Texar: A Modularized, Versatile, and Extensible Toolkit for Text Generation

Zhiting Hu*, **Haoran Shi**, **Bowen Tan**, **Wentao Wang**, **Zichao Yang**,
Tiancheng Zhao, **Junxian He**, **Lianhui Qin**, **Di Wang**, **Xuezhe Ma**, **Zhengzhong Liu**,
Xiaodan Liang, **Wangrong Zhu**, **Devendra Singh Sachan**, **Eric P. Xing**
Carnegie Mellon University, Petuum Inc.
zhitinghu@gmail.com*

Abstract

We introduce Texar, an open-source toolkit aiming to support the broad set of *text generation* tasks that transform any inputs into natural language, such as machine translation, summarization, dialog, content manipulation, and so forth. With the design goals of modularity, versatility, and extensibility in mind, Texar extracts common patterns underlying the diverse tasks and methodologies, creates a library of highly reusable modules and functionalities, and allows arbitrary model architectures and algorithmic paradigms. In Texar, model architecture, inference, and learning processes are properly decomposed. Modules at a high concept level can be freely assembled or plugged in/swapped out. Texar is thus particularly suitable for researchers and practitioners to do fast prototyping and experimentation. The versatile toolkit also fosters technique sharing across different text generation tasks. Texar supports both TensorFlow and PyTorch, and is released under Apache License 2.0 at https://www.texar.io.[1]

1 Introduction

Text generation spans a broad set of natural language processing tasks that aim to generate natural language from input data or machine representations. Such tasks include machine translation (Brown et al., 1990; Bahdanau et al., 2014), dialog systems (Williams and Young, 2007; Serban et al., 2016; Tang et al., 2019), text summarization (Hovy and Lin, 1998), text paraphrasing and manipulation (Madnani and Dorr, 2010; Hu et al., 2017; Lin et al., 2019), and more. Recent years have seen rapid progress of this active area, in part due to the integration of modern deep learning approaches in many of the tasks. On the other hand, considerable research efforts are still needed in order to improve techniques and enable real-world applications.

A few remarkable open-source toolkits have been developed (section 2) which largely focus on one or a few specific tasks or algorithms. Emerging new applications and approaches instead are often developed by individual teams in a more ad-hoc manner, which can easily result in hard-to-maintain custom code and duplicated efforts.

The variety of text generation tasks indeed have many common properties and share a set of key underlying techniques, such as neural encoder-decoders (Sutskever et al., 2014), attentions (Bahdanau et al., 2014; Luong et al., 2015; Vaswani et al., 2017), memory networks (Sukhbaatar et al., 2015), adversarial methods (Goodfellow et al., 2014; Lamb et al., 2016), reinforcement learning (Ranzato et al., 2015; Tan et al., 2018), structured supervision (Hu et al., 2018; Yang et al., 2018), as well as optimization techniques, data pre-processing and result post-processing, evaluations, etc. These techniques are often combined together in various ways to tackle different problems. Figure 1 summarizes examples of various model architectures.

It is therefore highly desirable to have an open-source platform that unifies the development of the diverse yet closely-related applications, backed with clean and consistent implementations of the core algorithms. Such a platform would enable reuse of common components; standardize design, implementation, and experimentation; foster reproducibility; and importantly, encourage technique sharing among tasks so that an algorithmic advance developed for a specific task can quickly be evaluated and generalized to many others.

We introduce *Texar*, a general-purpose text generation toolkit aiming to support popular and emerging applications in the field, by providing researchers and practitioners a unified and

[1] An expanded version of the tech report can be found at https://arxiv.org/abs/1809.00794

Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pages 159–164
Florence, Italy, July 28 - August 2, 2019. ©2019 Association for Computational Linguistics

