A Progressive Model to Enable Continual Learning for Semantic Slot Filling

Yilin Shen[1] **Xiangyu Zeng**[2*] **Hongxia Jin**[1]
[1] Samsung Research America, Mountain View, CA, USA
[2] Columbia University, New York, NY, USA
[1] {yilin.shen,hongxia.jin}@samsung.com [2] xz2571@columbia.edu

Abstract

Semantic slot filling is one of the major tasks in spoken language understanding (SLU). After a slot filling model is trained on pre-collected data, it is crucial to continually improve the model after deployment to learn users' new expressions. As the data amount grows, it becomes infeasible to either store such huge data and repeatedly retrain the model on all data or fine tune the model only on new data without forgetting old expressions. In this paper, we introduce a novel progressive slot filling model, *ProgModel*. ProgModel consists of a novel context gate that transfers previously learned knowledge to a small size expanded component; and meanwhile enables this new component to be fast trained to learn from new data. As such, ProgModel learns the new knowledge by only using new data at each time and meanwhile preserves the previously learned expressions. Our experiments show that ProgModel needs much less training time and smaller model size to outperform various model fine tuning competitors by up to 4.24% and 3.03% on two benchmark datasets.

1 Introduction

Spoken language understanding (SLU) systems play a vital role in ubiquitous artificially intelligent voice-enabled personal assistants. As one of the major tasks in SLU, semantic slot filling is treated as a sequential labeling problem to map a natural language sequence $\mathbf{x}$ to a slot label sequence $\mathbf{y}$ of the same length in IOB format (Yao et al., 2014). Typically, a slot filling model is trained offline on large scale corpora with pre-collected utterances. However, such corpora usually cannot cover all possible varieties of utterances exhaustively (e.g., personalized expressions, new vocabulary, utterances for new intent, etc.) from diverse users. Thus, it is critically desirable to develop a slot filling approach with the capability of continual learning after a personal assistant is deployed.

Unfortunately, existing approaches target on offline model training using a large scale training data. They are designed to either train a slot filling model independently (Yao et al., 2014; Peng et al., 2015; Kurata et al., 2016; Hakkani-Tür et al., 2016; Liu and Lane, 2016; Deng et al., 2019; Ray et al., 2019) or jointly with the other intent detection task in SLU (Guo et al., 2014; Liu and Lane, 2016; Zhang and Wang, 2016; Wang et al., 2018; Goo et al., 2018). Recently, (Shen et al., 2018a, 2019) developed cold start algorithms to generate training data with the hope of covering more varieties before deployment. On the other hand, (Ray et al., 2018; Shen et al., 2018b) attempt to personalize the slot filling model. However, they are still restricted to the offline training and cannot be applied to learn new user's expressions after deployment.

To support continual learning, a naive solution is to retrain the current model at each time. However, it suffers from several drawbacks: First, in order to maintain the SLU performance on both original and new expressions, it usually requires almost retraining the model using the whole dataset. However, as the size of training set grows, it becomes infeasible to repeatedly conduct time consuming retraining on such a large dataset. More importantly, the old training data typically is not stored permanently due to huge storage need and privacy protection. If only fine tuned on new utterances, the new model intends to lose the previously learned knowledge, a.k.a., *catastrophic forgetting* (French, 1999). Moreover, such fine-tuning of a large generic SLU model, even only on new utterances, is still quite inefficient. Recently, there are some progresses on continual learning in

[*] This work was done when Xiangyu Zeng was in Samsung Research.

