Replication of "Reasoning about Pragmatics with Neural Listeners and Speakers" - J. Andreas and D. Klein (2016)

Nils Riekers, 5742398
nils.riekers@student.uni-tuebingen.de

Wolfgang França Dantas, 5717688
wolfgang-michael.franca-dantas@student.uni-tuebingen.de

***Abstract*—In this project, we re-implemented the model presented in the paper "Reasoning about Pragmatics with Neural Listeners and Speakers" by Andreas and Klein using Apollocaffe. Our main goals were to successfully run the model, provide detailed documentation of the implementation, and perform hyperparameter optimization. We also considered migrating the code to PyTorch as an optional task. By achieving these goals, we aimed to improve the usability and accessibility of the model, and to contribute to the improvement of human-machine interaction in natural language, emphasising the importance of pragmatics and context in language generation.**

I. Introduction

In recent years, machine learning, especially in the form of deep learning, has made impressive progress in several areas of artificial intelligence. One of these areas is natural language processing, which enables human-machine interaction in natural language. An interesting problem in this area is the generation of contextual descriptions of scenes and objects by reasoning about the context and behaviour of the listener. In this project, we are concerned with reimplementing the paper "Reasoning about Pragmatics with Neural Listeners and Speakers" by Jacob Andreas and Dan Klein, which proposed a novel approach to this problem [1].

II. Purpose of the Project

The main goal of our project is to reimplement the model presented in the paper, with the following objectives:

1. System Environment: first, we want to successfully run the model with Apollocaffe and create detailed documentation about the whole process.

2. Implementation Details: Since the paper is quite vague in many places and describes the content on an abstract level, we would like to explain the implementation details of the code and create documentation on how to implement the model concretely. We will focus on the representation of the descriptions and the abstract scenes.

3. Hyperparameter Optimization: In order to improve the existing Apollocaffe model, we would like to perform hyperparameter optimization using a "grid search" procedure.

4. Migration to PyTorch (optional): As an additional optional task, we considered migrating the Apollocaffe code to PyTorch in order to make the model available in a more modern and widely used deep learning library. However, this step was not fully implemented due to time and resource constraints, but remains an interesting possibility for future work.

By successfully reimplementing the model and achieving the above goals, we hope to gain a deeper understanding of the methods presented in the paper and their practical applications. In addition, we aim to increase the usability and accessibility of the model to other researchers and developers by providing detailed implementation guidance. Ultimately, our project contributes to the further improvement of human-machine interaction in natural language and highlights the importance of pragmatics and context in language generation.

III. Code Replication

The core of the paper "Reasoning about Pragmatics with Neural Listeners and Speakers" by Jacob Andreas and Dan Klein is a fork of the obsolete deep learning framework Caffe, called ApolloCaffe, which was used for the implementation [1].

A. System Environment

In order to get the framework working, we have tested different system environments. Unfortunately, despite our best efforts, we were unable to install ApolloCaffe on Ubuntu 20.04 or 22.04 in a virtual machine.

According to the official Caffe documentation, it is recommended to install and run Caffe on Ubuntu 16.04-12.04, OS X 10.11-10.8 and via Docker and AWS [2]. However, unlike Caffe, there is no corresponding Docker image for ApolloCaffe, and the AWS AMI (Amazon Machine Image) mentioned in the description no longer exists.

We have gathered several possible reasons why we have not been able to get ApolloCaffe to run successfully on newer Ubuntu distributions. First, there is an incompatibility of some libraries in newer Ubuntu distributions. Many dependencies are outdated, no longer available by default through package managers like apt or pip, or have to be installed manually. Also, many files are no longer in the same paths and would need to be symlinked to the new locations. Further difficulties arise from the fact that Caffe and ApolloCaffe use the outdated Python 2.7 by default, mainly via Numpy as a programming interface. The default Python version on Ubuntu 16.04 is Python 2.7, and for these reasons we decided to use Ubuntu

16.04 in a virtual machine, as this is the latest of the Ubuntu distributions recommended for Caffe.

First, we tried to create a suitable system environment in a virtual machine running Ubuntu 16.04. We did this locally on our computers using VirtualBox virtualisation software. VirtualBox is an open source virtualisation software that allows multiple operating systems to run on a single host computer [3]. However, we found that training the model was ineffective due to resource constraints. For this reason, we switched to a virtual machine within the Google Cloud Platform, which is available as an infrastructure-as-a-service offering. The Google Cloud Platform is a cloud computing platform that is developed and operated by Google and offers a wide range of services [4].

