents as soon as possible; therefore, the head and dependent must be stored on the stack until the dependent gets all its right dependents and can only then be popped of the stack using the reduce transition [20]. Figure 3 shows an example transition sequence.

3.1.3 *Oracles.*

A core part of training classifiers for transition-based dependency parsers is an oracle. An oracle is a function that when given a gold graph returns the optimal transition sequence to recreate the graph [12].

It is possible that multiple sequences lead to the same gold graph. There are two types of oracles that each deal with this situation in their own way:

- **Static Oracles** define a forced derivation order, typically as rules for a given parser configuration, and returns a single sequence. This has its limitations as the sequence given

by the oracle may not be the easiest to learn. In addition since the parsing is greedy it possible for the parser to deviate from the gold sequence to configurations where the gold graph is no longer reachable. The oracle does not have a mechanism to deal with these configurations, this leads to error-propagation as the parser's classifier is faced with configurations not observed in training [12].

- **Dynamic Oracles** do not restrict parsers to a single sequence but instead return all valid sequences leading to the gold graph. Another improvement of dynamic oracles is that, unlike their static counterpart, they are well-defined and correct for all configurations. For configurations that have deviated from the gold sequence, the dynamic oracle allows all transitions that lead to a graph with minimum loss compared to the gold graph thus mitigating error propagation [12].

3.2 Sequence-to-Sequence Parsing

Sequence-to-Sequence parsers take advantage of sequence-to-sequence neural network architectures to predict graphs in an end-to-end manner only requiring minor postprocessing to transform the sequence back into a graph. To achieve this graphs are converted to a sequence that the neural network can be trained to predict using some form of graph linearization technique. Sequence-to-Sequence parsers remove the need for complex pipelines typically found in other models and have been found to outperform these models in many Natural Language Understanding tasks [3, 34].

In this section we look at other sequence-to-sequence parsers and discuss their relevancy to this work.

3.2.1 RNN Sequence-to-Sequence with Graph Linearizations.
[3] uses a RNN encoder-decoder architecture to implement a sequence-to-sequence parser focused mainly on two graph-based conversions of MRS, EDS and Dependency MRS (DMRS) [5] but can also parse AMR. [3] compares two graph linearization approaches across each framework. Firstly, a top-down linearization that linearizes the graph as a pre-order traversal of its spanning tree starting from a designated root node and secondly, as a sequences of actions that can be used by a transition-based parser to reconstruct the graph. This work is the basis on which we build our approach.

3.2.2 The Hitachi System.
The Hitachi System [26] is the current state-of-the-art parser for EDS it was presented at the Conference for Computational Language Learning (CoNLL) 2020 Shared Task [22]. The Hitachi System is a multi-framework Text-to-Graph-Notation Transducer of which EDS is included. The Hitachi System makes use of an encoder-decoder Transformer architecture for sequence-to-sequence prediction. The output sequence is encoded as a Plain Graph Notation (PGN) a novel notation designed by the authors. The encoder uses a pretrained language model (PLM). Decoding is done with a Transformer decoder controlled by a mode switching mechanism.

The approach of the Hitachi System is similar to ours and we will use the results of the Hitachi System as a comparison to the state-of-the-art.

3.2.3 AMR Parsing with BART.
[2] is a symmetric sequence-to-sequence AMR parser using BART, converting AMR-to-Text as well as Text-to-AMR, we will focus on the Text-to-AMR portion of this parser. [2] compares three graph linearization approaches; A PENMAN, DFS-based and BFS-based linearization. The PENMAN linearization uses the standard PENMAN serialization described in §4.1. The DFS and BFS based linearizations are created based on PENMAN notation. They make use of special tokens to replace the AMR variables and remove the '/'. Node and edge labels are added in the order of the DFS and BFS traversal respectively where brackets are used to indicate the depth. [2] also applies various optimizations to the BART model some of which we also apply and are discussed later in this paper.

4 METHODS

4.1 Graph Linearization

To fit the sequence-to-sequence model we need to linearize the output graph to a sequence. In this section we discuss our chosen method for graph linearization as well as other optimizations.

PENMAN [15] is a graph serialization format used to encode the semantic dependencies for directed, rooted graphs. The structure of PENMAN notation can be defined by the Parsing Expression Grammar (PEG) [11] shown in Figure 4. A graph is represented in PENMAN by nodes and relations, where the nodes contain the node labels and relations contain a role (edge label) and a target node. Edges that are inverted in the serialization are marked by adding `-of` to the role. PENMAN notation assigns variable names to nodes however these variable names do not appear in the graph and are only used to reference nodes that are re-used in the serialization. For example a typical PENMAN node would like this:

```
(e3 / _take_v_1 :lnk "<91:96>")
```

Where `:lnk` is the character level span (the characters in the input sentence) that the node refers to. This representation contains unnecessary information that the model will have to learn to predict, we therefore use the following simplified encoding:

```
(< 10 10 > _v_1)
```

The lemma (root word) is removed from the node label and the character level span is converted to a token level span, this reduces data sparsity and makes it easier for the model to predict the spans due to the input being tokenized. The concatenation of these two properties forms a unique identifier for a node and can therefore be used as the variable name. The node label and span information are omitted as they are now encoded in the variable name. Figure 5 shows the simplified serialization for the sentence "The results were in line with analysts' expectations." the graph is shown in Figure 1.

We make use of the `penman`[2] python library to handle the encoding and decoding of the graphs.