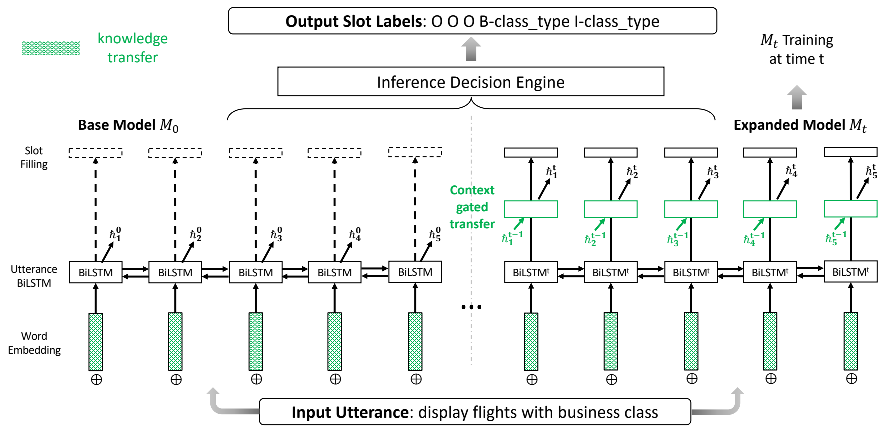

Figure 1: Our Proposed ProgModel Architecture: consists of an expanded component M^t at each batch t with context gated transfer of utterance contexts (Figure 2); and an inference decision engine to derive the label prediction. At each batch t, the last layers of base model and previous components (dotted lines) are only used for inference. Only the output of M^t is used to guide the training.

computer vision (Li and Hoiem, 2016; Lee et al., 2017), yet it still remains open in spoken language understanding systems.

In this paper, we consider a practical setting that a batch[1] of new training data U^t becomes available at each batch t. Our goal is to enable the continual learning capability of a slot filling model such that it can *keep learning new utterances efficiently as well as remember old knowledge without the needs of accessing old training data*. To achieve this, we design a novel *Progressive Slot Filling Model (ProgModel)* that can be gradually expanded at each batch by using a novel context gate for knowledge transfer. Unlike the baseline that repeatedly retrains the same model, ProgModel keeps the previously trained components untouched such that the catastrophic forgetting can be largely avoided. Using the transferred knowledge, each newly expanded component in ProgModel is trained in a progressive manner to achieve better performance with faster training compared with baseline model retraining approaches .

2 Proposed Approach

2.1 Progressive Model (ProgModel)

As the name indicates, the main idea of our proposed ProgModel is to progressively expand the model by transferring existing knowledge from the current model. Thus, ProgModel can continually enhance its capability of understanding user's new expressions without catastrophic forgetting. This is motivated by the recent success of progressive neural networks in various applications (Rusu et al., 2016).

As shown in Figure 1, ProgModel consists of the following components: (1) *Expanded Components:* The current model is expanded via context-gated knowledge transfer to allow only training on a new batch training set U^t at each batch t. (2) *Inference Decision Engine:* When we receive multiple outputs from base model and expanded components, the decision engine is to derive the slot filling label output without additional training.

2.1.1 Expanded Component M^t

At each batch t, a new component M^t (Figure 1 (right)) is expanded on the base model M^0 and previously expanded components $M^1 \dots M^{t-1}$, denoted as $M^{<t}$. The utterance BiLSTM^t is learned from scratch at each batch t such that it can learn the new sentence structures via word sequence correlations. Next, we focus on designing two knowledge transfer mechanisms (green parts in Figure 1) to maximally leverage the previously learned knowledge in $M^{<t}$.

Word Embeddings Transfer: Word embeddings in the newly expanded component M^t are initialized using those in M^0 based on the assumption that M^0 covers most vocabulary. For a new word w, we initialize using GloVe embedding. The embeddings will be fine tuned during training M^t.

Gated Utterance Context Transfer: We then design a novel *context gate* (Figure 2) by introducing lateral connections from most recent component M^{t-1}. *Note that this context gate is crucial to avoid retraining on all data or repeatedly duplicating a large base model every time.* Specifically, the lateral connections transfer the hidden states $\hbar_i^{t-1}$ for each word i from M^{t-1} to the newly expanded component M^t. They are projected to the same dimensional space of $\mathbf{h}_i^t$ via

[1]To avoid the confusion with the widely used timestamp in NLP (mean each word), we use each batch in our paper.

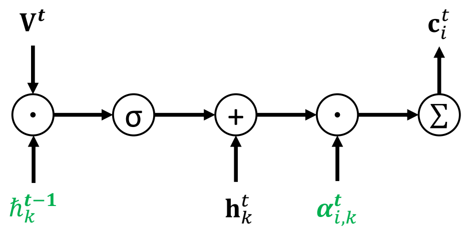

Figure 2: Context Gate for Transfer Utterance Context Knowledge from M^{t-1} (in green)

projection matrix $\mathbf{V}^t$ shared for each word:

$$\boldsymbol{\hbar}_i^t = \mathbf{h}_i^t + \sigma(\mathbf{V}^t \boldsymbol{\hbar}_i^{t-1})$$

where $\sigma(\cdot)$ is the sigmoid function.