B. Dependencies of ApolloCaffe

In order to install Apollocaffe, several dependencies are needed. But first, in order to use the framework in this project, we had to clone the Apollocaffe repository and switch to its directory using these commands:

```
git clone git@github.com:jacobandreas/
    apollocaffe.git
cd apollocaffe
```

APT (Advanced Packaging Tool) is a package manager for Linux-based operating systems that allows easy installation and management of packages and dependencies [5]. We have installed the following packages using the APT package installer:

```
sudo apt-get install libprotobuf-dev
    libleveldb-dev libsnappy-dev
    libopencv-dev libboost-all-dev
    libhdf5-serial-dev protobuf-compiler
    libatlas-base-dev libopenblas-dev
    python-dev python-numpy
    libgoogle-glog-dev libgflags-dev
    liblmdb-dev
```

In addition, we also installed the following Python related packages using APT:

```
sudo apt-get install python-dev
    python-pip python-scipy python-h5py
    python-numpy
```

The file ”requirements.txt” under ”apollocaffe/python/” tells you which other Python packages need to be installed. Pip is a package manager for Python that allows one to install and manage Python packages and dependencies [6]. We were able to install the required packages using pip with the following shell command:

```
for req in $(cat requirements.txt); do
    pip install $req; done
```

However, we found that two of these packages are not fully backwards compatible in the newer versions. For this reason, we have installed them in an older version:

```
sudo apt-get install build-essential
    autoconf libtool
pip install protobuf==3.15.8
pip install pyyaml==3.13
```

It should be noted that we were able to perform this installation in the VM with the Google Cloud Ubuntu 16.04 image without any further problems. However, in the local VM there were problems because Pip was accessing outdated package sources. We tried to revert to a newer package source, but an SSL error occurred that we could not resolve without further problems. Our solution was to manually install the required Python packages from source. This requires the following steps:

```
wget <link to the archive>
tar zxvf Package-Name.tar.gz
cd Package-Name
python setup.py install
```

C. Compilation of ApolloCaffe

Once all the dependencies are installed, Apollocaffe can be compiled. However, in our local VM, we had to enable RAM swapping first, as our VM was running out of memory and the compilation process was aborted. We placed a bash script called (name) in our repository to enable this.

To compile Apollocaffe, we first needed to modify the ”Makefile.config.example” file and save it as a new file called ”Makefile.config”. We set the flag ”CPU_ONLY := 1” because there was no graphics card available on our local VM or in the cloud without further configuration. We also added the path ”/usr/include/hdf5/serial/” to the ”INCLUDE_DIRS” variable, as the necessary header files for the HDF5 library were not automatically included during compilation. We also had to add the path ”/usr/lib/x86_64-linux-gnu/hdf5/serial/” to the ”LIBRARY_DIRS” variable, because the linker could not find the corresponding library files during compilation.

The actual installation of Apollocaffe can then be compiled using the ”make all” command. The compilation process will take some time. The ”make test” command can then be used to run the tests, and ”make runtest” can be used to run the tests and display the results. Finally, to use Apollocaffe in Python, two environment variables have to be set. The ”.bashrc” is a configuration file for the bash shell that is executed when the terminal is started. To make these environment variables automatically available in a new shell session, we added them to the file ” /.bashrc”.

```
export LD_LIBRARY_PATH=/home/<
    USER_HOME_DIR>/git/apollocaffe/build/
    lib:$LD_LIBRARY_PATH\
export PYTHONPATH=/home/<USER_HOME_DIR>/
    git/apollocaffe/python:$PYTHONPATH
```

D. Running the Model

In order to run the model with the now built Apollocaffe, some preparations have to be made first. First, the ”Abstract Scenes Dataset” by Zitnick and Parikh (2013) [7], used in

the paper for the reference game, is required. This dataset contains abstract images for studying semantic inference and similarity by examining the dependence of visual features, and the potential use of Conditional Random Fields to generate scenes [8]. We have downloaded this and unpacked it in the correct arrangement of files in a created folder called "data".

In addition, another folder called "models" had to be created. This folder will contain the trained models that will be used for inference. This folder is not created automatically by "main.py".

Since we do not have a GPU as described, we also had to comment out the line "apollocaffe.set_device(0)" within the "main.py". This line specifies which GPU to use.

After that the "main.py" can be started with the correct passing parameters. There is also a pre-built bash script called "run.sh" for this purpose. This just calls "main.py" again, takes the parameters and sets the environment variables described above. It is not documented with which parameters one needs to call "main.py" or the "run.sh" script, but it can be determined from various "if-statements" in the code. It needs a name for the values of the variables "corpus" and "job". Corpus describes the data set to be used (either "abstract" or "birds"). Job describes the action to be performed, e.g. "train.base" to train a base model or "sample.base" to perform the experiments described in the paper.

If you want to train the model on the abstract data set, then the script can be started with the command:

```
python main.py train.base abstract
```

Similarly, one can use the trained model to run the experiments with the following command:

```
python main.py sample.base abstract
```

IV. Details of Implementation

Implementation details of the reference implementation of paper *Reasoning about Pragmatics with Neural Listeners and Speakers*.

A. Motivation

In their paper "Reasoning about Pragmatics with Neural Listeners and Speakers", Jacob Andreas and Dan Klein described a theoretical model for contrastively describing scenes, in which context-specific behaviour results from a combination of inference-driven pragmatics and learned semantics. To achieve this, the model was tailored to play the role of a reasoning speaker in a reference game. In their paper, the authors explain their model in principle: They give equations or use strict mathematical notation yet always stay on a rather abstract level. This is particularly evident when it comes to the format of the input data where access to feature representations is just assumed. Despite the model being "completely indifferent to the nature of this representation" ($\rightarrow$ chapter *3.1 Preliminaries* in the original paper), it is crucial as sub-modules of the model (e.g., the referent describer D) make particular assumptions about what features are needed but it is up to the user to find a matching implementation. Furthermore, there are several files from the abstract scenes data set used which would qualify to draw the feature representations from which requires experienced users to make the right decisions for the representation choice to avoid later incompatibilities between sub-modules.