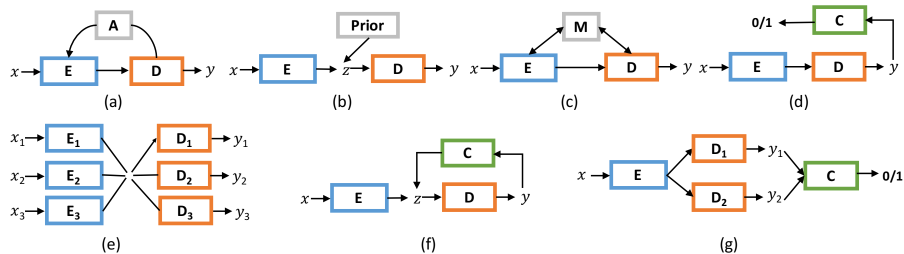

Figure 1: Examples of model architectures in recent text generation literatures (`E`: encoder, `D`: decoder, `C`: classifier). **(a):** The canonical encoder-decoder, sometimes with attentions `A` (Sutskever et al., 2014; Bahdanau et al., 2014; Vaswani et al., 2017) or copy mechanisms (Gu et al., 2016; Vinyals et al., 2015). **(b):** Variational encoder-decoder (Bowman et al., 2015). **(c):** Augmenting with external memory (Sukhbaatar et al., 2015). **(d):** Adversarial model using a binary discriminator `C`, w/ or w/o reinforcement learning (Zhang et al., 2017; Yu et al., 2017). **(e):** Multi-task learning with multiple encoders/decoders (Luong et al., 2016). **(f):** Augmenting with cyclic loss (Hu et al., 2017). **(g):** Adversarial alignment, either on samples y or hidden states (Lamb et al., 2016).

flexible framework for building their models. Texar has two versions, building upon TensorFlow (`tensorflow.org`) and PyTorch (`pytorch.org`), respectively, with the same uniform design.

Underlying the core of Texar's design is principled anatomy of extensive text generation models and learning algorithms, which subsumes the diverse cases in Figures 1 and beyond, enabling a unified formulation and consistent implementation. Texar emphasizes three key properties:

Versatility. Texar contains a wide range of features and functionalities for 1) arbitrary model architectures as a combination of encoders, decoders, embedders, discriminators, memories, and many other modules; and 2) different modeling and learning paradigms such as sequence-to-sequence, probabilistic models, adversarial methods, and reinforcement learning. Based on these, both workhorse and cutting-edge solutions to the broad spectrum of text generation tasks are either already included or can be easily constructed.

Modularity. Users can construct models at a high conceptual level just like assembling building blocks. It is convenient to plug in or swap out modules, configure rich module options, or even switch between distinct modeling paradigms. For example, switching from adversarial learning to reinforcement learning involves only minimal code changes (e.g., Figure 4). Modularity makes Texar particularly suitable for fast prototyping and experimentation.

Extensibility. The toolkit provides interfaces ranging from simple configuration files to full library APIs. Users of different needs and expertise are free to choose different interfaces for appropriate programmability and internal accessibility. The library APIs are fully compatible with the native TensorFlow/PyTorch interfaces, which allows seamless integration of user-customized modules, and enables the toolkit to take advantage of the vibrant open-source community by effortlessly importing any external components as needed.

Furthermore, Texar emphasizes on well-structured code, clean documentation, rich tutorial examples, and distributed GPU training.

2 Related Work

There exist several toolkits that focus on one or a few specific tasks. For neural machine translation and alike, there are Tensor2Tensor (Vaswani et al., 2018) on TensorFlow, OpenNMT (Klein et al., 2017) on PyTorch, Nematus (Sennrich et al., 2017) on Theano, MarianNMT (Junczys-Dowmunt et al., 2018) on C++, etc. ParlAI (Miller et al., 2017) is a specialized platform for dialogue. Differing from the task-focusing tools, Texar aims to cover as many text generation tasks as possible. The goal of versatility poses unique design challenges.

On the other end of the spectrum, there are libraries for more general NLP or ML applications: AllenNLP (`allennlp.org`), GluonNLP (`gluon-nlp.mxnet.io`) and others are designed for the broad NLP tasks in general, while Keras (`keras.io`) is for high conceptual-level programming without specific task focuses. In comparison, Texar has a proper focus on the text generation sub-area, and provide a comprehensive set of modules and functionalities that are well-tailored and readily-usable for relevant tasks. For example, Texar provides rich `text dococer` with optimized interfaces to support over ten decoding methods (see section 3.3 for an example).

