
Pixie: A Social Chatbot

**Oluwatosin Adewale, Alex Beatson, Davit Buniatyan, Jason Ge,
Mikhail Khodak, Holden Lee, Niranjani Prasad, Nikunj Saunshi,
Ari Seff, Karan Singh, Daniel Suo, Cyril Zhang, Sanjeev Arora**

Department of Computer Science, Princeton University
35 Olden Street, Princeton, NJ

{oadewale, abeatson, davit, jiange, mkhodak, holdenl, np6,
nsaunshi, aseff, karans, daniel.suo, cyril.zhang}@princeton.edu,
arora@cs.princeton.edu

Abstract

We present Pixie, a socialbot submitted to the 2017 Alexa Prize, developed with the objective of holding engaging casual conversations on open-ended topics. We discuss the modular, heterogeneous architecture of the Pixie system, and highlight the successes and shortcomings of our primarily template-based approach. We reflect on our findings about alternative paradigms used in the prototyping phase; in particular, we discuss the challenges of bridging recent breakthroughs in deep learning with the frontier of machine social dialogue.

1 Introduction

Conversation serves multiple purposes: not just as a quick way to retrieve information or ask for something done, but also an opportunity to encounter new and interesting thoughts, or perhaps to receive some comic relief after a long day of work. However, despite dramatic advances in artificial intelligence in recent years, machines still struggle to interpret and generate unrestricted natural language, with state-of-the-art chatbots only able to service short, highly-structured requests. While this use case is important and a subject of intense research, such chatbots fail to engross users in longer, more varied conversations. Without a programmatic understanding of the social skills required to engage them, users will and do relegate chatbots to the brief interactions we optimize them for.

Much exciting work has emerged in recent years in the related problems of either *query answering*, or in dialogue systems designed to complete tasks in a specific domain. For example, [VL15] looks to generate single-step responses to general user inputs using sequence-to-sequence models, while [BW16] focuses at goal-oriented dialogue systems that are able to retain context and multi-step information for very narrow domain tasks using memory networks. In the socialbot setting, which calls for both context and generalizability, progress is far more limited. [SSB⁺16] for example, looks to develop open-ended dialogue system trained on triples (sequences of three utterances) to improve context, using generative hierarchical recurrent encoder-decoder networks ([SSL⁺17] extends this by introducing hierarchical dependencies at variable time scales). [LMR⁺16] looks to go beyond shortsighted sequence generation in open domains by leveraging deep reinforcement learning methods in choosing the next utterance, and using metrics such as informativity, coherence, and ease of answering to do this. These methods tend to be dependent on huge data corpora, however, and extension to spoken dialogue (as with Alexa) from text-based systems in itself presents challenges.

In this work, we aim to develop a socialbot capable of maintaining a coherent conversation on popular topics or news events in the realms of technology, politics, sports, fashion or entertainment, with the ultimate goal of engaging the user for up to 20 minutes, premised on the vision of the Alexa Prize competition. Towards this end, we look to equip our conversational agent with a multi-paradigm mechanism for generating responses, combining the power of semantic knowledge bases,

the idiomatic general-purpose conversation flow exhibited by end-to-end neural networks, and the utility of rule-based systems. Each of these modules is provided with the raw input augmented with conversation history. A final component determines which of the strategies will produce the output most likely to continue a meaningful conversation.

The rest of the paper is organized as follows: Section 2 outlines the design choices made in terms of cloud infrastructure for use in production. In Sections 3-5, we describe the overall system architecture, and the different methodological elements explored in building our socialbot (including both deep learning and rule based methods). The evaluation phase for the socialbot developed is detailed in Section 6, and finally, Section 7 presents our conclusions from this competition and directions for future work.