4.2 BART

4.2.1 Model.
BART [17] is an encoder-decoder transformer model. BART uses a BERT-like [7] bidirectional encoder, which uses bidirectional self-attention and a Masked Language Model (MLM) to produce deep contextualized representations that combine left and right context.

[2] https://penman.readthedocs.io/en/latest/index.html

```
# Syntactic productions (whitespace is allowed around non-terminals)
Start     <- Node
Node      <- '(' Variable NodeLabel? Relation* ')'
NodeLabel <- '/' Concept Alignment?
Concept   <- Constant
Relation  <- Role Alignment? (Node / Atom Alignment?)
Atom      <- Variable / Constant
Constant  <- String / Symbol
Variable  <- Symbol

# Lexical productions (whitespace is not allowed)
Symbol    <- NameChar+
Role      <- ':' NameChar*
Alignment <- '~' ([a-zA-Z] '.'?)? Digit+ (',' Digit+)*
String    <- '"' (!'"' ('\\' . / .))* '"'
NameChar  <- ![ \n\t\r\f\v()/:~] .
Digit     <- [0-9]
```

Figure 4: Parsing Expression Grammar for PENMAN Notation[2]

For decoding BART uses a GPT-like [28] autoregressive decoder, meaning it uses previous predictions to generate output from left-to-right. BART makes use of various noising approaches for its pretraining scheme, such as token masking, token deletion, text infilling, sentence permutation and document rotation. Documents are corrupted via the above methods and then the reconstruction loss, the cross-entropy between the decoder output and the original document, is optimized to pretrain the model. BART performs well when fine tuned for generative tasks such as question answering and summarization [17].

We fine-tune the BART model to predict linearized graphs given an English sentence. We do this by further training the model on sentence-PENMAN pairs using the sentence as input and the PENMAN linearization as the target output.

4.2.2 *Tokenizer.*

The BART tokenizer uses the same byte-pair encoding (BPE) [31] as GPT-2 [29] [17]. BPE uses a subword vocabulary to encode out of vocabulary words [31], this vocabulary is optimized for English sentences therefore the tokenizer struggles with EDS labels. To combat this all role labels found in the training corpus as well as their inversions are added to the vocabulary as well all node labels that occur more than 100 times in the training corpus. A similar approach was also taken in [2].

4.3 Postprocessing

To make sure that graphs predicted by the model are valid, it is necessary to perform some postprocessing on the output of our models.

Firstly the brackets of the output sequence is balanced to restore the tree structure. Thereafter we attempt to decode with the `penman` library, if this fails then there are either unexpected or bad tokens in the output sequence, if this is the case a LL(1) parser based on the PEG in Figure 4 is used to correct the output.

At the tokenization stage of parsing two common problems emerge in the predicted output, firstly some node labels are separated by spaces when they should be a single label, and secondly some tokens are malformed. Both these problems are addressed using regular expression matching to rejoin labels or remove bad tokens.

At the parsing stage if an unexpected token is found then either; tokens are ignored until a valid token is found or in the cases where it is possible to correct, a new token is inserted, these cases include:

- Inserting a missing left bracket
- Inserting a generic node or role to complete an edge

The latter did not occur in the test set and therefore was not implemented, it is however simple to extend the parser to include this functionality.

If the sequence ended on an unexpected token; then, if the token is a role it can be replaced by a right bracket otherwise the end tokens are removed until the sequence ends on the correct token.

5 EXPERIMENTAL DESIGN

5.1 Data

Our data is sourced from the DeepBank [10] treebank, a MRS annotation of the Penn Treebank [18] Wall Street Journal (WSJ) corpus. DeepBank follows the dynamic treebanking approach [24] which couples treebank annotation with grammar development and is therefore coupled with the ERG. We use the data splits used in [3] for EDS which corresponds to ERG 1214[3], with sections 0 to 19 for training data, 20 for development and 21 for testing. To extract the EDS graphs we use a mrs-processing tool[4] which uses the pyDelphin library[5] to extract the EDS graph and then performs the following preprocessing steps:

- Removes lemmas from node labels
- Converts character level spans to token level spans

The EDS graphs are then converted into PENMAN following the method described in §4.1. These PENMAN linearizations are then paired with the input sentence to create sentence-PENMAN pairs which is used as input to train out models. Following this, encodings that are longer than 512 tokens are removed from the training pairs to minimize padding and allow for a reasonable batch size without running out of memory on the GPU.

The data is tokenized using a pretrained BART Tokenizer from the Huggingface transformers[33] library[6] with a modified vocabulary as described in §4.2.2.

5.2 Models

We train four models: a pretrained and non-pretrained `bart-base` and `bart-large`. The non-pretrained BART models, have the same architecture as the pre-trained BART models but do not use the weights found during pretraining. The training loop was implemented using the Huggingface `Trainer` API using the training data split for training and the development data split for validation. We use some of the hyperparameters suggested in [2] which are as follows: The model hyperparameters are the default model parameters (for bart-base or bart-large) defined in Huggingface's `transformers` library. The training hyperparameters are shown in Table 1. Our models are trained on the Lengau High Performance Cluster (HPC)[7] with a Nvidia V100 GPU and 10 CPU cores. Training information was sent to Weights and Biases for analysis,