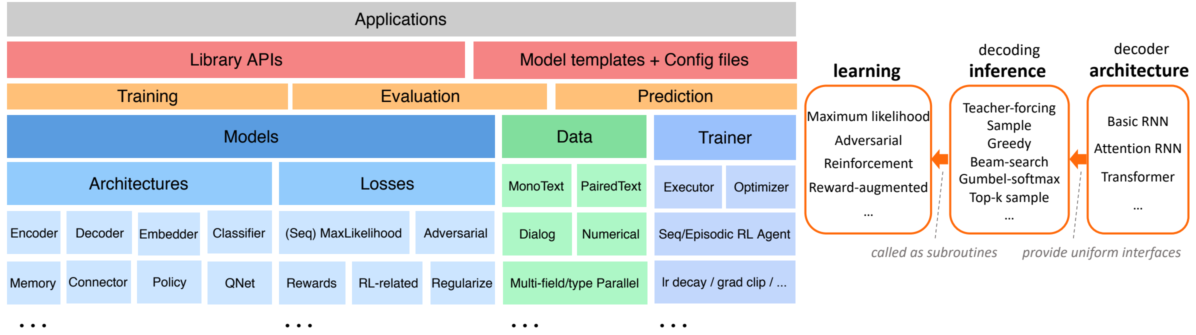

Figure 2: **Left:** The stack of main modules and functionalities in Texar. **Right:** The learning-inference-architecture anatomy, taking decoder for example. A sequence decoder can have an arbitrary architecture; all architectures expose uniform interfaces for specifying one of the tens of inference (decoding) strategies to generate samples or infer probabilities; a learning procedure repeated calls specified inference procedure during training.

3 Structure and Design

Figure 2, left panel, shows the stack of main modules and functionalities in Texar. In the following, we first present the design principles (sec 3.1) of the toolkit, and then describe the detailed structure of Texar with running examples to demonstrate the key properties (sec 3.2-3.4).

3.1 The Design of Texar

Designing a versatile toolkit is challenging due to the large variety of text generation tasks and fast-growing new models. We tackle the challenges by adopting principled anatomy of the modeling and experimentation pipeline. Specifically, we break down the complexity of rich tasks into three dimensions of variations, namely, varying data types/formats, arbitrary combinational model architectures and inference procedures, and diverse learning algorithms. Within the unified abstraction, all learning paradigms are each specifying one or multiple loss functions (e.g., cross-entropy loss, policy gradient loss), along with an optimization procedure that improves the losses:

$$\min_\theta \mathcal{L}(f_\theta, D) \quad (1)$$

where f_θ is the model that defines the model architecture and the inference procedure; D is the data; $\mathcal{L}$ is the learning objectives (losses); and **min** denotes the optimization procedure. Note that the above can have multiple losses imposed on different model parts (e.g., adversarial learning).

Further, as illustrated in Figure 2 right panel, we decouple learning, inference, and model architecture, forming abstraction layers of **learning – inference – architecture**. That is, different architectures implement the same set of inference procedures and provide the same interfaces, so that learning algorithms can call proper inference procedures as subroutines while staying agnostic to the underlying architecture and implementation details. For example, maximum likelihood learning uses teacher-forcing decoding (Mikolov et al., 2010); a policy gradient algorithm can invoke stochastic or greedy decoding (Ranzato et al., 2015); and adversarial learning can use either stochastic decoding for policy gradient-based updates (Yu et al., 2017) or Gumbel-softmax reparameterized decoding (Jang et al., 2016) for direct gradient back-propagation. Users can switch between different learning algorithms for the same model, by simply specifying the corresponding inference strategy and plugging into a new learning module, without adapting the model architecture (see section 3.3 for a running example).

3.2 Assemble Arbitrary Model Architectures

We develop an extensive set of frequently-used modules (e.g., various `encoders`, `decoders`, `embedders`, `classifiers`, etc). Crucially, Texar allows free concatenation between these modules in order to assemble arbitrary model architectures. Such concatenation can be done by directly interfacing two modules, or through an intermediate `connector` module that provides general functionalities of reshaping, reparameterization, sampling, and others.