In this section, we therefore describe what implementation choices the authors made when they implemented their model in Python. This can serve as guideline or inspiration for own implementations in case different data might want to be used and to further the understanding of how this model works in general.

In the following explanations, we will stick to the notation and naming used in the paper as closely as possible. Furthermore, we **extensively commented the original source code** (after automatically migrating it from Python 2 to Python 3 to get it to run) with the same information provided as in the following paragraphs as well as explanations about intermediate detail steps which happen either during data preparation or while working with the models (`main.py` and `modules.py`).

B. Overall Structure of the Full Model

The model presented by Andreas and Klein as well as their reference implementation follow a modular approach. There are three major classes for the literal listener L0, the literal speaker S0, and the sampling speaker S1 which make calls to sub-modules. These sub-modules are defined in section *3.2 Modules* in the paper and implemented in `modules.py` as classes with neuronal networks. As the same sub-modules are used in several major classes, one advantage is re-usability. Another advantage is flexibility as sub-modules, which all were implemented providing the same interfaces, can be exchanged. For example, to try different encoders for description or referent embeddings. The sub-modules described in the paper and used in the code are:

- A referent encoder E_r to create image embeddings.
- A description encoder E_d to create description embeddings.
- A choice ranker R which is used to choose an abstract scene based on a probability distribution.
- A referent describer D to generate new scene descriptions.

We explain all major classes as well as the sub-modules in details when we encounter them in the text.

C. Data Formats and Data Loader: Raw Data formats

1) Anatomy of raw scene data:: In the abstract scenes data set in version 1.1 [7], `Scenes_10020.txt` is a text file containing the information needed to render all the scenes (=images) in the directory *RenderedScenes*. These scenes were used in the online experiments of Andreas and Klein. Each abstract scene is described as a set of *props* (Fig. 1) that all have the same attributes; just with different values: At first, a non-unique scene ID and the number of props used are given. Each following line specifies one of these props:

- Name of the .png-file of the prop. However, this value is not used. (column 1).
- Which prop to be used as defined through clipart name and index type (columns 2 and 3).
  - The clip art type index can take 8 different values which is reflected by the constant `N_PROP_TYPES` in the first few lines of `modules.py`.
  - The clip art name can take 35 different values, that is, there are at maximum 35 different pictures of one type. For example, there are 6 animal type pictures and 35 pictures showing a boy. This is reflected by the constant `N_PROP_OBJECTS` in `modules.py`.
- x-, y-, and z-coordinates of the prop (columns 4, 5, 6)
- Is the prop in the scene horizontally flipped (yes or no)?

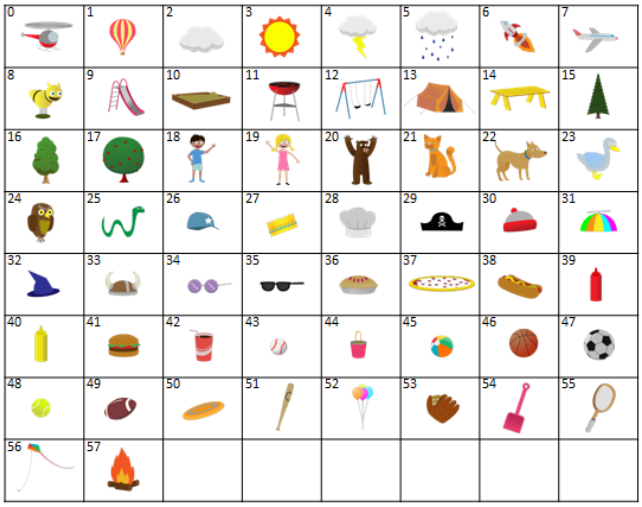

Fig. 1: Example of some available props to appear in abstract scene images (incomplete list; image taken from abstract scenes data set [7])

2) Anatomy of raw description (=sentence) data: Descriptions, that is, sentences captioning a scene, for initial training and evaluation of the model originate from `SimpleSentences1_10020.txt` and `SimpleSentences2_10020.txt` which both are part of the abstract scenes version 1.1 data set. Each file contains a list of 30,000 sentences which describe certain scenes. There are 3 sentences per scene.

D. Data Formats and Data Loader: Data preparation and loading

1) Basic aggregating data structures: Andreas and Klein used `namedtuples` as basic aggregating data structures which the raw data were converted into. These structures were used throughout the code to work with scenes and descriptions. *Props* were structured as follows:

```
Prop = namedtuple("Prop", ["type_index",
    "object_index", "x", "y", "z",
    "flip"])
```

The square brackets enclose the column headings and correspond to the *prop* description above.