[3] http://svn.delph-in.net/erg/tags/1214/
[4] https://gitlab.cs.uct.ac.za/jbuys/mrs-processing
[5] https://github.com/delph-in/pydelphin
[6] https://huggingface.co/docs/transformers/index
[7] https://www.chpc.ac.za/

```
(< 3 3 > _p :ARG1 (< 1 1 > _n_of :BV-of (< 0 0 > _q))
           :ARG2 (< 4 4 > _n_of :BV-of (< 4 4 > idiom_q_i))
           :ARG1-of (< 5 5 > _p :ARG2 (< 8 8 > _n_of :BV-of (< 7 7 > def_explicit_q)
                                                     :ARG1-of (< 7 7 > poss :ARG2 (< 6 6 > _n_1 :BV-of (< 6 6 > udef_q))))))
```

Figure 5: PENMAN Serialization for the sentence "The results were in line with analysts' expectations."

the evaluation and training loss can be seen in Appendix A, the large models are split into two runs as the runtime exceeded the maximum walltime allowed per job on the HPC.

Parameters	Value
Optimizer	AdamW
Epochs	30
LR	$5 * 10^{-5}$
Betas	0.9, 0.999
Dropout	0.25
W. Decay	0.004
Grad. accum.	10
Beam Size	5

Table 1: Training hyperparameters

5.3 Hyperparameter Search

The hyperparameters suggested in [2] were optimized for a pre-trained BART model, we therefore train an additional model for comparison after doing a hyperparameter search for the non-pretrained `bart-base` model.

As in [2] we perform a random search with a similar search space for 8 trials followed by a grid search for dropout rate in the range $[0, 0.25](+0.05)$. The hyperparamter search was performed using the Weights and Biases Sweeps API[8].

The final hyperparameters and search space are shown in Table 2. The evaluation loss and hyperparameter search results are shown in Appendix B.

Parameters	Value	Search Space
Optimizer	AdamW	-
Epochs	30	-
LR	$5 * 10^{-5}$	$1/5 * 10^{-5}$
Betas	0.9, 0.999	-
Dropout	0.05	0 to 0.25, (+0.05)
W. Decay	0.004	0.001 to 0.01, (+0.001)
Grad. accum.	1	1/5/10/15/20
Beam Size	5	-

Table 2: Final Hyperparameters for `bart-base` and search space

5.4 Evaluation

To evaluate our models we make use of Elementary Dependency Matching (EDM) [8] to calculate the precision, recall and F1 score and the Smatch [4] library[9] to calculate Smatch score. These are common metrics used to evaluate semantic graph structures [23].

[8]https://docs.wandb.ai/guides/sweeps
[9]https://github.com/snowblink14/smatch

5.4.1 *EDM.*

To calculate the precision, recall and F1 score via EDM, the gold graph and the predicted graph are converted into a series of tuples. A tuple can be either a node tuple or an edge tuple. Node tuples consist of the node span and the node label, whereas edge tuples consist of the span of the head node, the edge label and the span of the dependant node. Once the two graphs have been converted into tuples, the tuples are matched to get the true positives (TP), false positives (FP) and false negatives (FN) for the nodes and edges respectively. The precision, recall, and F1 score is then calculated as follows:

$$\text{TP} = \text{Tuples in the predicted graph that are in the gold graph}$$
$$\text{FP} = \text{Tuples in the predicted graph that are not in the gold graph}$$
$$\text{FN} = \text{Tuples in the gold graph that are not in the predicted graph}$$
$$\text{Precision} = \frac{\text{TP}}{(\text{TP} + \text{FP})}$$
$$\text{Recall} = \frac{\text{TP}}{(\text{TP} + \text{FN})}$$
$$\text{F1 Score} = \frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

The node and edge precision, recall and F1 score is averaged to get the overall precision, recall and F1 score.

5.4.2 *Smatch.*

Smatch is an evaluation metric for AMR graphs. The Smatch score is calculated by finding the a node mapping that maximizes the number of matching edge triples. The main difference between EDM and Smatch is that Smatch does not consider the alignment between the graph and its surface sentence.

To use the Smatch tool it is necessary to convert our EDS graphs into AMR graphs this is done similarly to §4.1 as PENMAN is an AMR representation, we now however include the node label again. Another thing to note is that PENMAN is a tree structure so a general graph can be represented by multiple trees, therefore the predicted graph and the gold graph may not have the same node and edge order when represented as triples (which is how penman represents the graph). To create consistent AMR graphs the triples of the predicted and gold graphs are sorted before encoding into PENMAN.

6 RESULTS AND DISCUSSION

6.1 Results

We report the results of evaluating our five models on the test data split in Table 3 where `pt-` indicates the model is pretrained and `bart-base-tuned` is the base model that was trained using the hyperparameters found from the hyperparameter search. The table includes the following metrics:

- EDM - the overall EDM score

Model	EDM	EDM_P	EDM_A	Start EDM	Start EDM_A	Smatch
bart-base	70.01	74.26	65.76	71.72	69.17	81.24
bart-base-tuned	70.65	74.61	66.69	72.28	69.96	82.39
pt-bart-base	**91.12**	**93.08**	**89.16**	**91.89**	**90.70**	91.60
bart-large	64.72	68.83	60.61	66.68	64.53	80.46
pt-bart-large	89.73	91.69	87.78	90.59	89.50	**92.59**

Table 3: EDM and Smatch scores of models. pt- indicates that the models was pretrained. -tuned indicates that the models was trained using the hyperparameters found during the hyperparameter search