Besides the flexibility of arbitrary assembling, it is critical for the toolkit to provide proper abstractions to relieve users from overly concerning low-level implementations. Texar provides two major types of user interfaces with different abstract levels, i.e., YAML configuration files and full Python library APIs. Figure 3 shows an exam-

```yaml
source_embedder: WordEmbedder
source_embedder_hparams:
  dim: 300
encoder: UnidirectionalRNNEncoder
encoder_hparams:
  rnn_cell:
    type: BasicLSTMCell
    kwargs:
      num_units: 300
    num_layers: 1
    dropout:
      output_dropout: 0.5
      variational_recurrent: True
embedder_share: True
decoder: AttentionRNNDecoder
decoder_hparams:
  attention:
    type: LuongAttention
beam_search_width: 5
optimization: …
```

```python
# Read data
dataset = PairedTextData(data_hparams)
batch = DataIterator(dataset).get_next()

# Encode
embedder = WordEmbedder(dataset.vocab.size, hparams=embedder_hparams)
encoder = TransformerEncoder(hparams=encoder_hparams)
enc_outputs = encoder(embedder(batch['source_text_ids']),
                      batch['source_length'])

# Decode
decoder = AttentionRNNDecoder(memory=enc_outputs,
                              hparams=decoder_hparams)
outputs, length, _ = decoder(inputs=embedder(batch['target_text_ids']),
                             seq_length=batch['target_length']-1)

# Loss
loss = sequence_sparse_softmax_cross_entropy(
    labels=batch['target_text_ids'][:,1:], logits=outputs.logits, seq_length=length)

```

Figure 3: Two ways of specifying an attentional encoder-decoder model. **Left:** Part of an example YAML config file of the model template. Hyperparameters taking default values can be omitted in the file. **Right:** Python code assembling an encoder-decoder model using library APIs. Modules are created as Python objects, and called as functions to create computation operations and return output tensors. Other code such as optimization is omitted.

ple of specifying an attentional encoder-decoder model through the two interfaces, respectively.

Configuration file passes hyperparameters to a predefined model template, which instantiates the model for training and evaluation. Text highlighted in blue in the figure (left panel) specifies the names of modules to use. Most hyperparameters have sensible default values. Users only have to specify hyperparameter values that differ from the default. The interface is easily understandable for non-expert users, and has also been adopted in other tools (e.g., Klein et al., 2017).

Library APIs offer clean function calls. Users can efficiently build any desired pipelines at a high conceptual level. Power users have the option to access the full internal states for low-level manipulations. Texar modules support convenient *variable re-use*. That is, each module instance creates its own sets of variables, and automatically re-uses them on subsequent calls. Hence TensorFlow *variable scope* is transparent to users.

3.3 Plug-in and Swap-out Modules

It is convenient to change from one modeling paradigm to another by simply plugging in/swapping out a single or few modules, or even merely changing a configuration parameter. For example, given the base code of an encoder-decoder model in Figure 3 (right panel), Figure 4 illustrates how one can switch between different learning paradigms by changing only Lines.14–19 of the original code (maximum-likelihood learning). In particular, Figure 4 shows adversarial learning and reinforcement learning, which invokes Gumbel-softmax decoding and random-sample decoding, respectively.

3.4 Customize with Extensible Interfaces

Texar emphasizes on extensibility and allows easy addition of customized/external modules without editing the Texar codebase. Specifically, with the YAML configuration file, users can directly insert their own modules by providing the Python importing path to the module. For example, to use a customized RNN cell in the encoder, one can simply change Line.7 of Figure 3 (left panel) to `type: path.to.MyCell`, as long as `MyCell` has a compatible interface to other parts of the model. Using customized modules with the library APIs is even more flexible, since the APIs are designed to be fully compatible with native TensorFlow/PyTorch programming interfaces.

4 Case Study: Transformer on Different Tasks

We present a case study to show that Texar can greatly reduce implementation efforts and enable technique sharing among different tasks. Transformer, as first introduced in (Vaswani et al., 2017), has greatly improved the machine translation results and created other successful models such as BERT for text embedding (Devlin et al., 2019) and GPT-2 for language modeling (Radford et al., 2018). Texar supports easy construction of

