PyText: A seamless path from NLP research to production

Ahmed Aly [1] **Kushal Lakhotia** [1] **Shicong Zhao** [1] **Mrinal Mohit** [1] **Barlas Oğuz** [2] **Abhinav Arora** [1] **Sonal Gupta** [1]
Christopher Dewan [2] **Stef Nelson-Lindall** [2] **Rushin Shah** [1]

[1]Facebook Conversational AI
[2]Facebook AI

Abstract

We introduce PyText[1] – a deep learning based NLP modeling framework built on PyTorch. PyText addresses the often-conflicting requirements of enabling rapid experimentation and of serving models at scale. It achieves this by providing simple and extensible interfaces for model components, and by using PyTorch's capabilities of exporting models for inference via the optimized Caffe2 execution engine. We report our own experience of migrating experimentation and production workflows to PyText, which enabled us to iterate faster on novel modeling ideas and then seamlessly ship them at industrial scale.

1 Introduction

When building a machine learning system, especially one based on neural networks, there is usually a trade-off between ease of experimentation and deployment readiness, often with conflicting requirements. For instance, to rapidly try out flexible and non-conventional modeling ideas, researchers tend to use modern imperative deep-learning frameworks like PyTorch[2] or TensorFlow Eager[3]. These frameworks provide an easy, eager-execution interface that facilitates writing advanced and dynamic models quickly, but also suffer from overhead in latency at inference and impose deployment challenges. In contrast, production-oriented systems are typically written in declarative frameworks that express the model as a static graph, such as Caffe2[4] and Tensorflow[5]. While being highly optimized for production scenarios, they are often harder to use, and make the experimentation life-cycle much longer. This conflict is even more prevalent in natural language processing (NLP) systems, since most NLP models are inherently very dynamic, and not easily expressible in a static graph. This adds to the challenge of serving these models at an industrial scale.

PyText, built on PyTorch 1.0 [6], is designed to achieve the following:

1. Make experimentation with new modeling ideas as easy and as fast as possible.
2. Make it easy to use pre-built models on new data with minimal extra work.
3. Define a clear workflow for both researchers and engineers to build, evaluate, and ship their models to production with minimal overhead.
4. Ensure high performance (low latency and high throughput) on deployed models at inference.

NLP Framework	Deep Learning Support	Easy Prototyping	Industrial Performance
CoreNLP	×	✓	✓
AllenNLP	✓	✓	×
FLAIR	✓	✓	×
Spacy 2.0	✓	×	✓
PyText	✓	✓	✓

Table 1. Comparison of NLP Modeling Frameworks

Existing popular frameworks for building state-of-the-art NLP models include Stanford CoreNLP (Manning et al., 2014), AllenNLP (Gardner et al., 2017), FLAIR (Akbik et al., 2018) and Spacy 2.0 [7]. CoreNLP has been a popular library for both research and production, but does not support neural network models very well. AllenNLP and

[2]https://pytorch.org/
[3]https://www.tensorflow.org/guide/eager
[4]https://caffe2.ai/
[5]https://www.tensorflow.org/
[6]https://pytorch.org/blog/the-road-to-1_0/
[7]http://spacy.io

FLAIR are easy-to-use for prototypes but it is hard to productionize the models since they are in Python, which doesn't support large scale real time requests due to lack of good multi-threading support. Spacy 2.0 has some state-of-the-art NLP models built for production use-cases but is not easily extensible for quick prototyping and building new models.

2 Framework Design

PyText is a modeling framework that helps researchers and engineers build end-to-end pipelines for training or inference. Apart from workflows for experimentation with model architectures, it provides ways to customize handling of raw data, reporting of metrics, training methodology and exporting of trained models. PyText users are free to implement one or more of these components and can expect the entire pipeline to work out of the box. A number of default pipelines are implemented for popular tasks which can be used as-is. We now dive deeper into building blocks of the framework and its design.

2.1 Component

Everything in PyText is a component. A component is clearly defined by the parameters required to configure it. All components are maintained in a global registry which makes PyText aware of them. They currently include –

Task: combines various components required for a training or inference task into a pipeline. Figure 1 shows a sample config for a document classification task. It can be configured as a JSON file that defines the parameters of all the children components.

Data Handler: processes raw input data and prepare batches of tensors to feed to the model.

Model: defines the neural network architecture.

Optimizer: encapsulates model parameter optimization using loss from forward pass of the model.