Model	EDM	EDM_P	EDM_A	Smatch
pt-bart-base	91.12	**93.08**	**89.16**	91.60
pt-bart-large	89.73	91.69	87.78	**92.59**
AE RNN [3]	85.48	88.14	82.20	86.50
Hitachi [26]	**93.56**	-	-	-

Table 4: Comparison between our models, Buys and Blunsom's Arc Eager Transition based parser and the Hitachi System's results for EDS

- EDM_P - EDM score for predicates (nodes)
- EDM_A - EDM score for arguments (edges)
- Start EDM - overall EDM score which requires only the start alignment span to match [3]
- Start EDM_A - Start EDM for arguments
- Smatch - the overall Smatch score

As seen in Table 3 the pt-bart-base model achieved the best results for all EDM scores while the pt-bart-large model achieved the highest Smatch score. This is a slightly unexpected result as the larger model should be more accurate in general. In addition to this, given that the large model achieved a slightly lower eval and train loss compared to the base model as seen in Appendix A, the lower EDM scores may be due slight overfitting in the large model.

Looking at the start EDM metrics we can see that pt-bart-large model is slightly less accurate at predicting node ends spans compared to that of the pt-bart-base model. The large model has a 1.72 difference between EDM_A and start EDM_A while the base model has a difference of 1.54.

The pt-bart-large model achieved a Smatch score of 92.59 compared to 91.60 of the pt-bart-base model. This indicates that the large model was better at predicting the graph structure. However, this is also an indication that the large model is less accurate at predicting node spans than the base model in general as the base model outperformed the large model in EDM scores which take node spans into account.

6.2 Pretrained vs Non-pretrained models

As expected both pretrained models greatly outperform their non-pretrained counterparts with a difference of 21.11 and 25.01 for EDM score and 10.36 and 12.13 for Smatch score for pt-bart-base and pt-bart-large respectively. Furthermore the hyperparameter tuned model bart-base-tuned only improved the results by an average of 0.74 points across all metrics. This could be attributed to the limited number and depth of the trials that were able to be run during the hyperparameter search (see §9). We however, believe that the understanding of the English language that is encoded into the pretrained models greatly influences the models ability to predict node and edge labels as these labels are closely related to the role an English word plays in the sentence which the pretrained model should have some understanding of.

6.3 Comparison to Other Work

We compare the results of our two pretrained models to that of Buys and Blunsom's Arc Eager Transition-based (AE RNN) model and The Hitachi System's results for EDS shown in Table 4.

Buys and Blunsom's AE RNN model outperformed their Top Down Graph Linearization (TD RNN) model and therefore only the results of AE RNN are reported in their paper for EDS. Both our models and Buys and Blunsom's are trained on the same data as well as tested on the same data, the results are therefore directly comparable. As seen in Table 4 both our base and large model outperformed the AE RNN model across all metrics and hence also their TD RNN model, notably pt-bart-base achieved a 5.64 greater EDM score and pt-bart-large achieved a 6.09 greater Smatch score.

When comparing our models to the Hitachi System while both were trained on the same training data the Hitachi System uses a slightly different data set containing data mixed in from other sources for validation and testing. We therefore, cannot directly compare the models. However, it is possible to assert some conjectures. Given that the test set for the Hitachi System is larger and consists of mixed texts it can be argued that the results of the Hitachi System are more accurate and the model has a better ability to generalize. While the EDM score of the Hitachi System is 2.44 points higher than our current models' we hypothesize that the BART model has the potential to outperform the Hitachi System (see §9) this however, requires further research to support this claim.

7 CHALLENGES AND RECOMMENDATIONS

The training of the models in this paper was implemented using the Huggingface Trainer API. This API greatly abstracts from the underlying training loop and prediction. While this may be good for common neural network tasks, for specialized fine tuning tasks such as the task in this paper, the high level of abstraction makes it a necessity to have a high level knowledge of the fundamentals of training a neural network. This led to many deep dives into the documentation and extra research whereas if the implementation was done in native PyTorch many configurations would be required parameters. We therefore recommend for any specialized task to use

a native machine learning framework so that nothing is overlooked because of abstraction.

8 CONCLUSIONS

Through our experiments we have shown that our BART-based models outperforms the previous RNN-based model improving EDM and Smatch score by 5.64 and 6.09 respectively. Furthermore our results show that the use of pretraining greatly impacts the performance of our models with the pretrained model outperforming the non-pretrained model by 21.11 and 25.01 for EDM score and 10.36 and 12.13 for Smatch score for the base and large models respectively. Finally our model achieves a EDM score 2.44 points lower than that of the state of the art Hitachi System, we therefore hypothesize that the BART has the potential to outperform the Hitachi System if further optimized.

9 LIMITATIONS AND FUTURE WORK

Due to the tight timeline allocated to this project many optimizations were omitted as well as the investigation of other graph linearization approaches. We therefore leave the following as future work:

9.1 Hyperparameter Tuning

Due to the long training times of the models and the maximum wall time of 12 hours on the CHPC hyperparameter tuning was only done for the non-pretrained `bart-base` model. In addition only a small number of trials were performed with a low number of epochs per trial. All models can therefore be further optimized with a more thorough hyperparameter search.

9.2 Transition-based Graph Linearization

In [3] the transition-based graph linearization outperformed the top-down graph linearization approach, therefore the next logical step would be to verify if this is also the case for the BART model.