2 Infrastructure

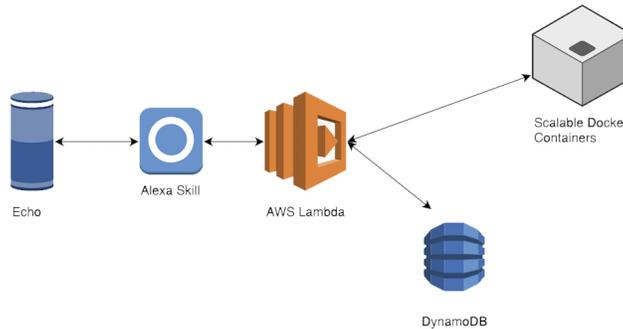


Figure 1: Infrastructure of the system

In this chapter, we will briefly describe all technicalities used for developing and running the production of our social bot. This will include cloud infrastructure, environments, packages and tools. When the request of the user is received from an Amazon Echo device, it lands on our primary server, running on the AWS Lambda service. The *Flask-ask* module handles the request through the Flask microservice architecture [fla, Ron10].

Despite the availability of the intent scheme by the Alexa API, we decided to avoid any outside templating machine and use our own, as described. Hence, we only need to specify that we need to capture the *'AllIntent'* scheme to receive the full text body sent by the user. After receiving the input, the *Message* object is constructed and passed to the bot module, which processes this input and returns a response, including the reprompt. Alexa API provides a specification for sending reprompts that is activated if the user does not respond to an answer from the skill. Besides handling Alexa specific calls, we also utilize the dual REST API for testing and deploying our bot to other platforms. For testing purposes we provided bot access through our website (<https://pixie.is>).

The language chosen for the socialbot was Python 2, for its compatibility with microframeworks such as Flask, modern deep learning frameworks such as Tensorflow, as well as a wide range of available toolkits including StanfordNLP, AIML and Profanity [MSB⁺14]. Using the *boto3* module, we constructed a stateless server by storing session information in DynamoDB [Siv12]. Our database consisted of a single table of entries, in which each entry is a *Message* object that holds *sessionID*, *userID*, *timestamp*, and information provided by the socialbot components. In addition, each component can also store its own memory as a dictionary object distributed across entries. This Memory component is described in further detail in Section 3.1. To ensure robustness of querying the DB, we added a secondary table that with *userID* as the primary key to quickly retrieve the history per user. The secondary table also automatically scales based on number of requests.

The AWS Lambda service is well-suited to fast, stateless retrieval-based tasks; however, for heavy computation such as running advanced deep learning models or a templating engine that needs to compile many thousands of templates during the initialization stage, we preferred to run scalable Docker instances powered with Nginx [Ree08]. When requests are landed, we call the component by using the REST API from the Lambda service to the component that resides in one of those

instances. In other words, the Lambda server becomes the manager of information flow by calling all components that do the computation outside this service. Such separation of responsibility allows us to concentrate on effectively handling requests on the Lambda service and efficient computation on Docker instances using stateless architecture by storing session information in the database. For development purposes we have a second clone of the infrastructure described above. Once testing is done within the ‘staging’ environment, it is merged with the ‘production’ system.

3 Architecture

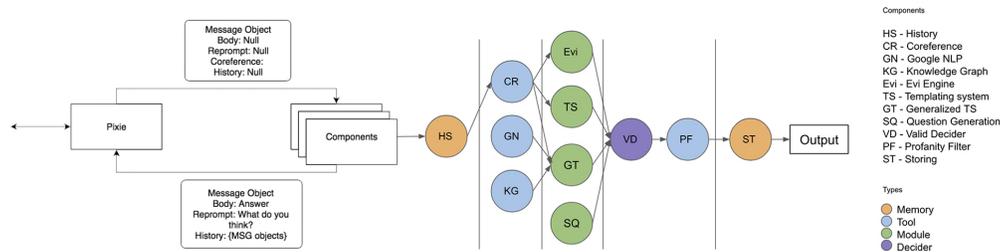


Figure 2: Information flow through the directed graph of components

Pixie is a computational graph system comprising multiple components. Upon receipt of the user input from Flask-ask (described in Section 1), the directed graph of components is initiated. Evaluation and choice of components makes up the core procedure for the *Pixie* socialbot. Each component takes as an input request (*Message* object) and generates a new *Message* object that will then be incorporated into the main request object. After the execution of a given component, other components in later stages have access to the previous information. If a component fails to return any output, or raises an issue, it is excluded from the next stage. All components should check required dependencies before processing the *Message* object. We will divide our components into four classifications: Memory, Modules, Tools, and Decider. We add these labels to distinguish between their roles; however, in terms of abstraction, the components are all identical units similar to tensor operations in TensorFlow.