Then, the context vector $\mathbf{c}_i^t$ for the i^{th} word in expanded component M^t is given as:

$$\mathbf{c}_i^t = \sum_{k=1}^{n} \alpha_{i,k}^t \boldsymbol{\hbar}_k^t$$

where $\boldsymbol{\alpha}_i^t = softmax(\mathbf{e}_i)$ is fine tuned based on $\boldsymbol{\alpha}_i^0$ trained in base model M^0 as in Att-BiRNN model (Liu and Lane, 2016). We have $\alpha_{i,j}^0 = \frac{\exp(e_{i,j}^0)}{\sum_{k=1}^{T} \exp(e_{i,k}^0)}$ in which $e_{i,k}^0 = g^0(\mathbf{h}_{i-1}^0 \oplus \mathbf{c}_{i-1}^0, \mathbf{h}_k^0)$ is learned from a feed forward neural network g^0. In ProgModel, each model M^t has an independent g^t which is initialized by g^0 in M^0. Thus, $\boldsymbol{\alpha}_i^t$ will be fine tuned from $\boldsymbol{\alpha}_i^0$ during training M^t via updated g^t and $\boldsymbol{\hbar}_i^t$.

2.1.2 Inference Decision Engine

We design the inference decision engine (IDE) as a non-trainable separate component to avoid the potential catastrophic forgetting. Thanks to the capability of knowledge transfer in ProgModel, M^t can already remember quite much previously learned knowledge to give good label prediction in many cases. Thus, we consider two types of decision engines: (1) *t-IDE*: ProgModel using only the output of M^t as decision engine; (2) *c-IDE*: for i^{th} word, it combines all outputs from each component M^t, $\sum_{k=0}^{t} P^k(i) I^k(i)$. $I^k(i)$ is an indicator function which is 1 when i^{th} word is in the vocabulary of M^k and 0 otherwise. The label with maximum probability is selected.

2.2 Progressive Training

The training procedure is progressively conducted at each batch t. The first step is to train the base model M^0 using the loss function $\mathcal{L}^0$:

$$\mathcal{L}^0(\boldsymbol{\theta}^0) \triangleq -\frac{1}{n} \sum_{j=1}^{|S|} \sum_{i=1}^{n} y_j(i) \log P_j^0(i)$$

where $\boldsymbol{\theta}^0$ are the parameters in M^0; $|S|$ is the number of semantic slots in IOB format; and n is the sequence length.

Table 1: Grouping Statistics in All Datasets

Dataset	Domain	Group	Vocab Size	#Slots	Train Size
ATIS	-	1	706	49	3,666
		2	301	24	423
		3	271	18	232
		4	312	29	266
		5	309	33	391
Snips	Add To Playlist	1	2,561	6	1417
		2	802	5	259
		3	764	5	259
	Book Restaurant	1	1,588	14	995
		2	1,044	14	487
		3	990	14	486
	Get Weather	1	1,095	9	805
		2	977	9	592
		3	946	9	591
	Play Music	1	2,333	9	1,546
		2	464	9	210
		3	492	9	210
	Rate Book	1	916	7	822
		2	708	6	539
		3	722	6	538
	Search Creative Work	1	1,353	2	690
		2	1,298	2	631
		3	1,335	2	630
	Search Screening Event	1	863	7	883
		2	687	7	472
		3	668	7	471

At each batch t, we train the expanded component M^t while fixing the parameters $\boldsymbol{\theta}^{<t}$ in previous components. The loss is backpropagated from the output of M^t using the loss function $\mathcal{L}^t$:

$$\mathcal{L}^t(\boldsymbol{\theta}^t, \boldsymbol{\phi}_{t-1}^t) \triangleq -\frac{1}{n} \sum_{j=1}^{|S|} \sum_{i=1}^{n} y_j(i) \log P_j^t(i)$$

where $\boldsymbol{\theta}^t$ and $\boldsymbol{\phi}_{t-1}^t$ are the parameters in model M^t and in the context gate between M^{t-1} and M^t. In both loss functions, $P_j^t(i)$ is output probability of slot j for the i^{th} word from M^t.

3 Experimental Results

3.1 Datasets & Settings

Dataset: We evaluate ProgModel on the following two benchmark datasets:

ATIS (Airline Travel Information Systems) dataset (Hemphill et al., 1990): a widely used dataset in SLU research. The training set contains 4,978 utterances from the ATIS-2 and ATIS-3 corpora, and the testing set contains 893 utterances from the ATIS-3 data sets. There are 127 distinct slot labels. We do not use the intent labels in ATIS.

