Jack the Reader – A Machine Reading Framework

Dirk Weissenborn[1], Pasquale Minervini[2], Tim Dettmers[3], Isabelle Augenstein[4], Johannes Welbl[2], Tim Rocktäschel[5], Matko Bošnjak[2], Jeff Mitchell[2], Thomas Demeester[6], Pontus Stenetorp[2], Sebastian Riedel[2]

[1]German Research Center for Artificial Intelligence (DFKI), Germany
[2]University College London, United Kingdom
[3]Università della Svizzera italiana, Switzerland
[4]University of Copenhagen, Denmark
[5]University of Oxford, United Kingdom
[6]Ghent University - imec, Ghent, Belgium

Abstract

Many Machine Reading and Natural Language Understanding tasks require reading supporting text in order to answer questions. For example, in Question Answering, the supporting text can be newswire or Wikipedia articles; in Natural Language Inference, premises can be seen as the supporting text and hypotheses as questions. Providing a set of useful primitives operating in a single framework of related tasks would allow for expressive modelling, and easier model comparison and replication. To that end, we present Jack the Reader (JACK), a framework for Machine Reading that allows for quick model prototyping by component reuse, evaluation of new models on existing datasets as well as integrating new datasets and applying them on a growing set of implemented baseline models. JACK is currently supporting (but not limited to) three tasks: Question Answering, Natural Language Inference, and Link Prediction. It is developed with the aim of increasing research efficiency and code reuse.

1 Introduction

Automated reading and understanding of textual and symbolic input, to a degree that enables question answering, is at the core of Machine Reading (*MR*). A core insight facilitating the development of MR models is that most of these tasks can be cast as an instance of the Question Answering (*QA*) task: an input can be cast in terms of *question*, *support documents* and *answer candidates*, and an output in terms of *answers*. For instance, in case of Natural Language Inference (*NLI*), we can view the hypothesis as a multiple choice question about the underlying premise (support) with predefined set of specific answer candidates (entailment, contradiction, neutral). Link Prediction (*LP*) – a task which requires predicting the truth value about facts represented as *(subject, predicate, object)*-triples – can be conceived of as an instance of QA (see Section 4 for more details). By unifying these tasks into a single framework, we can facilitate the design and construction of multi-component MR pipelines.

There are many successful frameworks such as STANFORD CORENLP (Manning et al., 2014), NLTK (Bird et al., 2009), and SPACY[1] for NLP, LUCENE[2] and SOLR[3] for Information Retrieval, and SCIKIT-LEARN[4], PYTORCH[5] and TENSORFLOW (Abadi et al., 2015) for general Machine Learning (*ML*) with a special focus on Deep Learning (*DL*), among others. All of these frameworks touch upon several aspects of Machine Reading, but none of them offers dedicated support for modern MR pipelines. Pre-processing and transforming MR datasets into a format that is usable by a MR model as well as implementing common architecture building blocks all require substantial effort which is not specifically handled by any of the aforementioned solutions. This is due to the fact that they serve a different, typically much broader purpose.

In this paper, we introduce Jack the Reader (JACK), a reusable framework for MR. It allows for the easy integration of novel tasks and datasets by exposing a set of high-level primitives and a common data format. For supported tasks it is straight-forward to develop new models without worrying about the cumbersome implementation

[1] https://spacy.io
[2] https://lucene.apache.org
[3] http://lucene.apache.org/solr/
[4] http://scikit-learn.org
[5] http://pytorch.org/

of training, evaluation, pre- and post-processing routines. Declarative model definitions make the development of QA and NLI models using common building blocks effortless. JACK covers a large variety of datasets, implementations and pre-trained models on three distinct MR tasks and supports two ML backends, namely PYTORCH and TENSORFLOW. Furthermore, it is easy to train, deploy, and interact with MR models, which we refer to as *readers*.

2 Related Work

Machine Reading requires a tight integration of Natural Language Processing and Machine Learning models. General NLP frameworks include CORENLP (Manning et al., 2014), NLTK (Bird et al., 2009), OPENNLP[6] and SPACY. All these frameworks offer pre-built models for standard NLP preprocessing tasks, such as tokenisation, sentence splitting, named entity recognition and parsing.