Scenes comprise a unique image ID, a list of several *props* with one entry per *prop*, a description, which is a list of integers indexing words in the global vocabulary corpus, and, finally, a features item which is not used in the context of the abstract scenes data set:

```
Scene = namedtuple("Scene", ["image_id",
    "props", "description", "features"])
```

2) Feature representation of a description $f(d)$: The representation of the description of an abstract scene image is implemented as a vector of indicator features on n-grams, that is, as a binary vector that represents the presence or absence of each possible n-gram in the global vocabulary. Here, it is a vector that represents the number of occurrences of each possible n-gram in a text corpus. This happens in `corpus.py` in the function `load_scenes()` which first parses `SimpleSentences1_10020.txt` and appends a word/token at the end of the global vocabulary (instantiated as variable `WORD_INDEX` in indices.py) if it has not been added yet. After parsing this file, there are 1063 different words/tokens in the global vocabulary, including start and stop tokens. Next, it generates description representations for each scene. Here, a representation is a list of indices (counting from 1). Each index corresponds to an entry in the vocabulary. Instead of the word, its index in the global vocabulary is used. These lists are stored in the *description*-column of the *scenes*.

For example, the scene with `image_id='707_5'` has the description `[1, 2, 248, 11, 295, 14]`. While being parsed, all descriptions are enclosed in start and end tokens <s> and </s>, thus, all description representations have the same first and last index numbers in the array, i.e., 1 and 14. By looking up the indices in the vocabulary, we can retrieve the description attached to the scene in human-understandable representation: "<s> jenny threw the basketball </s>".

The actual representation is generated in the referent encoder (equation (1)): From what we have discussed so far, it becomes clear why a description is represented as a 1×1063 vector. Each position corresponds to a word in the global vocabulary. As one word can (and actually sometimes does) occur several times, it can contain any positive number equal to or bigger than zero. The model is trained on mini batches of set size 100 (set in `main.py` in the YAML-variable `CONFIG.opt.batch_size`). Thus, batches of descriptions are represented as matrices of shape 100×1063.

3) Feature representation of a referent $f(r)$: These representations are created in the data loader corpus.py over the course of several function calls: `load_props()` is the starting point reading all the props that constitute a certain image from the abstract scenes data base `Scenes_10020.txt`. At this point, the data only holds descriptions what *props* each scene contains. Next,

these data are passed to `normalize_props()` which normalises position and flipping state amongst all props of all scenes. Finally, `load_scenes()` assembles all the information gathered so far and fills the `image_strid`, `props`, `word_ids`, and `features` attributes of each *scene*. The `load_abstract()` function splits this bulk of data and returns three sub sets to the caller: One for training, development, and test. These sets are used throughout the rest of the code and have to be accessed according to the respective `namedtuple` structures of *props* and *scenes*.

The actual representation is generated in the description encoder (equation (2)): From what we discussed earlier follows that there is a theoretical total of `N_PROP_TYPES` · `N_PROP_OBJECTS` = 280 different clip arts. Actually, there are not 35 different images for each type. To reflect the theoretically possible variety of clip arts, each scene is represented as a binary 1×280 vector. Scenes are mini-batched too with the same number of mini batches as the descriptions. Thus, batches of scenes are represented as matrices of shape 100×280.

E. Literal Listener L0

The literal listener L0 takes a description and a set of referents, and chooses the referent (i.e., abstract scene) most likely to be described. To this, it transforms description and referent (=scene) representations into linear embeddings using the description encoder (equation (1)) and the referent encoder (equation (2) in the paper). These linear embeddings are fed into the choice ranker R which produces a probability distribution over referents, that is, it takes a string encoding and a collection of referent encodings, assigns a score to each (string, referent)-pair, and finally transforms these scores into a distribution over referents (equations (3) and (4)).

Implementation:

L0 is implemented in `main.py` as class `Listener0Model` and only has a `forward()` function with the following parameters:

- `data`: Target scenes (i.e., referents coming from text-based descriptions in `Scenes_10020.txt` which are the basis of the rendered images).
- `alt_data`: Distractor scenes (i.e., other referents) as the model is trained contrastively.

The literal listener's main constituents, as stated in equation (4) in the paper, are:

1) Referent encoder E_r (equation (1)): It is implemented in `modules.py` as class `LinearSceneEncoder` whose `forward()` takes mini batch of scenes as produced in `corpus.py`. The referent encoder is a small neuronal net with only one fully connected layer with 280 inputs and an output layer size of 100 (defined in the YAML-constant `CONFIG.model.hidden_size` in `main.py`). There is one difference between the implementation and equation (1) in the paper: The actual implementation from Andreas and Klein does have a bias vector, unlike stated in the paper.

2) Description encoder E_d (equation (2)): It is implemented in `modules.py` as class `LinearStringEncoder`. All interaction happens through its `forward()` function whose only relevant input parameter has the confusing name `scenes` even though descriptions,i.e., sentences to be encoded, must be passed. It is implemented as a neuronal net with one fully connected layer with 1063 inputs and an output layer size of 100 (defined in the YAML-constant `CONFIG.model.hidden_size` in `main.py`). Again, there is a bias vector included in contrast to equation (2) in the paper.