9.3 Special Token-based Graph Linearizations

Both [2] and [26] made use of special tokens in their graph linearizations to make it easier for the model to learn the graph structure this is therefore an optimization that can be explored in future work.

REFERENCES

[1] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract Meaning Representation for Sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*. Association for Computational Linguistics, Sofia, Bulgaria, 178–186. https://aclanthology.org/W13-2322

[2] Michele Bevilacqua, Rexhina Blloshmi, and Roberto Navigli. 2021. One SPRING to Rule Them Both: Symmetric AMR Semantic Parsing and Generation without a Complex Pipeline. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 14 (May 2021), 12564–12573. https://ojs.aaai.org/index.php/AAAI/article/view/17489

[3] Jan Buys and Phil Blunsom. 2017. Robust Incremental Neural Semantic Graph Parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, 1215–1226. https://doi.org/10.18653/v1/P17-1112

[4] Shu Cai and Kevin Knight. 2013. Smatch: an Evaluation Metric for Semantic Feature Structures. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, Sofia, Bulgaria, 748–752. https://aclanthology.org/P13-2131

[5] Ann Copestake. 2009. Invited Talk: Slacker Semantics: Why Superficiality, Dependency and Avoidance of Commitment can be the Right Way to Go. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*. Association for Computational Linguistics, Athens, Greece, 1–9. https://aclanthology.org/E09-1001

[6] Ann Copestake, Dan Flickinger, Carl Pollard, and Ivan Sag. 2005. Minimal Recursion Semantics: An Introduction. *Reseach On Language And Computation* 3 (07 2005), 281–332. https://doi.org/10.1007/s11168-006-6327-9

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[8] Rebecca Dridan and Stephan Oepen. 2011. Parser Evaluation Using Elementary Dependency Matching. In *Proceedings of the 12th International Conference on Parsing Technologies*. Association for Computational Linguistics, Dublin, Ireland, 225–230. https://aclanthology.org/W11-2927

[9] Dan Flickinger. 2000. On building a more effcient grammar by exploiting types. *Natural Language Engineering* 6 (03 2000), 15–28. https://doi.org/10.1017/S1351324900002370

[10] Dan Flickinger, Yi Zhang, and Valia Kordoni. 2012. DeepBank : a dynamically annotated treebank of the Wall street journal. In *Proceedings of the Eleventh International Workshop on Treebanks and Linguistic Theories (TLT11)*. Lisbon, 85–96. HU.

[11] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *SIGPLAN Not.* 39, 1 (jan 2004), 111–122. https://doi.org/10.1145/982962.964011

[12] Yoav Goldberg and Joakim Nivre. 2012. A Dynamic Oracle for Arc-Eager Dependency Parsing. In *Proceedings of COLING 2012*. The COLING 2012 Organizing Committee, Mumbai, India, 959–976. https://aclanthology.org/C12-1059

[13] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural computation* 9, 8 (1997), 1735–1780.

[14] Daniel Jurafsky and James H. Martin. 2000. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition* (1st ed.). Prentice Hall PTR, USA.

[15] Robert T. Kasper. 1989. A Flexible Interface for Linking Applications to Penman's Sentence Generator. In *Speech and Natural Language: Proceedings of a Workshop Held at Philadelphia, Pennsylvania, February 21-23, 1989*. https://aclanthology.org/H89-1022

[16] Paul Kingsbury and Martha Palmer. 2002. From TreeBank to PropBank. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC'02)*. European Language Resources Association (ELRA), Las Palmas, Canary Islands - Spain. http://www.lrec-conf.org/proceedings/lrec2002/pdf/283.pdf

[17] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 7871–7880. https://doi.org/10.18653/v1/2020.acl-main.703

[18] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics* 19, 2 (1993), 313–330. https://aclanthology.org/J93-2004

[19] Ryan McDonald and Joakim Nivre. 2007. Characterizing the Errors of Data-Driven Dependency Parsing Models. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. Association for Computational Linguistics, Prague, Czech Republic, 122–131. https://aclanthology.org/D07-1013

[20] Joakim Nivre. 2008. Algorithms for Deterministic Incremental Dependency Parsing. *Computational Linguistics* 34, 4 (2008), 513–553. https://doi.org/10.1162/coli.07-056-R1-07-027

[21] Joakim Nivre. 2009. Non-Projective Dependency Parsing in Expected Linear Time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*. Association for Computational Linguistics, Suntec, Singapore, 351–359. https://aclanthology.org/P09-1040

[22] Stephan Oepen, Omri Abend, Lasha Abzianidze, Johan Bos, Jan Hajic, Daniel Hershcovich, Bin Li, Tim O'Gorman, Nianwen Xue, and Daniel Zeman. 2020. MRP 2020: The Second Shared Task on Cross-Framework and Cross-Lingual Meaning Representation Parsing. In *Proceedings of the CoNLL 2020 Shared Task: Cross-Framework Meaning Representation Parsing*. Association for Computational Linguistics, Online, 1–22. https://doi.org/10.18653/v1/2020.conll-shared.1

[23] Stephan Oepen, Omri Abend, Jan Hajic, Daniel Hershcovich, Marco Kuhlmann, Tim O'Gorman, Nianwen Xue, Jayeol Chun, Milan Straka, and Zdenka Uresova. 2019. MRP 2019: Cross-Framework Meaning Representation Parsing. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 Conference on Natural Language Learning*. Association for Computational