Dividing the system into components helps us to add modularity, flexibility and separation of responsibilities. In other words, we can easily turn off a single component without affecting the whole system and quickly do A/B testing. The components are divided into stages based on their dependencies and executed in parallel. Each component is responsible for taking care of its dependencies and resolving issues if one of its inputs are not available. Overall, given that each component is proof-checked and can behave in ‘emergency’ situations, the bot is reasonably failure-proof. We will describe the classification of components and dive into each component we developed over the course of the Alexa Prize competition.

3.1 Memory

Memory components include all tasks in which the bot needs to connect to the DynamoDB database, such as getting the conversation history or storing the session. For example, one Memory component is the ‘History’ component, which in the initial stage, queries the DB for the conversation history of the given user. The output of this component is attached to the *Message* object so that further components can access it. The ‘Storing’ component takes the *Message* object at last stage of computation and saves the current output of the bot into the DB. It also stores outputs of each component. Hence, components become error proof and isolated from other component information flow. ‘History’ and ‘Storing’ components allow for stateless execution of sessions keeping user context and the ability to scale on any number of AWS Lambda calls. We also added a third component that helps to store arbitrary information in dictionary format. For example, if the user mentioned his favorite movie, this component can store it as context. During the next session all components will have access to this contextual information without needing to traverse the whole history. This component is useful for storing user interests.

3.2 Modules

Components of the ‘Module’ type provide an individual valid candidate response for the final output. They are executed at the final stage of computation by combining outputs of other components and completing final computations to give the candidate response. In addition to this output response, they also provide a confidence score (how confident they are in their answer). The modules we developed include a templating engine, retrieval-based model, question answering engine (Evi), etc. [evi]. For reprompt generation, we have a Question Generation module that is slightly different to other modules, as it has a different output channel.

3.3 Decider

Once all modules have computed their candidate outputs, the ‘Decider’ component takes all their outputs, as well as the user’s request, as inputs and decides which one is the most contextually equivalent. For now, this module relies on the confidence score provided by each Module component. We discuss later the possibility of extending this module to pick more appropriate responses. Another extension is to combine separate inputs and output the combination. This component type can also switch context by switching Module responses. For example, if the template machine is speaking in the ‘movies’ context, and the retrieval-based machine finds a better context, then the decider component can alternate or switch between these contexts.

3.4 Tools

Due to the need for much preprocessing and postprocessing, we have defined a new component type, tools. Tools are components that analyze and modify the input to make modules wiser. For example, the coreference component takes the input and replaces the pronouns with objects based on the history provided by Memory component. The semantic analyzer takes the input and identifies subject of the input through StanfordNLP or GoogleNLP [MSB⁺14]. We have another component which filters profanity. Tools usually handle single task through connections with external servers.

4 Deep Learning Methods

4.1 Generative Models

Generative models comprise the class of systems that take context as input and directly generate responses. One of the most popular approaches along this line of research are the sequence-to-sequence neural network models [SVL14, LJO16, SSN12, SM13] which used recurrent neural networks such as Long-Short-Term-Memory (LSTM) networks to capture the long range dependencies in the conversation word sequence.

4.2 Retrieval Models

Generating sentences directly from neural network models lacks the robustness that is required for a real world product. A neural network model with large expressive power can capture rich semantic contexts and help the bot improvise in very flexible ways. However, as we don’t have effective regularization methods for generative language modeling, grammatical errors can be made and logical flow of the conversation can be lost.