3) Choice ranker R (equation (3)): The choice ranker is implemented in `modules.py` as class `MlpScorer`.It takes a string encoding and a collection of referent encodings, assigns a score to each (string, referent) pair, and then transforms these scores into a distribution over referents.

All interaction happens through its `forward()` function whose following input parameters are relevant:

- `l_query`: Embedding of a description (i.e., scene caption) of shape torch.Size([100, 100]).
- `ll_targets`: Embedding of scenes (target and distractor). It is a list of two tensors each of shape torch.Size([100, 100]).
- `labels`: Always passed as array of zeros by the caller.

The embeddings s1 and s2 of the target and the distractor scenes that enter equation (3) are not computed separately in the implementation as indicated by the equations but rather their embeddings are concatenated as the code can deal with more than one distractor.

Next, the description embedding is copied several times such that there is one description embedding for each scene embedding. Their shapes have to match after this step.

Apparently, there is no linear transformation before the representations are added element-wise. That is, there were no multiplications with matrices W4 and W5 implemented in the code.

This step is followed by a *ReLU* non-linearity and a concluding fully connected layer, again with bias term.

The code also inputs this data to a *SoftMax* but, strangely, the output is not used. However, the result of the last fully connected layer is forwarded to a *LogSumExp* function. This matches the denominator of equation (3) just with a further application of a log-function.

The result corresponds to a distribution over referent choices (i.e., over scenes) as negative log-likelihoods. The sum of these numbers is used as loss during the training process of L0.

F. Literal Speaker S0

The literal speaker S0 takes a referent in isolation (i.e., single scene) and outputs a description. It is used for efficient inference over the space of possible descriptions, i.e., a neural captioning model. S0 contains a referent describer D from

which words can be sampled, i.e., new descriptions can be generated given a certain abstract scene embedding.

Implementation:

S0 is implemented in `main.py` as class `Speaker0Model` which consists of two functions:

- `forward()` is needed for training this base S0 model. It takes the following input parameters:
  - `data`: Target scene which was secretly assigned to the speaker.
  - `alt_data`: One or more distractor scenes. In the paper, there is only one distractor which also is the default value in the implementation.
- `sample()` is used to draw a sequence of words (c.f. step "1." of the reasoning model in section 3.4). The following input parameters are used:
  - `data`: As above.
  - `viterbi`: Determines the decoding scheme (explained later).

Both functions at first create a linear referent embedding and then make calls to the referent describer D which is the main workhorse here whose `forward()` and `sample()` functions are called respectively from `forward()` and `sample()` of L0.

Referent describer D:

The referent describer takes an image encoding and outputs a description using a (feed-forward) conditional neural language model.

In the paper it was described as "a '2-plus-skip-gram' model, with local positional history features, global position-independent history features, and features on the referent being described.´´ whose meaning was, at least to us, not clear from the beginning. This had changed when we looked at the code and commented it while our understanding of it was growing.

The referent describer was implemented in `modules.py` as class `MlpStringDecoder` and comprises two functions:

1) `forward()`: This function is used during training and to score samples in the reasoning speaker S1. As input parameters, it takes:

- `scenes`: contain the batched target scenes.
- `encoding`: is the linear embedding tensor of the targets (equation (1) in the paper).

The paper roughly outlines that *indicator features* are the basis of and the first structures created by the referent describer. Indicator feature $d_{<n}$ corresponds to variable `history_features`. It is implemented, conceptually, as follows: For every word in the scene, count how often any possible word occurs in n-grams of decreasing length. Indicator feature d_n corresponds to variable `last_features` and its implementation can be conceptualised as: For every word in the scene, count how often this word is used in a description. Another important variable, later used for loss computation, is `targets` which contains copies of the description (in the format of vocabulary indices) of each scene.

As an intermediate step, vectors of scores are computed by a multilayer perceptron:

1) Concatenation of all indicator features.
2) Fully connected layer with bias which transforms only the concatenated indicator features. According to the paper, however, these features should be processed together with a referent embedding (matrix $W7$) in one step but this only happens separately.
3) Concatenation of transformed indicator features and referent embedding.
4) Next, to obtain the scores, the result from the last step is passed through a *ReLU* function and transformed in a fully connected layer with bias vector (unlike multiplication with $W6$ suggests in the paper).
5) Finally, and unlike described in the paper, not just a probability distribution is generated but a multinomial logistic loss between the scores and the target scene description.

2) `sample()`: This function is used to make predictions, i.e., generate new descriptions as a sequence of words. Thus, it samples $d_1, ..., d_n \sim p_{S0}(\cdot|r_i)$. As input parameters, it takes:

- `encoding`: Encoding of abstract scene, i.e., referent encoding.
- `viterbi`: Decoding scheme.
  - `False` corresponds to greedy sampling: Randomly sample index of one word from the vocabulary. This is the default value in the pragmatic speaker S1 (`SamplingSpeaker1`).
  - `True` corresponds to pure sampling (non-deterministic; truly random).