Linguistics, Hong Kong, 1–27. https://doi.org/10.18653/v1/K19-2001

[24] Stephan Oepen, Dan Flickinger, Kristina Toutanova, and Christopher D. Manning. 2004. LinGO Redwoods. *Research on Language and Computation* 2, 4 (01 Dec 2004), 575–596. https://doi.org/10.1007/s11168-004-7430-4

[25] Stephan Oepen and Jan Tore Lønning. 2006. Discriminant-Based MRS Banking. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*. European Language Resources Association (ELRA), Genoa, Italy. http://www.lrec-conf.org/proceedings/lrec2006/pdf/364_pdf.pdf

[26] Hiroaki Ozaki, Gaku Morio, Yuta Koreeda, Terufumi Morishita, and Toshinori Miyoshi. 2020. Hitachi at MRP 2020: Text-to-Graph-Notation Transducer. In *Proceedings of the CoNLL 2020 Shared Task: Cross-Framework Meaning Representation Parsing*. Association for Computational Linguistics, Online, 40–52. https://doi.org/10.18653/v1/2020.conll-shared.4

[27] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 28)*, Sanjoy Dasgupta and David McAllester (Eds.). PMLR, Atlanta, Georgia, USA, 1310–1318. https://proceedings.mlr.press/v28/pascanu13.html

[28] Alec Radford and Karthik Narasimhan. 2018. Improving Language Understanding by Generative Pre-Training.

[29] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners.

[30] Kenji Sagae and Jun'ichi Tsujii. 2008. Shift-Reduce Dependency DAG Parsing. In *Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008)*. Coling 2008 Organizing Committee, Manchester, UK, 753–760. https://aclanthology.org/C08-1095

[31] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 1715–1725. https://doi.org/10.18653/v1/P16-1162

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[33] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6

[34] Jiawei Zhou, Tahira Naseem, Ramón Fernandez Astudillo, Young-Suk Lee, Radu Florian, and Salim Roukos. 2021. Structure-aware Fine-tuning of Sequence-to-sequence Transformers for Transition-based AMR Parsing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 6279–6290. https://doi.org/10.18653/v1/2021.emnlp-main.507