GATE (Cunningham et al., 2002) and UIMA (Ferrucci and Lally, 2004) are toolkits that allow quick assembly of baseline NLP pipelines, and visualisation and annotation via a Graphical User Interface. GATE can utilise NLTK and CORENLP models and additionally enable development of rule-based methods using a dedicated pattern language. UIMA offers a text analysis pipeline which, unlike GATE, also includes retrieving information, but does not offer its own rule-based language. It is further worth mentioning the Information Retrieval frameworks APACHE LUCENE and APACHE SOLR which can be used for building simple, keyword-based question answering systems, but offer no ML support.

Multiple general machine learning frameworks, such as SCIKIT-LEARN (Pedregosa et al., 2011), PYTORCH, THEANO (Theano Development Team, 2016) and TENSORFLOW (Abadi et al., 2015), among others, enable quick prototyping and deployment of ML models. However, unlike JACK, they do not offer a simple framework for defining and evaluating MR models.

The framework closest in objectives to JACK is ALLENNLP (Gardner et al., 2017), which is a research-focused open-source NLP library built on PYTORCH. It provides the basic low-level components common to many systems in addition to pre-assembled models for standard NLP tasks, such as coreference resolution, constituency parsing, named entity recognition, question answering and textual entailment. In comparison with ALLENNLP, JACK supports both TENSORFLOW and PYTORCH. Furthermore, JACK can also learn from Knowledge Graphs (discussed in Section 4), while ALLENNLP focuses on textual inputs. Finally, JACK is structured following a modular architecture, composed by input-, model-, and output modules, facilitating code reuse and the inclusion and prototyping of new methods.

[6] https://opennlp.apache.org

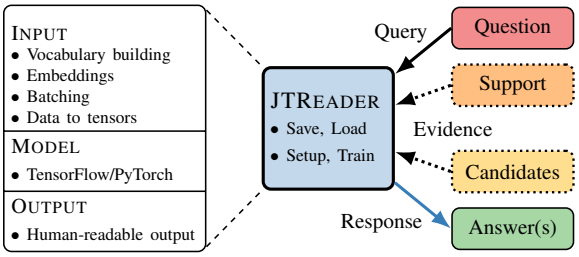

Figure 1: Our core abstraction, the JTREADER. On the left, the responsibilities covered by the INPUT, MODEL and OUTPUT modules that compose a JTREADER instance. On the right, the data format that is used to interact with a JTREADER (dotted lines indicate that the component is optional).

3 Overview

In Figure 1 we give a high-level overview of our core abstraction, the JTREADER. It is a task-agnostic wrapper around three typically task-dependent modules, namely the *input*, *model* and *output* modules. Besides serving as a container for modules, a JTREADER provides convenience functionality for interaction, training and serialisation. The underlying modularity is therefore well hidden from the user which facilitates the application of trained models.

3.1 Modules and Their Usage

Our abstract modules have the following high-level responsibilities:

- INPUT MODULES: Pre-processing that transforms a text-based input to tensors.
- MODEL MODULES: Implementation of the actual end-to-end MR model.
- OUTPUT MODULES: Converting predictions into human readable answers.

The main design for building models in JACK revolves around functional interfaces between the three main modules: the input-, model-, and output module. Each module can be viewed as a thin wrapper around a (set of) function(s) that additionally provides explicit signatures in the form of *tensor ports* which can be understood as named placeholders for tensors.

The use of explicit signatures helps validate whether modules are correctly implemented and invoked, and to ensure correct behaviour as well as compatibility between modules. Finally, by implementing modules as classes and their interaction via a simple functional interface, JACK allows for the exploitation of benefits stemming from the use of object oriented programming, while retaining the flexibility offered by the functional programming paradigm when combining modules.

Given a list of training instances, corresponding to question-answer pairs, a *input module* is responsible for converting such instances into tensors. Each produced tensor is associated with a pre-defined *tensor port* – a named placeholder for a tensor – which can in turn be used in later modules to retrieve the actual tensor. This step typically involves some shallow forms of linguistic pre-processing such as tokenisation, building vocabularies, etc. The *model module* runs the end-to-end MR model on the now tensorised input and computes a new mapping of output tensor ports to newly computed tensors. Finally, the joint tensor mappings of the input- and model module serve as input to the *output module* which produces a human-readable answer. More in-depth documentation can be found on the project website.