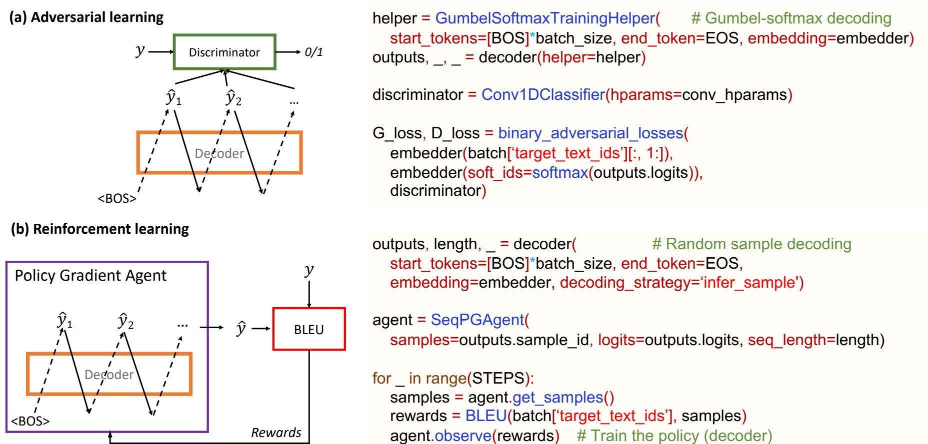

Figure 4: Switching between different learning paradigms of a decoder involves only modification of Line.14-19 of Figure 3 (maximum-likelihood learning). The same decoder is called with different decoding modes, and discriminator or reinforcement learning agent is added as needed. **(Left):** Module structure of each paradigm; **(Right):** The respective code snippets. For adversarial learning in (b), continuous Gumbel-softmax approximation (Jang et al., 2016) to generated samples is used to enable gradient propagation from the discriminator to the decoder.

these models and fine-tuning pretrained weights. We can also deploy the Transformer components to various other tasks and get improved results.

The first task we explored is the variational autoencoder (VAE) language modeling (Bowman et al., 2015). We test two models, one with an LSTM RNN decoder which is traditionally used in the task, and the other with a Transformer decoder. All other model configurations including parameter size are the same across the two models. Table 1, top panel, shows the Transformer VAE consistently improves over the LSTM VAE. With Texar, changing the decoder from an LSTM to a Transformer is easily achieved by modifying only 3 lines of code. It is also worth noting that, building the VAE language model (including data reading, model construction, and optimization) on Texar uses only 70 lines of code (with the length of each line $<$ 80 chars). As a (rough) reference, a popular public TensorFlow code (Li, 2017) of the same model has used around 400 lines of code for the same part (without line length limit).

The second task is conversation generation. The dialog history is encoded with the `HierarchicalRNNEncoder` module which is followed by a decoder to generate the response. We study the performance of a Transformer decoder v.s. a conventional GRU RNN decoder. Table 1, bottom panel, shows the Transformer outperforms GRU. Regarding the implementation effort, the Texar code has around 100 lines of code, while the reference TensorFlow code (Zhao et al., 2017) involves over 600 lines.

Dataset	Metrics	VAE-LSTM	VAE-Tran
Yahoo (Yang et al.)	PPL NLL	68.31 337.36	**61.26** **328.67**
PTB (Bowman et al.)	PPL NLL	105.27 102.06	**102.46** **101.46**
Dataset	**Metrics**	**HERD-GRU**	**HERD-Tran**
Switchboard (Zhao et al.)	BLEU4-p BLEU4-r	0.228 0.205	**0.232** **0.214**

Table 1: **Top:** Transformer *vs* LSTM for VAE LM. Perplexity (PPL) and sentence negative log likelihood (NLL) are evaluated (The lower the better). **Bottom:** Transformer *vs* GRU decoders in HERD (Serban et al., 2016) for conversation response generation. BLEU4-p and -r are precision and recall (Zhao et al., 2017).

References

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv:1409.0473*.

Samuel R Bowman, Luke Vilnis, Oriol Vinyals, Andrew M Dai, Rafal Jozefowicz, and Samy Bengio. 2015. Generating sentences from a continuous space. *arXiv:1511.06349*.

Peter F Brown, John Cocke, Stephen A Della Pietra, Vincent J Della Pietra, Fredrick Jelinek, John D Lafferty, Robert L Mercer, and Paul S Roossin. 1990. A statistical approach to machine translation. *CL*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. *NAACL*.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. *NeurIPS*.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK

Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. *arXiv:1603.06393*.

Eduard Hovy and Chin-Yew Lin. 1998. Automated text summarization and the SUMMARIST system. In *Advances in automatic text summarization*.

Zhiting Hu, Zichao Yang, Xiaodan Liang, Ruslan Salakhutdinov, and Eric P Xing. 2017. Toward controlled generation of text. In *ICML*.

Zhiting Hu, Zichao Yang, Ruslan Salakhutdinov, Xiaodan Liang, Lianhui Qin, Haoye Dong, and Eric Xing. 2018. Deep generative models with learnable knowledge constraints. In *NeurIPS*.

Eric Jang, Shixiang Gu, and Ben Poole. 2016. Categorical reparameterization with Gumbel-softmax. *arXiv:1611.01144*.