Metric Reporter: implements the relevant metric computation and reporting for the models.

Trainer: uses the data handler, model, loss and optimizer to train a model and perform model selection by validating against a holdout set.

Predictor: uses the data handler and model for inference given a test dataset.

Exporter: exports a trained PyTorch model to a Caffe2 graph using ONNX[8].

[8] https://onnx.ai/

```
{
  "config": {
    "task": {
      "DocClassificationTask": {
        "data_handler": {
          "columns_to_read": ["doc_label", "text"],
          "shuffle": true
        },
        "model": {
          "representation": {
            "BiLSTMPooling": {
              "pooling": {
                "SelfAttention": {
                  "attn_dimension": 128,
                  "dropout": 0.4
                }
              },
              "bidirectional": true,
              "dropout": 0.4,
              "lstm": { "lstm_dim": 200, "num_layers":
                  2 }
            }
          },
          "output_config": {
            "loss": { "CrossEntropyLoss": {} }
          },
          "decoder": { "hidden_dims": [128] }
        },
        "features": {
          "word_feat": {
            "embed_dim": 200,
            "pretrained_embeddings_path": "/tmp/embeds"
                ,
            "vocab_size": 250000,
            "vocab_from_train_data": true
          }
        },
        "trainer": {
          "random_seed": 0,
          "epochs": 15,
          "early_stop_after": 0,
          "log_interval": 1,
          "eval_interval": 1,
          "max_clip_norm": 5
        },
        "optimizer": {
          "type": "adam",
          "lr": 0.001,
          "weight_decay": 0.00001
        },
        "metric_reporter": {
          "output_path": "/tmp/test_out.txt"
        },
        "exporter": {}
      }
    }
  }
}
```

Figure 1. Document Classification Task Config

2.2 Design Overview

The task bootstraps a PyText job and creates all the required components. There are two modes in which a job can be run:

- *Train*: Trains a model either from scratch or from a saved check-point. Task uses the Data Handler to create batch iterators over training, evaluation and test data-

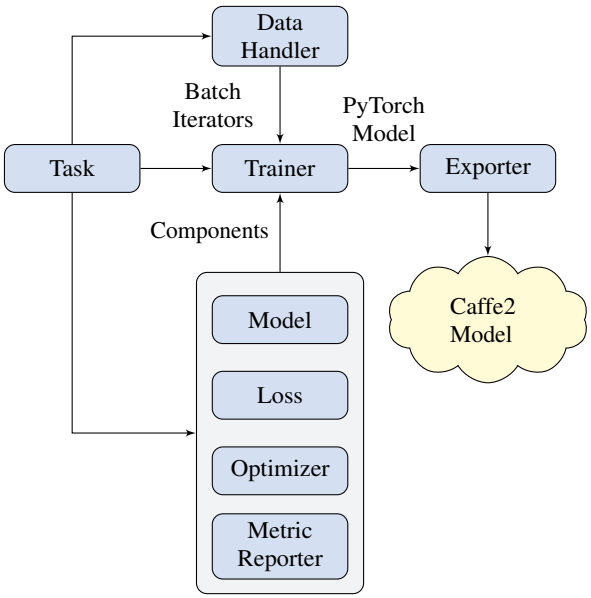

Figure 2. PyText Framework Design

sets and passes these iterators along with model, optimizer and metrics reporter to the trainer. Subsequently, the trained model is serialized in PyTorch format as well as converted to a static Caffe2 graph.

- *Predict*: Loads a pre-trained model and computes its prediction for a given test set. The task Manager, again, uses the Data Handler to create a batch iterator over the test data-set and passes it with the model to the predictor for inference.

Figure 2 illustrates the overall design of the framework.

3 Modeling Support

We now discuss the native support for building and extending models in PyText.

3.1 Terminology

Module: is a reusable component that is implemented without any knowledge of which model it will be used in. It defines a clear input and output interface such that it can be plugged into another module or model.

Model: has a one-to-one mapping with a task. Each model can be made up of a combination of modules for running a training or prediction job.

3.2 Model Abstraction

PyText provides a simple, easily extensible model abstraction. We break up a single-task model into Token Embedding, Representation, Decoder and Output layers, each of which is configurable. Further, each module can be saved and loaded individually to be reused in other models.