To strike a balance between expressiveness and robustness, we adopted the retrieval based framework. More specifically, we train a neural network model that takes a {conversation context, response} pair tuple as an input, and computes the matching score of this pair. With the trained network, given any conversation context, a list of candidate follow-ups is first generated by conservative and robust methods. Matching scores between the context and responses are computed and a response with the highest matching score is selected from the list and serves as the final response to the conversation context. The final matching score is normalized to be the confidence score used by the Decider component down the road.

We used the standard LSTM module [SVL14, HS97] to encode the context and the response sentence, where the context is a sequence of words from the beginning of the conversation. If we encode the words as d -dimensional word vectors, any sentence can be represented as a sequence $\{x_t\}_{t=1}^T$ with

$x_t \in \mathbb{R}^d$. The LSTM module describes a recurrent neural network with hidden cells $h_t \in \mathbb{R}^{d_1}$ and memory cells $c_t \in \mathbb{R}^{d_2}$. As new input x_t arrives at time t , the input gate i_t , forget gate f_t and output gate o_t are computed by

$$\begin{aligned} i_t &= \sigma(W_i \cdot [c_{t-1}, h_{t-1}, x_t] + b_i), \\ f_t &= \sigma(W_f \cdot [c_{t-1}, h_{t-1}, x_t] + b_f), \\ o_t &= \sigma(W_o \cdot [c_{t-1}, h_{t-1}, x_t] + b_o) \end{aligned}$$

respectively, where σ is an activation function, W_i, W_f, W_o are weight matrices, and b_i, b_f, b_o are bias vectors. The input vector x_t is mixed with h_{t-1} to generate \tilde{x}_t , i.e:

$$\tilde{x}_t = \tanh(W_x \cdot [h_{t-1}, x_t] + b_x)$$

The memory cell are updated by the mixed input vector \tilde{x}_t and the forget gate f_t by $c_t = f_t * c_{t-1} + (1 - f_t) * \tilde{x}_t$. The new hidden cell is derived from the memory cell c_t and the output gate o_t by the equation $h_t = o_t * \tanh(c_t)$.

We have two separate LSTM modules for the context sequence $\{x_t^c\}_{t=1}^{T_0}$ and the response sequence $\{x_t^r\}_{t=1}^{T_1}$. Denote $\{h_t^c\}_{t=1}^{T_0}$ as the hidden cell sequence for context sequence and $\{h_t^r\}_{t=1}^{T_1}$ as the hidden cell sequence for response sequence. The matching score is calculated by the cosine distance:

$$MS = \frac{\langle h_{T_0}^c, h_{T_1}^r \rangle}{\|h_{T_0}^c\| \cdot \|h_{T_1}^r\|}$$

Training data consists of {context, response} pairs and their matching scores. For each context, we have a real response with matching score 1 and K random responses with matching score 0. Real {context, response} pairs are mined from the online discussion forum Reddit [red] and the Cornell movie dialogue corpus [DNML11]. Random responses are complete sentences randomly picked from the entire datasets.

4.2.1 Data Cleaning

Conversations via online forum discussion can be very different from real-world human-to-human interaction. Although reddit discussions have the thread structure of a child post replying to a previous parent post, the discussions happening there are often just people expressing their own opinions on the same topic without directly replying to their parent post. Besides, to build a conversation system, we want to avoid exposing the system to profanity. We also want to limit the text mostly to spoken language and common words. In order to do this, the following set of empirical rules are used to filter out the posts that are not desired.

1. **Profanity filters:** Python package profanity [pro] is used to filter out posts with profane words.
2. **Common word filters:** We filtered out replies with more than 3 words out of the google 10000 most common English word list [goo].
3. **Grammar checker:** Python package grammar-check [gra] is used to filter out responses with grammatical errors.
4. **Long Sentence checker:** We only use the first sentence in the response and filter out sentences with more than 30 words.