Marcin Junczys-Dowmunt, Roman Grundkiewicz, Tomasz Dwojak, Hieu Hoang, Kenneth Heafield, Tom Neckermann, Frank Seide, Ulrich Germann, Alham Fikri Aji, Nikolay Bogoychev, Andr F. T. Martins, and Alexandra Birch. 2018. Marian: Fast neural machine translation in C++. *arXiv:1804.00344*.

Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M Rush. 2017. OpenNMT: Open-source toolkit for neural machine translation. *arXiv:1701.02810*.

Alex M Lamb, Anirudh Goyal Alias Parth Goyal, Ying Zhang, Saizheng Zhang, Aaron C Courville, and Yoshua Bengio. 2016. Professor forcing: A new algorithm for training recurrent networks. In *NeurIPS*.

Zhong-Yi Li. 2017. https://github.com/Chung-I/Variational-Recurrent-Autoencoder-Tensorflow.

Shuai Lin, Wentao Wang, Zichao Yang, Haoran Shi, Frank Xu, Xiaodan Liang, Eric Xing, and Zhiting Hu. 2019. Towards unsupervised text content manipulation. *arXiv preprint arXiv:1901.09501*.

Minh-Thang Luong, Quoc V Le, Ilya Sutskever, Oriol Vinyals, and Lukasz Kaiser. 2016. Multi-task sequence to sequence learning. In *ICLR*.

Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. *arXiv:1508.04025*.

Nitin Madnani and Bonnie J Dorr. 2010. Generating phrasal and sentential paraphrases: A survey of data-driven methods. *CL*.

Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *INTERSPEECH*.

Alexander H Miller, Will Feng, Adam Fisch, Jiasen Lu, Dhruv Batra, Antoine Bordes, Devi Parikh, and Jason Weston. 2017. ParlAI: A dialog research software platform. *arXiv:1705.06476*.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2018. Language models are unsupervised multitask learners. Technical report, Technical report, OpenAI.

Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2015. Sequence level training with recurrent neural networks. *arXiv:1511.06732*.

Rico Sennrich, Orhan Firat, Kyunghyun Cho, Alexandra Birch, Barry Haddow, Julian Hitschler, Marcin Junczys-Dowmunt, Samuel Läubli, et al. 2017. Nematus: a toolkit for neural machine translation. *arXiv:1703.04357*.

Iulian Vlad Serban, Alessandro Sordoni, Yoshua Bengio, Aaron C Courville, and Joelle Pineau. 2016. Building end-to-end dialogue systems using generative hierarchical neural network models.

Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. 2015. End-to-end memory networks. In *NeurIPS*.

Ilya Sutskever, Oriol Vinyals, and Quoc Le. 2014. Sequence to sequence learning with neural networks. *NeurIPS*.

Bowen Tan, Zhiting Hu, Zichao Yang, Ruslan Salakhutdinov, and Eric Xing. 2018. Connecting the dots between MLE and RL for sequence generation. *arXiv:1811.09740*.

Jianheng Tang, Tiancheng Zhao, Chengyan Xiong, Xiaodan Liang, Eric P Xing, and Zhiting Hu. 2019. Target-guided open-domain conversation. In *ACL*.

Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, et al. 2018. Tensor2Tensor for neural machine translation. *arXiv:1803.07416*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NeurIPS*.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *NeurIPS*.

Jason D Williams and Steve Young. 2007. Partially observable Markov decision processes for spoken dialog systems. *CSL*.

Zichao Yang, Zhiting Hu, Chris Dyer, Eric P Xing, and Taylor Berg-Kirkpatrick. 2018. Unsupervised text style transfer using language models as discriminators. *arXiv:1805.11749*.

Zichao Yang, Zhiting Hu, Ruslan Salakhutdinov, and Taylor Berg-Kirkpatrick. 2017. Improved variational autoencoders for text modeling using dilated convolutions. *ICML*.

Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. 2017. SeqGAN: Sequence generative adversarial nets with policy gradient. In *AAAI*.

Yizhe Zhang, Zhe Gan, Kai Fan, Zhi Chen, Ricardo Henao, Dinghan Shen, and Lawrence Carin. 2017. Adversarial feature matching for text generation. *arXiv:1706.03850*.

Tiancheng Zhao, Ran Zhao, and Maxine Eskenazi. 2017. Learning discourse-level diversity for neural dialog models using conditional variational autoencoders. In *ACL*.