Token Embedding: converts a batch of numericalized tokens into a batch of vector embeddings for each token. It can be configured to use embeddings of a number of styles: pre-trained word-based, trainable word-based, character-based with CNN and highway networks(Kim et al., 2016), pre-trained deep contextual character-based (e.g., ELMo(Peters et al., 2018)), token-level gazetteer features or morphology-based (e.g. capitalization).

Representation: processes a batch of embedded tokens to a representation of the input. The implementation of what it emits as output depends on the task, e.g., the representation of the document for a text classification task will differ from that for a word tagging task. Logically this part of the model should implement the sub-network such that its output can be interpreted as features over the input. Examples of the different representations that are present in PyText are; Bidirectional LSTM and CNN representations.

Decoder: is responsible for generating logits from the input representation. Logically this part of the model should implement the sub-network that generates model output over the features learned by the representation.

Output Layer: concerns itself with generating prediction and the loss (when label or ground truth is provided).

These modules compose the base model implementation, they can be easily extended for more complicated architectures.

3.3 Multi-task Model Training

PyText supports multi-task training (Collobert & Weston, 2008) to optimize multiple tasks jointly as a first-class citizen. We use multi-task model by allowing parameter sharing between modules of the multiple single task models. We use the model abstraction for single task discussed in Section 3.2 to define the tasks and let the user declare which modules of those single tasks should be shared. This enables training a model with one or more input representations jointly against multiple tasks.

Multi-task models make the following assumptions:

- If there are n tasks in the multi-task model setup then there must be n data sources containing data for one task each.

- The single task scenario must be implemented for it to be reused for the multi-task setup.

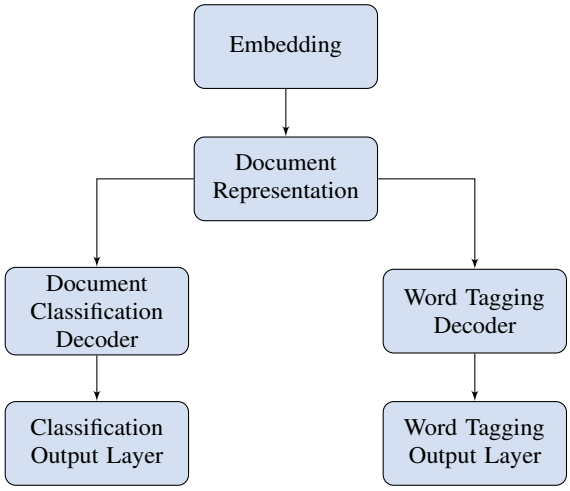

Figure 3. Joint document classification and word tagging model

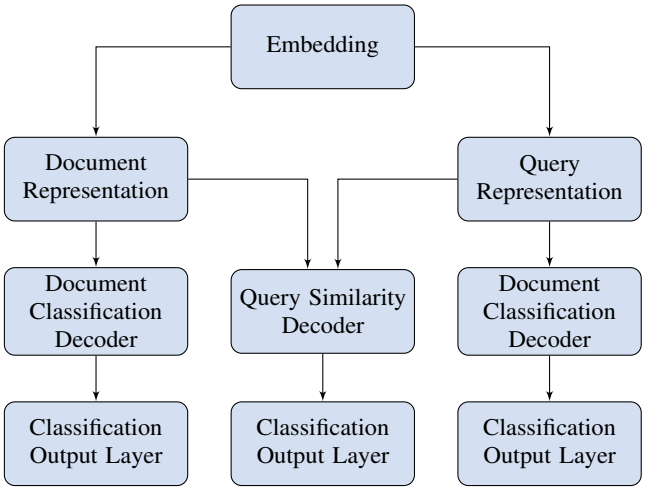

Figure 4. Joint query-document relevance and document classification model

3.3.1 *Multi-task Model Examples*

PyText provides the flexibility of building any multi-task model architecture with the appropriate model configuration, if the two assumptions listed above are satisfied. The examples below give a flavor of two sample model architectures built with PyText for joint learning against more than one task.

Figure 3 illustrates a model that learns a shared document representation for document classification and word tagging tasks. This model is useful for natural language understanding where given a sentence, we want to predict the intent behind it and tag the slots in the sentence. Jointly optimizing for two tasks helps the model learn a robust sentence representation for the two tasks. Further, we can use this pre-trained sentence representation for other tasks where training data is scarce.