3.2 Distinguishing Features

Module Reusability. Our shallow modularisation of readers into input-, model- and output modules has the advantage that they can be reused easily. Most of nowadays state-of-the-art MR models require the exact same kind of input pre-processing and produce output of the same form. Therefore, existing input- and output modules that are responsible for pre- and post-processing can be reused in most cases, which enables researchers to focus on prototyping and implementing new models. Although we acknowledge that most of the pre-processing can easily be performed by third-party libraries such as CORENLP, NLTK or SPACY, we argue that additional functionality, such as building and controlling vocabularies, padding, batching, etc., and connecting the pre-processed output with the actual model implementation pose time intensive implementation challenges. These can be avoided when working with one of our currently supported tasks – Question Answering, Natural Language Inference, or Link Prediction in Knowledge Graphs. Note that modules are typically task specific and not shared directly between tasks. However, utilities like the pre-processing functions mentioned above and model building blocks can readily be reused even between tasks.

Supported ML Backends. By decoupling modelling from pre- and post-processing we can easily switch between backends for model implementations. At the time of writing, JACK offers support for both TENSORFLOW and PYTORCH. This allows practitioners to use their preferred library for implementing new MR models and allows for the integration of more back-ends in the future.

Declarative Model Definition. Implementing different kinds of MR models can be repetitive, tedious, and error-prone. Most neural architectures are built using a finite set of basic *building blocks* for encoding sequences, and realising interaction between sequences (e.g. via attention mechanisms). For such a reason, JACK allows to describe these models at a high level, as a composition of simpler building blocks [7], leaving concrete implementation details to the framework.

The advantage of using such an approach is that is very easy to change, adapt or even create new models without knowing any implementation specifics of JACK or its underlying frameworks, such as TENSORFLOW and PYTORCH. This solution also offers another important advantage: it allows for easy experimentation of automated architecture search and optimisation (AutoML). JACK already enables the definition of new models purely within configuration files without writing any source code. These are interpreted by JACK and support a (growing) set of pre-defined building blocks. In fact, many models for different tasks in JACK are realised by high-level architecture descriptions. An example of an high-level architecture definition in JACK is available in Appendix A.

[7] For instance, see https://github.com/uclmr/jack/blob/master/conf/nli/esim.yaml

Dataset Coverage. JACK allows parsing a large number of datasets for QA, NLI, and Link Prediction. The supported QA datasets include SQuAD (Rajpurkar et al., 2016), TriviaQA (Joshi et al., 2017), NewsQA (Trischler et al., 2017), and QAngaroo (Welbl et al., 2017). The supported NLI datasets include SNLI (Bowman et al., 2015), and MultiNLI (Williams et al., 2018). The supported Link Prediction datasets include WN18 (Bordes et al., 2013), WN18RR (Dettmers et al., 2018), and FB15k-237 (Toutanova and Chen, 2015).

Pre-trained Models. JACK offers several pre-trained models. For QA, these include FastQA, BiDAF, and JackQA trained on SQuAD and TriviaQA. For NLI, these include DAM and ESIM trained on SNLI and MultiNLI. For LP, these include DistMult and ComplEx trained on WN18, WN18RR and FB15k-237.

4 Supported MR Tasks

Most end-user MR tasks can be cast as an instance of question answering. The input to a typical question answering setting consists of a *question*, *supporting texts* and *answers* during training. In the following we show how JACK is used to model our currently supported MR tasks.

Ready to use implementations for these tasks exist which allows for rapid prototyping. Researchers interested in developing new models can define their architecture in TENSORFLOW or PYTORCH, and reuse existing of input- and output modules. New datasets can be tested quickly on a set of implemented baseline models after converting them to one of our supported formats.

Extractive Question Answering. JACK supports the task of *Extractive Question Answering* (EQA), which requires a model to extract an answer for a question in the form of an answer span comprising a document id, token start and -end from a given set of supporting documents. This task is a natural fit for our internal data format, and is thus very easy to represent with JACK.