A EVALUATION AND TRAINING LOSS OF MODELS

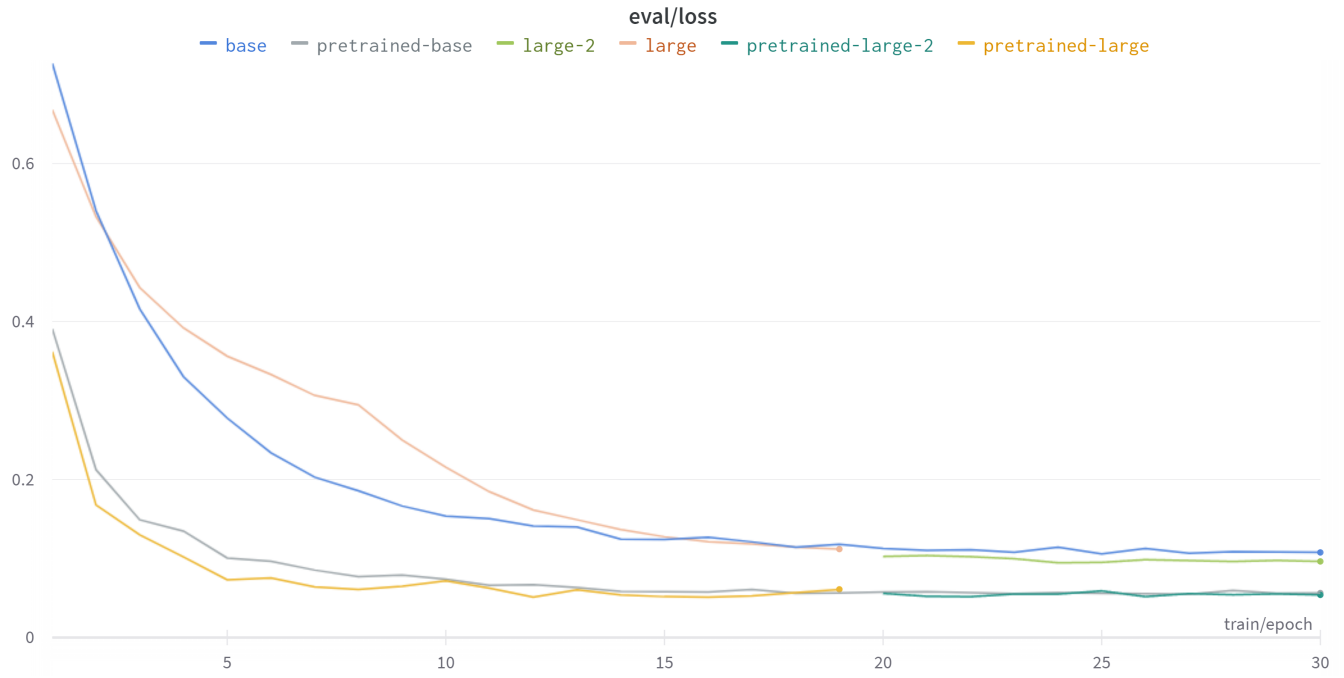

Figure 6: Evaluation loss per Epoch

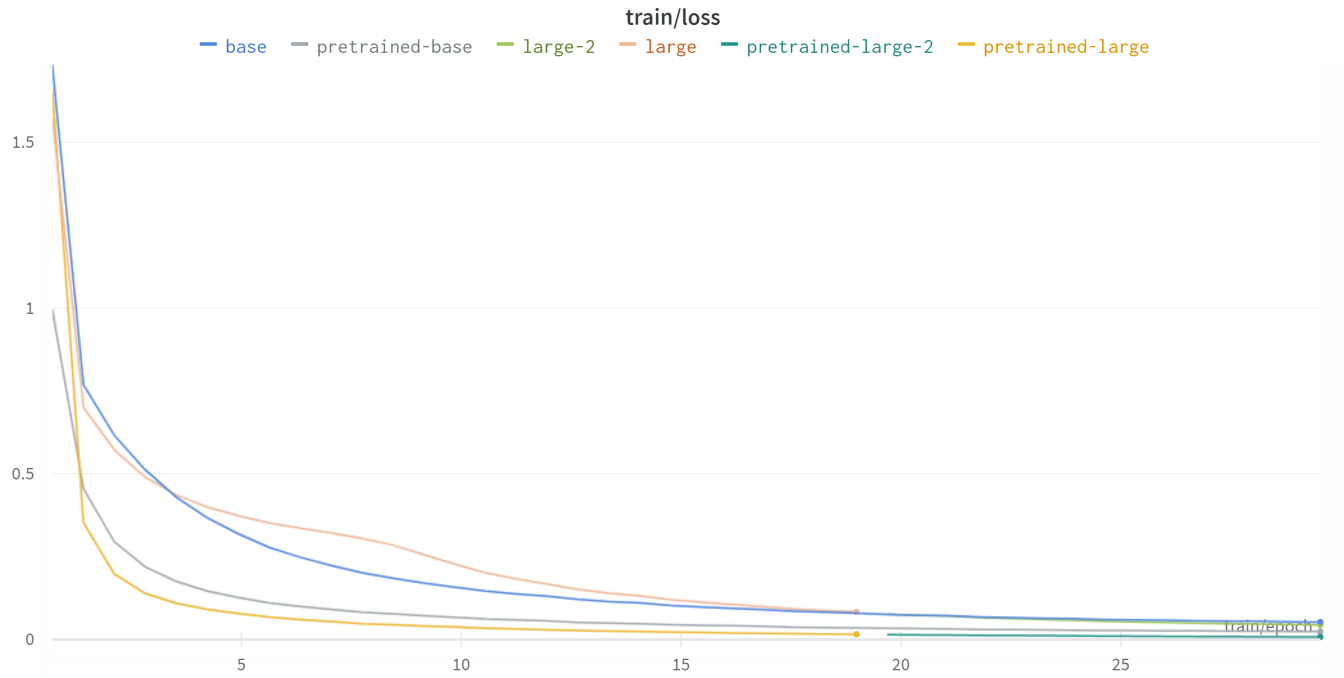

Figure 7: Train loss per Epoch

B EVALUATION LOSS AND HYPERPARAMETER SEARCH FOR NON-PRETRAINED MODEL