Figure 4 illustrates a model that learns document and query representations using query-document relevance and individual query and document classification tasks. This is often used in information retrieval where, given a query and a document, we want to predict their relevance; but we also add query and document classification tasks to increase robustness of learned representations.

3.4 Model Zoo

PyText models are focused on NLP tasks that can be configured with a variety of modules. We enumerate here the classes of models that are currently supported.

- Text Classification: classifies a sentence or a document into an appropriate category. PyText includes reference implementations of Bidirectional LSTM (Schuster & Paliwal, 1997) with Self-Attention (Lin et al., 2017) and Convolutional Neural Network (Kim, 2014) models for text classification.
- Word Tagging: labels word sequences, i.e. classifies each word in a sequence to an appropriate category. Common examples of such tasks include Part-of-Speech (POS) tagging, Named Entity Recognition (NER) and Slot Filling in spoken language understanding. PyText contains reference implementations of Bidirectional LSTM with Slot-Attention and Bidirectional Sequential Convolutional Neural Network (Vu, 2016) for word tagging.
- Semantic Parsing: maps a natural language sentence into a formal representation of its meaning. PyText provides a reference implementation for Recurrent Neural Network Grammars (Dyer et al., 2016) (Gupta et al., 2018) for semantic parsing.
- Language Modeling: assigns a probability to a sequence of words (sentence) in a language. It also assigns a probability for the likelihood of a given word to follow a sequence of words. PyText provides a reference implementation for a stacked LSTM Language Model (Mikolov et al., 2010).
- Joint Models: We utilize the multi-task training support illustrated earlier to fuse and train models for two or more of the tasks mentioned here and optimize their parameters jointly.

4 PRODUCTION WORKFLOW

4.1 From Idea to Production

Researchers and engineers can follow the following steps to validate their ideas and quickly ship them to production –

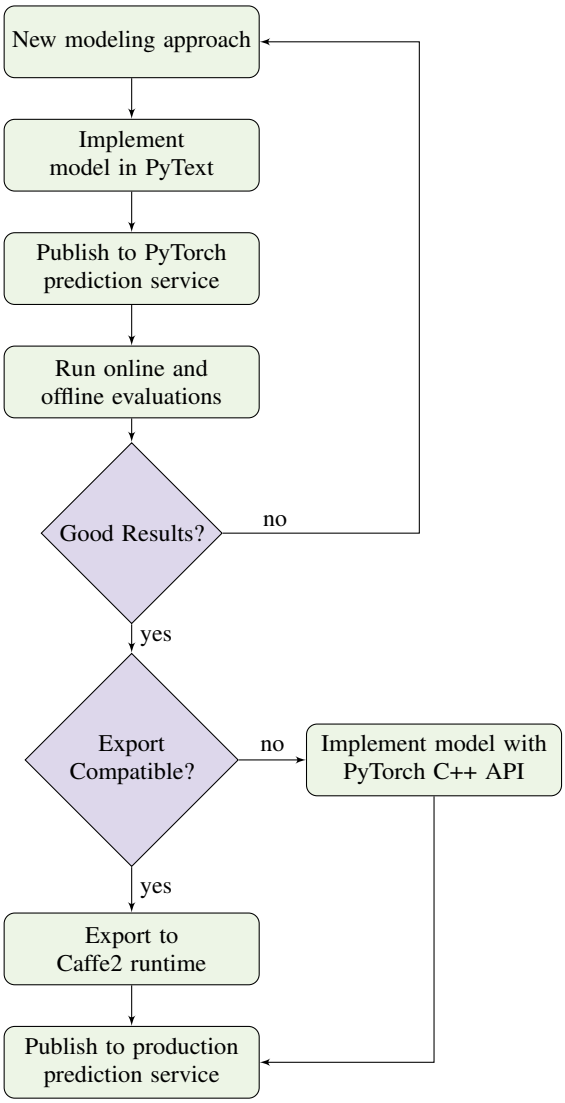

Figure 5. From Idea to Production flowchart

1. Implement the model in PyText, and make sure offline metrics on the test set look good.

2. Publish the model to the bundled PyTorch-based inference service, and do a real-time small scale evaluation on a live traffic sample.

3. Export it automatically to a Caffe2 net. In some cases, e.g. when using complex control flow logic and custom data-structures, this might not yet be supported via PyTorch 1.0.