The `sample()` function outputs a distribution over strings and a scene description. It is implemented, conceptually, as follows:

1) Compute sampling distribution p_{S0}: This is performed very similarly to how the scores are computed but with the difference, that the last layer actually is a *SoftMax* function which is as described in the paper and makes it easier to develop an intuitive understanding of this data's role in further steps.
2) Draw samples (i.e., words d_i) from probability distribution p_{S0}: Now, the decoding scheme comes into play and is determined by the parameter `viterbi` which the authors of the paper set to `False` in their implementation. In consequence, the index of one single word from the vocabulary is drawn at random. The result of this step are log-probabilities over vocabulary indices.
3) Create a candidate description, i.e., full sentence caption, for the image e_r: The list of sampled vocabulary indices is parsed and each index is replaced by the corresponding word or token from the global vocabulary. All tokens combined form the description of the referent.

G. Reasoning Speaker S1

The reasoning speaker S1 is implemented in `main.py` as class `SamplingSpeaker1Model`. It is defined as sampling neural reasoning speaker in section 3.4 of the paper and not trained itself but rather based on the separately trained S0 and L0 models which are central parts of S1. It mainly operates by calling functions of S0 and L0. Interaction with S1 happens through its only function `sample()` which implements steps 1., 2., and 3. of the reasoning model as defined in section 3.4 of the Andreas and Klein paper:

1) Draw sample, i.e., a sequence of single word samples d_k: The reasoning speaker S1 makes a call to the literal speaker's `sample()` function to draw samples and obtain a distribution over words given one input image.
2) Score samples: S1 determines which target image (referred to as *i* in the paper) most likely is best described by the sampled word d_k. To this, it calls the literal listener's `forward()` method.
3) Select next, most likely best, word by re-weighting the scores: S1 uses a simple greedy search (`argmax()` of the scores) to always select the "best" sampled word d_k.

The reasoning speaker S1 returns an array of speaker scores, listener scores, and the generated descriptions.

V. Hyperparameter Optimization

In the context of our work, we decided to go beyond replicating the results and perform hyperparameter optimization. Here we use a grid search over different hyperparameters. The parameters are stored at the beginning of the script in a hardcoded variable called "config" in YAML format. YAML is a simple data description language designed for human readability and is commonly used for application configuration.

A. Examining the Hyperparameters

There are several hyperparameters in the model that affect the performance and behaviour of the model. These parameters and their functions are listed below:

- epochs: Number of epochs for which the model is trained.
- batch_size: Number of samples processed in one step for training.
- alternatives: Number of scenes to choose from. Set to the value one by default, indicates how many distractor scenes are used.
- rho: A value between 0 and 1 that indicates how much weight the current and past gradients should be given when calculating the update step size. A higher value means that the current gradient influence is weighted more heavily, while a lower value means that the past gradient influence is weighted more heavily.
- Epsilon: A small positive value used to avoid division by zero when calculating the update step size. It ensures that the denominator of the update rule never becomes zero.
- lr: The learning rate, which determines how much the weights should change at each update. A higher learning rate leads to faster convergence speeds, but also to more unstable training processes.
- clip: Threshold value that specifies when a normalisation of the gradients is required.
- prop_embedding_size: Configured but not used in the project.
- word_embedding_size: Configured but not used in the project.
- hidden_size: Hidden layer size of the model. This refers to the number of neurons in the "horizontal" plane.

As we only wanted to train on the CPU for resource reasons and wanted to keep the number of epochs the same, we decided to limit ourselves to the hyperparameters that we expected to have the greatest impact on learning. We therefore decided to fine tune the hyperparameters "lr", "batch_size" and "hidden_size". The learning rate (lr) has a significant impact on the learning behaviour of the model, as too high or too low a learning rate can lead to unstable or slow learning processes. Similarly, the batch size (batch_size) has a major impact on learning, as using too large a batch size can lead to a decrease in the generalisation ability of the model. Finally, the size of the hidden layer (hidden_size) can also have a large impact on the performance of the model, as smaller hidden layers may not be sufficient to learn complex patterns, while too large a hidden layer may lead to overfitting. We hypothesise that optimizing these hyperparameters can significantly improve the performance of our model.

B. Implementation of the Hyperparameter Optimization

For the hyperparameter optimization, we created a Python script called "hyperparameter_optimization.py". The selected hyperparameters were defined in lists and then iterated through. For each "hyperparameter combination", the configuration variable was updated and the "main_wrapper()" function was called with the updated configuration. The "corpus" and "job" variables were hardcoded with the values "train.base" and "abstract". In addition, a unique "model_name" was passed containing the values of the hyperparameters.

To perform the optimization, a method called "main_wrapper()" was added to "main.py". The script trained the model for each combination of the selected hyperparameters and stored the loss and accuracy of the test/validation dataset of the trained "listener0_model" or "speaker0_model" in a file named model_name in the "performance" folder. If the performance folder did not already exist, it will be created automatically. An empty text file with the model name was created and the loss and accuracy were stored in a new line for each epoch.

As training on the CPU is very slow, we decided to use a virtual machine in the Google Cloud with 4 CPU cores and 8 gigabytes of RAM. To use all the CPU cores, we wrote the hyperparameter optimization script to start a new subprocess on one of the available CPU cores for each model training. Since Apollocaffe does not natively support multiprocessing, we implemented this functionality in our script to speed up training.