Natural Language Inference. Another popular MR task is *Natural Language Inference*, also known as Recognising Textual Entailment (RTE). The task is to predict whether a *hypothesis* is entailed by, contradicted by, or neutral with respect to a given *premise*. In JACK, NLI is viewed as an instance of multiple-choice Question Answering problem, by casting the hypothesis as the question, and the premise as the support. The answer candidates to this question are the three possible outcomes or classes – namely *entails*, *contradicts* or *neutral*.

Link Prediction. A Knowledge Graph is a set of (s, p, o) triples, where s, o denote the *subject* and *object* of the triple, and p denotes its *predicate*: each (s, p, o) triple denotes a fact, represented as a relationship of type p between entities s and o, such as: (LONDON, CAPITALOF, UK). Real-world Knowledge Graphs, such as Freebase (Bollacker et al., 2007), are largely incomplete: the *Link Prediction* task consists in identifying missing (s, p, o) triples that are likely to encode true facts (Nickel et al., 2016).

JACK also supports Link Prediction, because existing LP models can be cast as multiple-choice Question Answering models, where the question is composed of three words – a subject s, a predicate p, and an object o. The answer candidates to these questions are *true* and *false*.

In its original formulation of the Link Prediction task, the support is left empty. However, JACK facilitates enriching the questions with additional support – consisting, for instance, of the neighbourhood of the entities involved in the question, or sentences from a text corpus that include the entities appearing in the triple in question. Such a setup can be interpreted as an instance of NLI, and existing models not originally designed for solving Link Prediction problems can be trained effortlessly.

5 Experiments

Experimental setup and results for different models on the three above-mentioned MR tasks are reported in this section. Note that our re-implementations or training configurations may not be entirely faithful.We performed slight modifications to original setups where we found this to perform better in our experiments, as indicated in the respective task subsections. However, our results still vary from the reported ones, which we believe is due to the extensive hyper-parameter engineering that went into the original settings, which we did not perform. For each experiment, a ready to use training configuration as well as pre-trained models are part of JACK.

Model	Original F1	JACK F1	Speed	#Params
BiDAF	77.3	77.8	1.0x	2.02M
FastQA	76.3	77.4	2.2x	0.95M
JackQA	–	79.6	2.0x	1.18M

Table 1: Metrics on the SQuAD development set comparing F1 metric from the original implementation to that of JACK, number of parameters, and relative speed of the models.

Model	Original	JACK
cBiLSTM (Rocktäschel et al., 2016)	–	82.0
DAM (Parikh et al., 2016)	86.6	84.6
ESIM (Chen et al., 2017)	88.0	87.2

Table 2: Accuracy on the SNLI test set achieved by cBiLSTM, DAM, and ESIM.

Question Answering. For the Question Answering (QA) experiments we report results for our implementations of FastQA (Weissenborn et al., 2017), BiDAF (Seo et al., 2016) and, in addition, our own JackQA implementations. With JackQA we aim to provide a fast and accurate QA model. Both BiDAF and JackQA are realised using high-level architecture descriptions, that is, their architectures are purely defined within their respective configuration files. Results of our models on the SQuAD (Rajpurkar et al., 2016) development set along with additional run-time and parameter metrics are presented in Table 1. Apart from SQuAD, JACK supports the more recent NewsQA (Trischler et al., 2017) and TriviaQA (Joshi et al., 2017) datasets too.

Natural Language Inference. For NLI, we report results for our implementations of conditional BiLSTMs (cBiLSTM) (Rocktäschel et al., 2016), the bidirectional version of conditional LSTMs (Augenstein et al., 2016), the Decomposable Attention Model (DAM, Parikh et al., 2016) and Enhanced LSTM (ESIM, Chen et al., 2017). ESIM was entirely implemented as a *modular* NLI model, i.e. its architecture was purely defined in a configuration file – see Appendix A for more details. Our models or training configurations contain slight modifications from the original which we found to perform better than the original setup. Our results are slightly differ from those reported, since we did not always perform an exhaustive hyper-parameter search.