5 Rule-based Methods

5.1 Templating engine

We use Artificial Intelligence Markup (AIML) [Wal03] for our templates. Each entry in an AIML file consists of a pattern and a template. When the pattern matches the user's utterance, the template is triggered and the corresponding reply is given. Several AIML features make the system flexible. Firstly, wildcards and reductions allow an entry to match different utterances that mean the same thing. Second, AIML can capture parts of the user's utterance and use them for its response. Thirdly, AIML can set variables, and use them for matching or for supplying its dialogue. This feature is

useful as it allows us to route the conversation: When Pixie asks a question, it sets the topic to a name corresponding to the action. The entries under that topic will be possible user answers to the question, with corresponding responses. Finally, AIML can also randomly select from a list.

We use many of the base templates supplied by AIML (ALICE), adding or modifying templates to give set answers to commonly asked questions such as “Where are you?” (“I am in the Cloud.”) and set variables such as the bot’s favorite food, as well as select randomly from a list of responses in response to statements such as “Tell me something interesting.” We also added templates to guide the conversation away from controversial topics or profanity. Most importantly, we made templates within a number of topics such as books, Game of Thrones, machine learning, Oscars, and self-driving cars, and announced these topics to users to let them choose the subject of conversation. Within these templates, the bot leads the conversations, asking the user a question pertaining to the topic at every turn, and then reacting to the user’s answer.

The AIMLTemplateComponent also returns a confidence score. All the topical conversations were given a high confidence score (with some templates having an even higher score), and templates that are more generic are given low confidence scores.

However, templates are limited: everything must be hard-coded in from the start, and the bot has no access to outside information. Just using templates, only the conversation paths that we have written can be generated by the bot. Examples of AIML templates for self-driving cars are given in Figure 3. Note that we have extended the AIML language to allow “or” patterns, to allow a greater number of user utterances to be matched. Figure 4 shows a simulated conversation that calls this template.

```
<category>
  <pattern>[_|] [SELF DRIVING CARS|SELF DRIVING] [_|]</pattern>
  <template>
    Sure. It's a very interesting topic.
    <think><set name="topic">NODECARS</set></think>
    Self driving cars could be on the road very soon.
    Will you buy a self-driving car?
  </template>
</category>

<topic name="NODECARS">
<category>
  <pattern> [_|] [YES | YEAH] [_|] </pattern>
  <template>I will do the same if their prices are affordable
    but are you worried about the reported accidents
    of self driving cars?
    <think><set name="topic">SELFDIVINGACCIDENTS
    </set></think>
  </template>
</category>

<category>
  <pattern> [_|] [ NOT SURE | DONT | NO | NAH | NOT ] [_|] </pattern>
  <template>Right I saw some reported accidents
    of self driving cars.
    Are you worried about those accidents?
    <think><set name="topic">SELFDIVINGACCIDENTS
    </set></think>
  </template>
</category>
</topic>
```

Figure 3: AIML custom template: example

Bot: Hi there! I can chat about the Oscars, books, self-driving cars, machine learning. What would you like to chat about?
User: self driving cars
Bot: Sure. It's a very interesting topic. Self driving cars could be on the road very soon. Will you buy a self-driving car?
User: I don't know
Bot: Right I saw some reported accidents of self driving cars. Are you worried about those accidents?
User: yes
Bot: Yeah I think we need third party analysis on those accidents. Just need to find out if the self driving system has any flaws. What I heard is that human drivers can make much more mistakes than self driving systems. Do you think computers can be better drivers than us?
User: I'm not sure how they would do when there's ice on the road
Bot: Well maybe the algorithms are still not good enough for complicated road conditions. But in simple driving conditions, like high-way driving, computers can be more reliable because humans always get tired.
User: that makes sense

Figure 4: Sample conversation on self-driving cars

5.2 Question-answering component

For simple question-answering requests, we used Evi (<https://www.evi.com/>), an online QA service. It can answer simple, direct factual questions such as “What is the capital of France?”