C. Results

The baseline model has the following hyperparameters Hidden Size: 100, Batch Size: 100, Learning Rate: 1. Figure 2 shows the loss per epoch for the listener0 model with different hyperparameters. After the 10th epoch, the three best hyperparameter combinations for listener0 are as follows

- Listener Loss: 9.312 with "h150_b50_lr0.5".
- Listener loss: 9.402 for "h150_b50_lr1.0".
- Listener loss: 9.534 for "h100_b50_lr1.0".

From the results it can be concluded that mainly increasing the hidden layer size and reducing the batch size can improve the Loss value after the 10th epoch. However, the difference between the best combinations is small.

Similarly, figure three shows the loss per epoch for the speaker0 model with different hyperparameters. After the 10th epoch, the three best hyperparameter combinations for speaker0 are as follows

- Speaker Loss: 10.724 with "h150_b150_lr1.0".
- Speaker Loss: 10.775 for "h150_b100_lr1.0".
- Speaker loss: 10.788 for "h150_b100_lr1.0".

Again, the difference to the baseline is small. However, in contrast to the listener0 model, a larger batch size seems to be beneficial. Interestingly, in this model the values converge with a similar slope, regardless of the hyperparameters. In summary, it can be said that the models of Andreas and Klein can be slightly improved by performing a hyperparameter optimization.

VI. Migration to PyTorch

Our initial motivation for migrating the code *PyTorch* was our struggle to get *ApolloCaffe* to work. We spent quite some time understanding the code and commenting it while we were experimenting with it which was particularly important to gain insight into how the raw data is processed and prepared for later use in the actual model.

1) Approach:: Because of all the differences we encountered while comparing the descriptions and equations in the paper, we decided to stick for our migration as closely as possible to the paper, that is, literally implement the equations. For all the parts which were only insufficiently detailed, we copied the code from the reference implementation and marked these lines. Our commented results are provided in `manual_implementation_model.py`.

2) Experiences during code migration:: Fully migrating the base models for S0 and L0 to *PyTorch* worked well. However, the training loop was hard as *ApolloCaffe* and PyTorch follow entirely different approaches. For example, in *ApolloCaffe*, some sub-module functions in the forward pass already have dedicated loss layers and a home-brew *Adadelta* optimiser was used. Furthermore, the official *Caffe* [2] documentation lacks an explanation of how the framework manages the flow of gradients and loss computation. For a full migration, thus, quite some trial and error would have been necessary. As a consequence, we only had sufficient time to fully migrate the base models L0 and S0 to *PyTorch*. Due to this lack of time, we could not test our re-implementation but are certain, that it correctly reflects at least the concepts of the base models. Nevertheless, it helped us gain a good understanding in how the internals of these models and the approach of the overall model presented in the paper work.

VII. Conclusion

This project successfully replicated the model presented in Jacob Andreas and Dan Klein's paper "Reasoning about Pragmatics with Neural Listeners and Speakers". We achieved our main objectives, which were to set up a system environment, provide detailed documentation of the implementation, and perform hyperparameter optimization. Although the optional task of migrating the code to PyTorch was not fully implemented, we hope that our work can serve as a solid foundation for future efforts in this direction.

We have provided a thorough explanation of the data formats, data loading and data preparation processes, as well as a detailed description of the main components of the model, such as the Literal Listener L0, the Literal Speaker S0 and the Reasoning Speaker S1. We have also presented the results of our hyperparameter optimization, which revealed potential improvements in the model's performance.

Our project contributes to a better understanding of the methods and practical applications of the model presented in the paper. It also highlights the importance of pragmatics and context in natural language generation, which are essential for improving human-machine interaction. By providing a detailed implementation guide, we hope to make the model more accessible and usable for other researchers and developers, thus encouraging further progress in the field.

References

[1] Andreas, J. (n.d.). Jacobandreas/apollocaffe. GitHub. Retrieved April 7, 2023, from https://github.com/jacobandreas/apollocaffe

[2] Jia, Y., & Shelhamer, E. (n.d.). Installation instructions. Caffe. Retrieved April 7, 2023, from http://caffe.berkeleyvision.org/installation.html

[3] VirtualBox. (n.d.). In Oracle VM VirtualBox. Retrieved April 7, 2023, from https://www.virtualbox.org/

[4] Google. (n.d.). Google Cloud: Cloud Computing Services. Retrieved April 7, 2023, from https://cloud.google.com/?hl=en

[5] Canonical Ltd. (n.d.). Package management. Ubuntu. Retrieved April 7, 2023, from https://ubuntu.com/server/docs/package-management

[6] The Python Packaging Authority. (n.d.). pip: pip install package installer. PyPI. Retrieved April 07, 2023, from https://pypi.org/project/pip/

[7] Zitnick and Devi Parikh. 2013. Bringing semantics into focus using visual abstraction. In Proceedings of the Conference on Computer Vision and Pattern Recognition, (pp. 3009-3016).