Dataset	Model	MRR	Hits@3	Hits@10
WN18	DistMult	0.822	0.914	0.936
	ComplEx	0.941	0.936	0.947
WN18RR	DistMult	0.430	0.443	0.490
	ComplEx	0.440	0.461	0.510
FB15k-237	DistMult	0.241	0.263	0.419
	ComplEx	0.247	0.275	0.428

Table 3: Link Prediction results, measured using the Mean Reciprocal Rank (MRR) and Hits@10, for DistMult (Yang et al., 2015), and ComplEx (Trouillon et al., 2016).

Link Prediction. For Link Prediction in Knowledge Graphs, we report results for our implementations of DistMult (Yang et al., 2015) and ComplEx (Trouillon et al., 2016) on various datasets. Results are otlined in Table 3.

6 Demo

We created three tutorial *Jupyter* notebooks at `https://github.com/uclmr/jack/tree/master/notebooks` to demo JACK's use cases. The quick start notebook shows how to quickly set up, load and run the existing systems for QA and NLI. The model training notebook demonstrates training, testing, evaluating and saving QA and NLI models programmatically. However, normally the user will simply use the provided training script from command line. The model implementation notebook delves deeper into implementing new models from scratch by writing all modules for a custom model.

7 Conclusion

We presented Jack the Reader (JACK), a shared framework for Machine Reading tasks that will allow component reuse and easy model transfer across both datasets and domains.

JACK is a new unified Machine Reading framework applicable to a range of tasks, developed with the aim of increasing researcher efficiency and code reuse. We demonstrate the flexibility of our framework in terms of three tasks: Question Answering, Natural Language Inference, and Link Prediction in Knowledge Graphs. With further model additions and wider user adoption, JACK will support faster and reproducible Machine Reading research, enabling a building-block approach to model design and development.

References

Martín Abadi et al. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. https://www.tensorflow.org/.

Isabelle Augenstein, Tim Rocktäschel, Andreas Vlachos, and Kalina Bontcheva. 2016. Stance detection with bidirectional conditional encoding. In *Proceedings or EMNLP*. pages 876–885.

Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.".

Kurt D. Bollacker, Robert P. Cook, and Patrick Tufts. 2007. Freebase: A shared database of structured general human knowledge. In *Proceedings of AAAI*. pages 1962–1963.

Antoine Bordes, Nicolas Usunier, Alberto García-Durán, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. In *Proceedings of NIPS*. pages 2787–2795.

Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. 2015. A large annotated corpus for learning natural language inference. In *Proceedings of EMNLP*. pages 632–642.

Qian Chen, Xiaodan Zhu, Zhen-Hua Ling, Si Wei, Hui Jiang, and Diana Inkpen. 2017. Enhanced LSTM for natural language inference. In *Proceedings of ACL*. pages 1657–1668.

Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. 2002. GATE: an Architecture for Development of Robust HLT applications. In *Proceedings of ACL*.

Tim Dettmers, Minervini Pasquale, Stenetorp Pontus, and Sebastian Riedel. 2018. Convolutional 2d knowledge graph embeddings. In *Proceedings of AAAI*.

David Ferrucci and Adam Lally. 2004. Uima: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering* 10(3-4):327–348.

Matt Gardner et al. 2017. AllenNLP: A Deep Semantic Natural Language Processing Platform. White paper.

Mandar Joshi, Eunsol Choi, Daniel S. Weld, and Luke Zettlemoyer. 2017. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of ACL*. pages 1601–1611.

Christopher Manning et al. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of ACL: System Demonstrations*. pages 55–60.

Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. 2016. A review of relational machine learning for knowledge graphs. In *Proceedings of IEEE*. volume 104, pages 11–33.

Ankur P. Parikh, Oscar Täckström, Dipanjan Das, and Jakob Uszkoreit. 2016. A decomposable attention model for natural language inference. In *Proceedings of EMNLP*. pages 2249–2255.

Fabian Pedregosa et al. 2011. Scikit-learn: Machine learning in Python. *JMLR* 12(Oct):2825–2830.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. In *Proceedings of EMNLP*. pages 2383–2392.

Tim Rocktäschel, Edward Grefenstette, Karl Moritz Hermann, Tomas Kocisky, and Phil Blunsom. 2016. Reasoning about Entailment with Neural Attention. In *Proceedings of ICLR*.