4. If the procedure in 3 isn't supported, use the PyTorch C++ API[9] to rewrite the model (only the torch.nn.Module[10] subclass) and wrap it in a Caffe2 operator.

5. Publish the model to the production-grade Caffe2 prediction service and start serving live traffic

4.2 Benchmarks

Model	Implementation	P50	P90	P99
JointBLSTM	PyTorch	34.08	47.23	64.94
	Exported to Caffe2	**19.65**	**24.69**	**30.21**
RNNG	PyTorch	19.74	28.53	36.37
	PyTorch C++	**18.73**	**25.47**	**32.63**

Table 2. Latency Comparison (in milliseconds, smaller is better) of Python and C++ implementations of PyText models

We compared the performance of Python and C++ models (either directly exported to Caffe2 or re-written with the PyTorch C++ API[11]) on an intent-slot detection task. We note that porting to C++ gave significant latency boosts (Table 2) for the JointBLSTM model and a slight boost for the RNNG model. The latter is still valuable though, since the highly performant production serving infrastructure in many companies don't support Python code.

The experiments were performed on a CPU-only machine with 48 Intel Xeon E5-2680 processors clocked at 2.5GHz, with 251 GB RAM and CentOS 7.5. The C++ code was compiled with gcc -O3.

4.3 Production Challenges

4.3.1 Data pre-processing

One limitation of PyTorch is that it doesn't support string tensors; which means that any kind of string manipulation and indexing needs to happen outside the model. This is easy during training, but makes productionization of the model tricky. We addressed this by writing a featurization library in C++[11]. This is accessible during training via Pybind[12] and at inference as part of the runtime services suite shown in Figure 6. This library preprocesses the raw input by performing tasks like –

[9] https://pytorch.org/cppdocs/
[10] https://pytorch.org/docs/stable/nn.html#module
[11] Currently not a part of PyText's open-source repository
[12] https://github.com/pybind/pybind11

- Text tokenization and normalization
- Mapping characters to IDs for character-based models
- Perform token alignments for gazetteer features

By sharing the featurization code across training and inference we ensure data consistency in the different stages of the model.

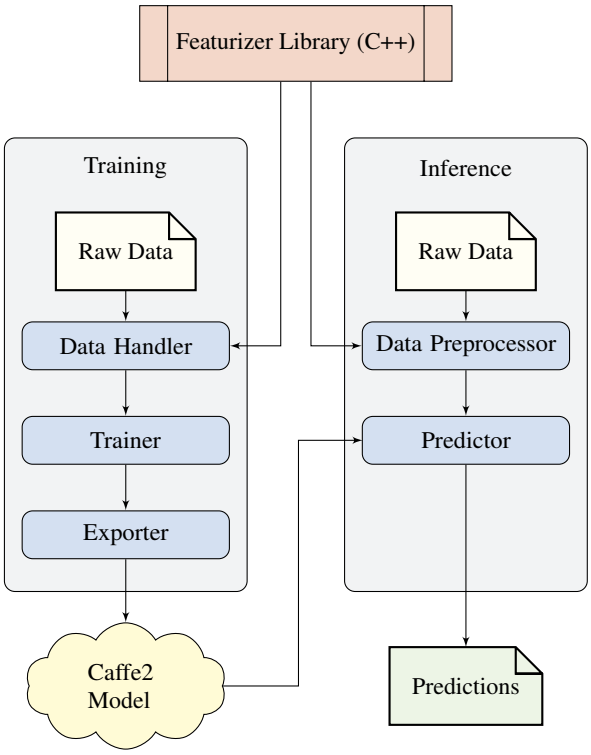

Figure 6. Training and Inference Workflow Architecture

4.3.2 *Vocabulary management*

Another consequence of string tensors not being supported yet is that we can't maintain vocabularies inside the model. We explored two solutions to this –

- Maintain the vocabularies in the remote featurization service.
- After exporting the model, post-process the resultant Caffe2 graph and prepend the vocabularies to the net

We ultimately opted for the second option since its non-trivial to maintain synchronization and versioning between training-time and test-time vocabularies, across different use cases and languages.