5.3 Knowledge Graph

To extract detailed information about entities, we call upon the Google Knowledge Graph [Sin12] in the KnowledgeGraphComponent. Given a query, the Knowledge Graph API returns a ranked list of matches with relevancy scores; it can also be filtered by type of entity (book, person, place, etc.). Along with each match is a summary, similar to summaries on Wikipedia.

When there is a strong match to a user's utterance, and no template matches, Pixiewill respond by giving the summary of the entity obtained from the knowledge graph.

5.4 NLP

Often, giving the entire user utterance to the knowledge graph will give nonsensical results. We must extract the entity from the context, for example, “Taylor Swift” from “Who is Taylor Swift?”

We compared the performance of StanfordNLP and GoogleNLP and found that GoogleNLP worked better for us. We introduced to the system a GoogleNLPCOMPONENT which sends the query to GoogleNLP. In addition to named entity recognition, GoogleNLP can also do sentiment analysis, part-of-speech tagging, dependency parsing, etc. For our bot we only use named entity recognition. The sentiment analysis could be used to let the AIMLComponent know whether a response to a question was positive or negative, instead of hard-coding in affirmative and negative answers.

5.5 Generalized Templates

Finally, we began experimenting with ‘generalized templates’, which have the advantages of templates—structured conversations—but allow for a different conversation every time, by utilizing an external knowledge base, namely, Google Knowledge Graph.

In an entry of the template file, we now also set an ‘expecting’ field. For example, if the bot asks “What is your favorite movie?”, it sets the ‘expecting’ field to “movie”. The KnowledgeGraphComponent would then pick the top match of type “movie” and return the summary.

We add the GeneralizedTemplateComponent after the AIMLTemplateComponent and KnowledgeGraphComponent to combine the answers from the two components. In this case, in the next turn it would output a summary of the movie and ask the next question (What's your favorite actor in the

movie?). From there it can look up the actor and then continue the conversation with that information, for example, to bring up other movies in which the actor stars, etc.

6 Evaluation

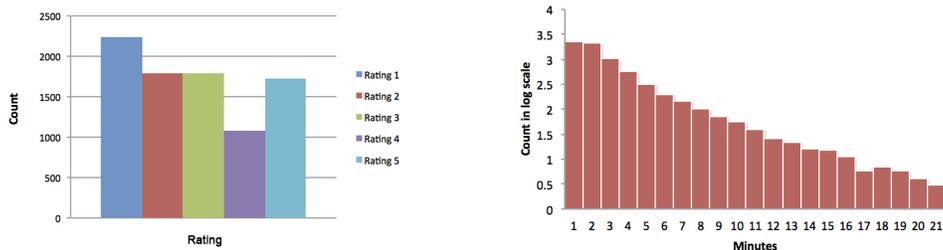


Figure 5: Distribution of ratings (left) and the distribution of conversation duration in log scale (right). Conversations presented in the diagram are filtered by feedback availability

We evaluated the performance of our chatbot through approximately 25,000 interactions with real users. Each conversation was anonymous. Bots, within Alexa competition rules, withhold contextual information about their creators in order to minimize the bias of the feedback rating. After each conversation, user rating was given from 1-5. Upon receiving feedback statistics every week, we finetuned our bot by changing parameters of components or introducing new ones. Some users also provided qualitative feedback. We have found that when using a pure question-answering system (Evi Engine) the average user rating was just 2.5; introducing templating system and other components help raise this to around 3.1.

The median and 90th percentile of conversation duration has been 1.32 and 5.12 minutes correspondingly. However, during experimentation stage we have found out that by tweaking the bot we are easily able to achieve 5 and 10 minutes accordingly, however the rating is substantially affected. During longer interactions, it is easier to discern weaknesses in the bot, and conversations have more chances to go in unexpected directions. It is however noteworthy that our bot managed several conversations that lasted more than 20 minutes with average rating of 4.