Min Joon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. 2016. Bidirectional attention flow for machine comprehension. In *Proceedings of ICLR*.

Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688.

Kristina Toutanova and Danqi Chen. 2015. Observed versus latent features for knowledge base and text inference. *CVSC workshop, ACL* pages 57–66.

Adam Trischler et al. 2017. NewsQA: A machine comprehension dataset. *Rep4NLP workshop, ACL* .

Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. 2016. Complex embeddings for simple link prediction. In *Proceedings of ICML*. pages 2071–2080.

Dirk Weissenborn, Georg Wiese, and Laura Seiffe. 2017. Making neural QA as simple as possible but not simpler. In *Proceedings of CoNLL*. pages 271–280.

Johannes Welbl, Pontus Stenetorp, and Sebastian Riedel. 2017. Constructing datasets for multi-hop reading comprehension across documents. *CoRR* abs/1710.06481.

Adina Williams, Nikita Nangia, and Samuel R. Bowman. 2018. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of NAACL*.

Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. 2015. Embedding Entities and Relations for Learning and Inference in Knowledge Bases. In *Proceedings of ICLR*.

A High-level Architecture Design in Jack

We provide support for the modular composition of QA and NLI architectures within configuration files, so there is no need to touch code at all. An example configuration snippet that shows the definition of our JackQA model is presented in Listing 1. We start with a set of pre-defined start keys ('question', 'char_question', 'support' and 'char_support' for QA). These refer to their respective embedded sequences. The architecture is built by a sequence of modular neural building blocks, in short *modules*. Each module receives an input (a tensor or list of tensors) as determined by the given input keys and produces an output which can be referred to in subsequent modules using the provided output key. In case no output key is given, it defaults to the given input key or the first of a list of given input keys. More detailed information can be found in our online documentation.

```
repr_dim: 100

model:
  encoder_layer:
  # Shared encoding
  # Support
  - input: ['support', 'char_support']
    output: 'support'
    module: 'concat'
  - input: 'support'
    name: 'embedding_highway'
    module: 'highway'
  - input: 'support'
    output: 'emb_support'
    name: 'embedding_projection'
    module: 'dense'
    activation: 'tanh'
    dropout: True
  - input: 'emb_support'
    output: 'support'
    module: 'conv_glu'
    conv_width: 5
    num_layers: 2
    name: 'contextual_encoding'
    dropout: True
  - input: ['emb_support', 'support']
    output: 'enc_support'
    module: 'concat'

  # Question
  - input: ['question', 'char_question']
    output: 'question'
    module: 'concat'
  - input: 'question'
    name: 'embedding_highway'
    module: 'highway'
  - input: 'question'
    output: 'emb_question'
    name: 'embedding_projection'
    module: 'dense'
    activation: 'tanh'
    dropout: True
  - input: 'emb_question'
    output: 'question'
    module: 'conv_glu'
    conv_width: 5
    num_layers: 2
    name: 'contextual_encoding'
    dropout: True
  - input: ['emb_question', 'question']
    output: 'enc_question'
    module: 'concat'

  # Attention
  - input: 'enc_support'
    dependent: 'enc_question'
    output: 'support'
    module: 'attention_matching'
    attn_type: 'diagonal_bilinear'
    with_sentinel: True
    scaled: True
  - input: 'support'
    output: 'support_self'
    module: 'dense'
    activation: 'tanh'

  # Self Attention
  - input: 'support_self'
    module: 'self_attn'
    attn_type: 'diagonal_bilinear'
    scaled: True
    with_sentinel: True
    num_attn_heads: 1

  # Concatenate outputs
  - input: ['support', 'support_self']
    output: 'support'
    module: 'concat'
  - input: 'support'
    module: 'dense'
    activation: 'relu'
    dropout: True

  # BiLSTM, the only application
  - input: 'support'
    module: 'lstm'
    with_projection: True
    activation: 'tanh'
    dropout: True

  - input: 'support'
    module: 'conv_glu'
    conv_width: 5
    num_layers: 1
    residual: True

  answer_layer:
    support: 'support'
    question: 'enc_question'
    module: 'bilinear'
```

Listing 1: Sample YAML architecture description for our JackQA model.