[8] Georgia Tech College of Computing. (n.d.). Bringing Semantics Into Focus Using Visual Abstraction Learning the Visual Interpretation of Sentences. Retrieved April 7, 2023, from http://optimus.cc.gatech.edu/clipart/

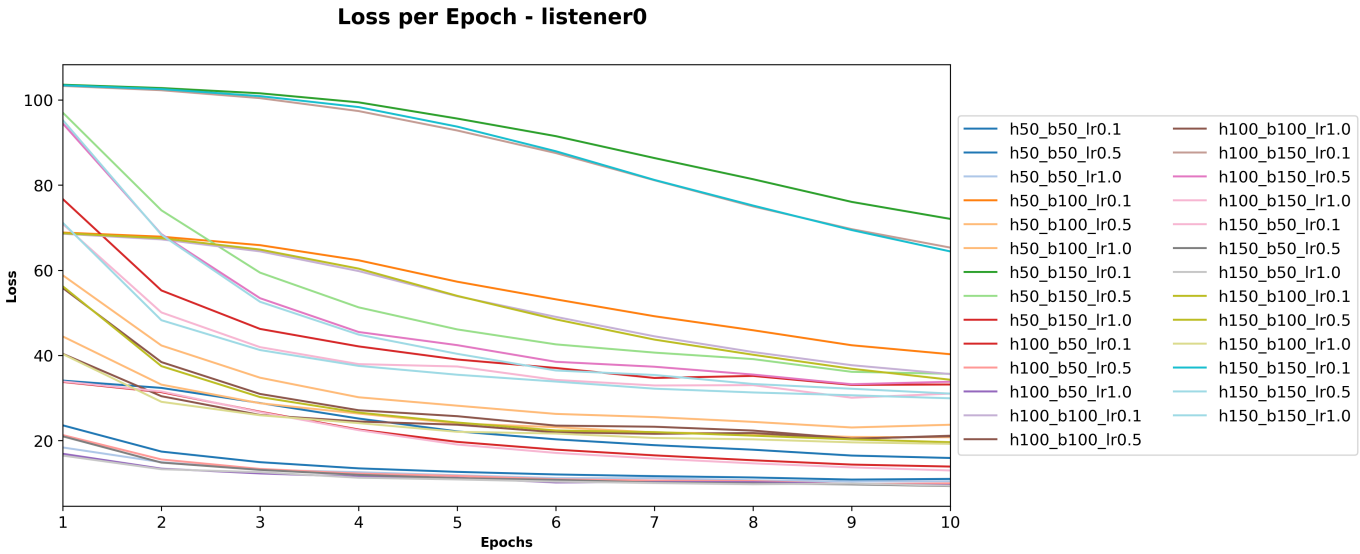

Fig. 2: The figure shows the loss per epoch for the listener0 model using different hyperparameters.

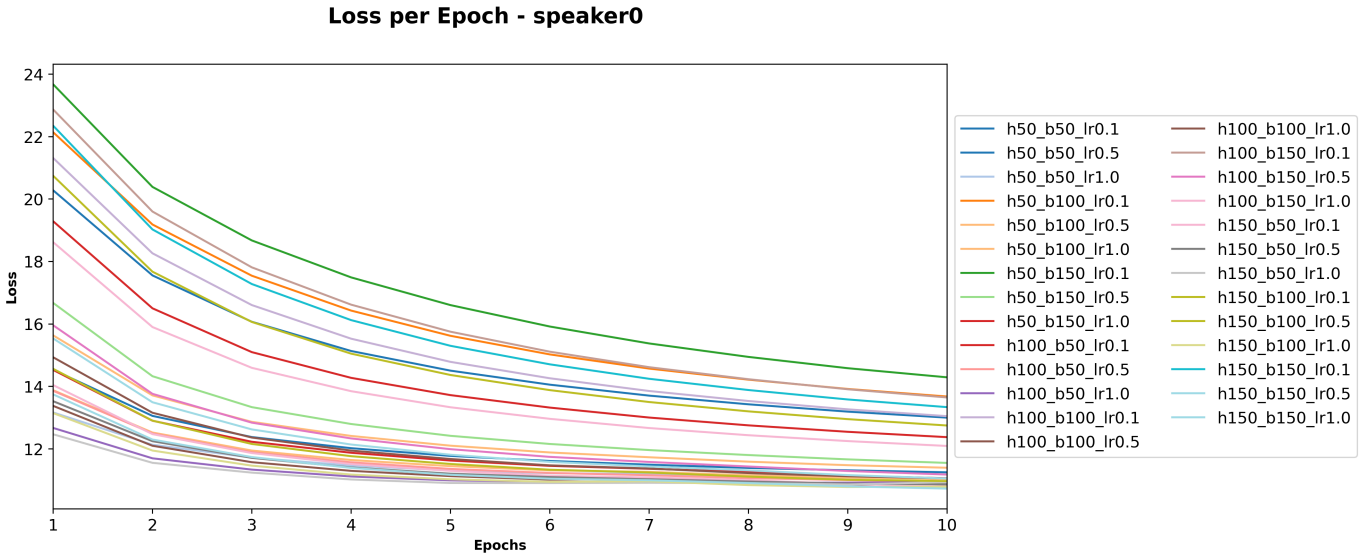

Fig. 3: The figure shows the loss per epoch for the speaker0 model using different hyperparameters.