7 Conclusions

We introduced a bot architecture that combines deep learning and rule-based methods by proposing the notion of components. The current system has foundations in each domain including a templating engine, retrieval system, knowledge graph and other natural language processing toolkits. Our experience suggests that present deep learning methods are not effective in producing open-ended conversations (as opposed to goal-oriented interactions); however, one may train separate components solving specific aspects of the problem and combine these into a single system using the approach described in this paper, until better methods for end-to-end dialogue systems are developed.

Possible directions for further work include the introduction of deep learning methods in the decider component. More precisely encoding sentences using, for example, InferSent [CKS⁺17] and choosing outputs within the scope of candidate responses, as in the case of the retrieval-based system. One can also use reinforcement learning methods between switching components through temporal sequences, to learn the most effective conversation paths or strategies. Finally, generalized templates take the first steps towards integrating template utterances with a knowledge graph.

Though there is still a long way to go in harnessing methodological breakthroughs of recent years towards advancing the frontier of machine social dialogue, we believe that it is possible to achieve to open-ended conversation, by tackling first specific domains and generalizing into the arbitrary while preserving structure, through the creative combination of rule-based and deep learning methods.

References

- [BW16] Antoine Bordes and Jason Weston. Learning end-to-end goal-oriented dialog. *arXiv preprint arXiv:1605.07683*, 2016.
- [CKS⁺17] Alexis Conneau, Douwe Kiela, Holger Schwenk, Loic Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data. *arXiv preprint arXiv:1705.02364*, 2017.
- [DNML11] Cristian Danescu-Niculescu-Mizil and Lillian Lee. Chameleons in imagined conversations: A new approach to understanding coordination of linguistic style in dialogs. In *Proceedings of the Workshop on Cognitive Modeling and Computational Linguistics, ACL 2011*, 2011.
- [evi] Evi engine qa system. <https://evi.com>.
- [fla] Flask-ask. <https://github.com/johnwheeler/flask-ask>.
- [goo] Most common english words. <https://github.com/first20hours/google-10000-english>.
- [gra] Python package grammar-check.
- [HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [LJO16] Yi Luan, Yangfeng Ji, and Mari Ostendorf. Lstm based conversation models. *arXiv preprint arXiv:1603.09457*, 2016.
- [LMR⁺16] Jiwei Li, Will Monroe, Alan Ritter, Michel Galley, Jianfeng Gao, and Dan Jurafsky. Deep reinforcement learning for dialogue generation. *arXiv preprint arXiv:1606.01541*, 2016.
- [MSB⁺14] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *ACL (System Demonstrations)*, pages 55–60, 2014.
- [pro] Python package profanity. <https://pypi.python.org/pypi/profanity>.
- [red] Reddit dataset. https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/.
- [Ree08] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.
- [Ron10] Armin Ronacher. Welcome—flask (a python microframework). URL: <http://flask.pocoo.org/visited> on 02/02/2015), 2010.
- [Sin12] Amit Singhal. Introducing the knowledge graph: things, not strings. *Official google blog*, 2012.
- [Siv12] Swaminathan Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730. ACM, 2012.
- [SM13] Daniel Soutner and Luděk Müller. Application of lstm neural networks in language modelling. In *International Conference on Text, Speech and Dialogue*, pages 105–112. Springer, 2013.
- [SSB⁺16] Iulian Vlad Serban, Alessandro Sordoni, Yoshua Bengio, Aaron C Courville, and Joelle Pineau. Building end-to-end dialogue systems using generative hierarchical neural network models. In *AAAI*, pages 3776–3784, 2016.
- [SSL⁺17] Iulian Vlad Serban, Alessandro Sordoni, Ryan Lowe, Laurent Charlin, Joelle Pineau, Aaron C Courville, and Yoshua Bengio. A hierarchical latent variable encoder-decoder model for generating dialogues. 2017.

- [SSN12] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Thirteenth Annual Conference of the International Speech Communication Association*, 2012.
- [SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.
- [VL15] Oriol Vinyals and Quoc Le. A neural conversational model. *arXiv preprint arXiv:1506.05869*, 2015.
- [Wal03] Richard Wallace. The elements of aiml style. *Alice AI Foundation*, 2003.