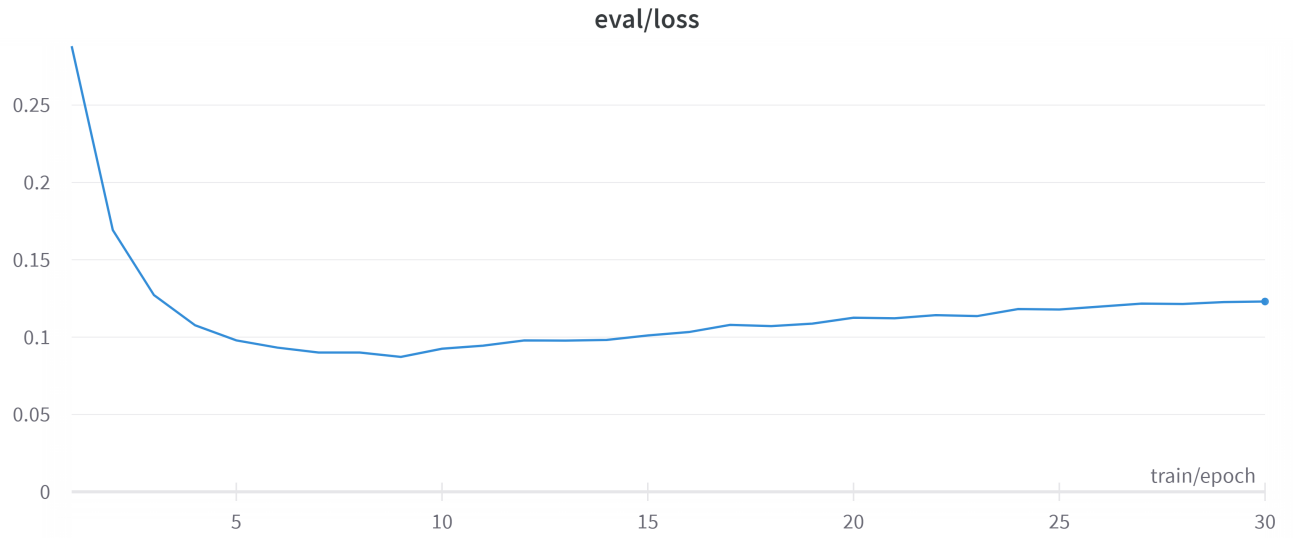

Figure 8: Evaluation loss per Epoch

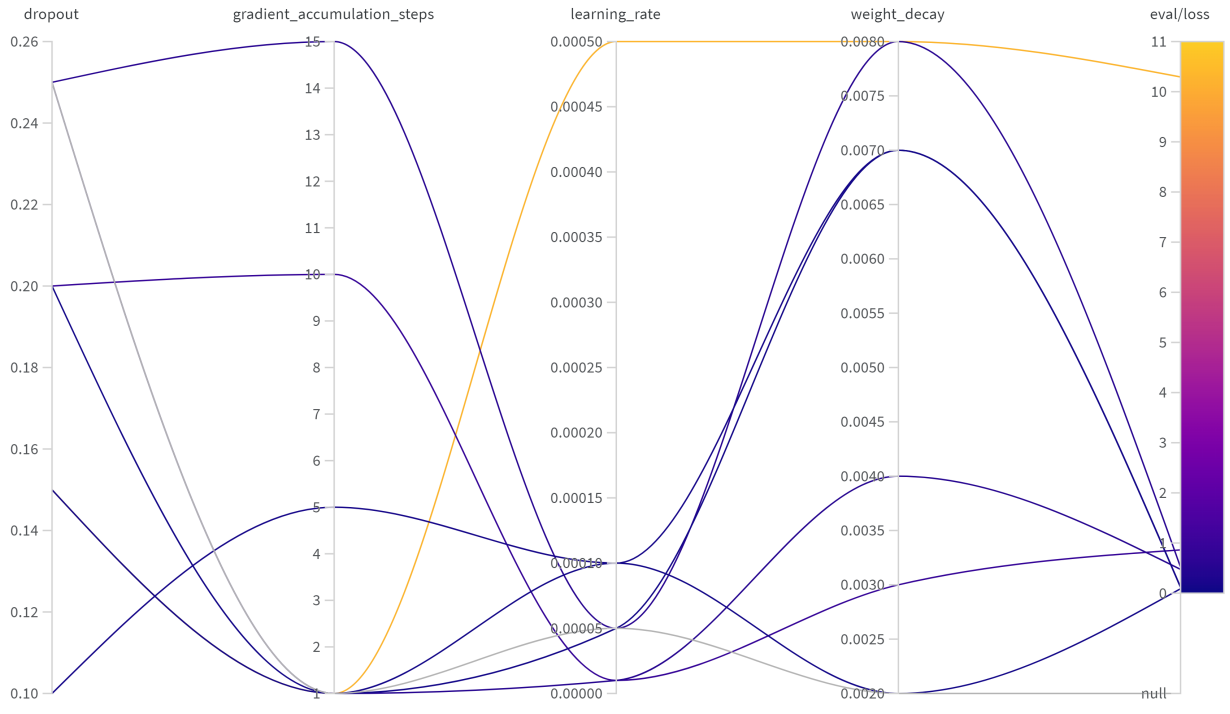

Figure 9: Random Search

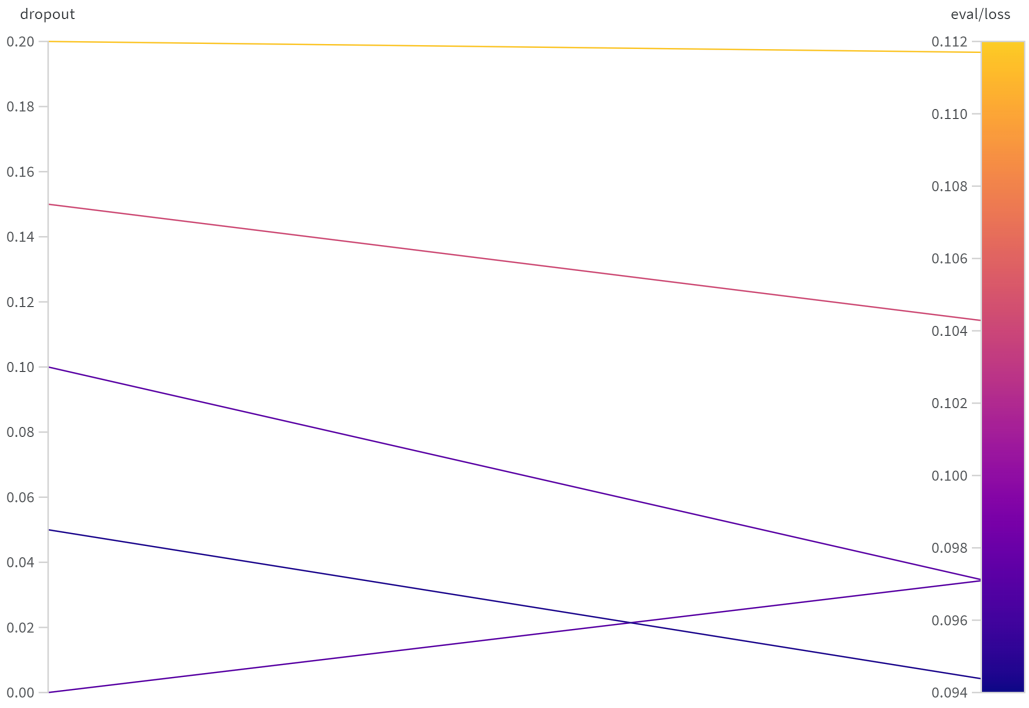

Figure 10: Grid Search