5 Future Work

Upcoming enhancements to PyText span multiple domains:

- **Modeling Capabilities:** Adding support for advanced NLP models for more use cases, e.g.
  - Question answering, reading comprehension and summarization tasks
  - Multilingual and language-agnostic tasks
- **Performance Benchmarks and Improvements :** A core goal of PyText is to enable building highly scalable models, with can run with low latency and high throughput. We plan to invest in –
  - Training speed – by augmenting the current distributed-training support with lower precision computations support like fp16[13]
  - Inference speed – by benchmarking performance and tuning the model deployment for expected load patterns.
- **Model Interpretability:** We plan to add more tooling support for monitoring metrics and debugging model internals –
  - Tensorboard[14] and Visdom [15] integration for visualizing the different layers of the models and track evaluation metrics during training
  - Explore and implement different model explanation approaches, e.g LIME [16] and SHAP (Lundberg & Lee, 2017)
- **Model Robustness:** Adversarial input, noise, and differences in grammar and syntax can often hurt model accuracy. To analyze and improve robustness against these perturbations, we plan to invest in adversarial training and data augmentation techniques.
- **Mobile Deployment Support:** We utilize the optimized Caffe2 runtime engine to serve our models, and plan to leverage its optimization for mobile devices [17], as well as support training light-weight models.

6 Conclusion

In this paper we presented PyText – a new NLP modeling platform built on PyTorch. It blurs the boundaries between

[13] https://en.wikipedia.org/wiki/Half-precision_floating-point_format
[14] https://github.com/tensorflow/tensorboard
[15] https://github.com/facebookresearch/visdom
[16] https://github.com/marcotcr/lime
[17] https://caffe2.ai/docs/mobile-integration.html

experiments and large scale deployment and makes it easy for both researchers and engineers to rapidly try out new modeling ideas and then productionize them. It does so by providing an extensible framework for adding new models and by defining a clear production workflow for rigorously evaluating and serving them. Using this framework and the processes defined here, we significantly reduced the time required for us to take models from research ideas to industrial-scale production.

REFERENCES

Akbik, A., Blythe, D., and Vollgraf, R. Contextual string embeddings for sequence labeling. In *COLING 2018, 27th International Conference on Computational Linguistics*, pp. 1638–1649, 2018.

Collobert, R. and Weston, J. A unified architecture for natural language processing: deep neural networks with multitask learning. pp. 160–167, 2008.

Dyer, C., Kuncoro, A., Ballesteros, M., and Smith, N. A. Recurrent neural network grammars. In *HLT-NAACL*, pp. 199–209. The Association for Computational Linguistics, 2016.

Gardner, M., Grus, J., Neumann, M., Tafjord, O., Dasigi, P., Liu, N. F., Peters, M., Schmitz, M., and Zettlemoyer, L. S. Allennlp: A deep semantic natural language processing platform. 2017.

Gupta, S., Shah, R., Mohit, M., Kumar, A., and Lewis, M. Semantic parsing for task oriented dialog using hierarchical representations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

Kim, Y. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1746–1751. Association for Computational Linguistics, 2014. doi: 10.3115/v1/D14-1181. URL http://www.aclweb.org/anthology/D14-1181.

Kim, Y., Jernite, Y., Sontag, D., and Rush, A. M. Character-aware neural language models. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pp. 2741–2749, 2016. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12489.

Lin, Z., Feng, M., dos Santos, C. N., Yu, M., Xiang, B., Zhou, B., and Bengio, Y. A structured self-attentive sentence embedding. 2017. URL https://openreview.net/forum?id=BJC_jUqxe.

Lundberg, S. and Lee, S. A unified approach to interpreting model predictions. *CoRR*, abs/1705.07874, 2017. URL http://arxiv.org/abs/1705.07874.

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. J., and McClosky, D. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pp. 55–60, 2014. URL http://www.aclweb.org/anthology/P/P14/P14-5010.

Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., and Khudanpur, S. Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, pp. 1045–1048, 2010. URL http://www.isca-speech.org/archive/interspeech_2010/i10_1045.html.

Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 2227–2237. Association for Computational Linguistics, 2018. URL http://aclweb.org/anthology/N18-1202.

Schuster, M. and Paliwal, K. Bidirectional recurrent neural networks. *Trans. Sig. Proc.*, 45(11):2673–2681, November 1997. ISSN 1053-587X. doi: 10.1109/78.650093. URL http://dx.doi.org/10.1109/78.650093.

Vu, N. T. Sequential convolutional neural networks for slot filling in spoken language understanding. In *Interspeech 2016*, pp. 3250–3254, 2016. doi: 10.21437/Interspeech.2016-395. URL http://dx.doi.org/10.21437/Interspeech.2016-395.