Snips dataset (Snips, 2017): another NLU dataset custom-intent-engines collected by Snips for model evaluation. It contains 7 domains. In each domain, the training set contains 1,800 to 2,000 utterances and the testing set contains around 100 utterances. Since each domain in

Table 2: Performance (F1 Score) on ATIS Dataset

Approach	Batch 0	1	2	3	4
AttRNN (upper bound)	92.12	92.89	93.04	93.56	95.13
FT-AttRNN		91.85	89.98	91.25	88.03
FT-Lr-AttRNN		91.96	86.46	88.03	86.58
FT-Cp-AttRNN	92.12	92.10	90.06	91.98	89.67
t-ProgModel		92.33	92.43	92.57	92.58
c-ProgModel		**92.40**	**92.64**	**92.71**	**93.91**

Snips contains completely different slots and very few vocabulary are shared between the domains, we evaluate on each domain independently.

We use Amazon Mechanical Turk (MTurk) to split both training set into non-overlapping groups in each dataset (each domain in Snips as a separate dataset). Based on the size of each dataset, we consider 5 groups in ATIS and 3 groups in each domain of Snips dataset. Each turker is given 100 utterances from training set in one dataset; as well as the number of groups G for this dataset. He is asked to put utterances into no more than G groups based on their similarities. At last, we review the grouped utterances from turkers to further combine the similar utterances and derive the final grouping of the whole dataset. For each dataset, we consider the largest group of training set as a base dataset. At each batch t, one of leftover groups in training set is randomly selected to be U^t. Table 1 shows the detailed data statistics.

Competitors: First, only for reference purpose, we consider a *performance upper bound baseline*, i.e., train AttRNN on the all available dataset at each batch t. We compare with the following competitor approaches: (1) *FT-AttRNN* : fine tunes the current model only using new training data U^t at each batch t; (2) *FT-Lr-AttRNN* : fine tunes the current model using an adjusted lower learning rate (we use 0.3 times of base model learning rate which has the best performance) on the new training set U^t; (3) *FT-Cp-AttRNN* : copies the previous model and fine tunes the new copied model on new training data U^t at each batch t. During inference, FT-Cp-AttRNN uses both t-IDE and c-IDE decision engines and reports the one with better performance (F-1 score). We evaluate our ProgModel model with different inference engines: (1) *t-ProgModel*: ProgModel using only output of M^t as decision engine; (2) *c-ProgModel*: ProgModel using combined inference decision engine. All base models M^0 are trained on state-of-the-art AttRNN model (Liu and Lane, 2016). For fair evaluation, we test both ProgModel and competitors on the all standard testing sets.

Table 3: Performance (F1 Score) on Snips Dataset

Domain	Approach	Batch 0	1	2
	AttRNN (upper bound)	79.58	86.74	88.89
	FT-AttRNN		81.23	87.07
Add To Playlist	FT-Lr-AttRNN		78.99	86.61
	FT-Cp-AttRNN	79.58	84.67	87.15
	t-ProgModel		**86.12**	**88.30**
	c-ProgModel		85.51	87.25
	AttRNN (upper bound)	79.49	89.78	90.03
	FT-AttRNN		88.71	88.09
Book Restaurant	FT-Lr-AttRNN		88.57	87.89
	FT-Cp-AttRNN	79.49	89.06	88.14
	t-ProgModel		**89.45**	**89.54**
	c-ProgModel		89.40	89.40
	AttRNN (upper bound)	76.48	91.12	93.56
	FT-AttRNN		89.52	88.93
Get Weather	FT-Lr-AttRNN		89.09	88.56
	FT-Cp-AttRNN	76.48	89.82	90.09
	t-ProgModel		**90.73**	**93.12**
	c-ProgModel		89.92	90.95
	AttRNN (upper bound)	77.48	87.79	89.13
	FT-AttRNN		84.71	84.63
Play Music	FT-Lr-AttRNN		84.53	84.16
	FT-Cp-AttRNN	77.48	84.85	86.10
	t-ProgModel		86.05	87.26
	c-ProgModel		**87.00**	**88.45**
	AttRNN (upper bound)	92.64	98.45	99.07
	FT-AttRNN		96.87	96.83
Rate Book	FT-Lr-AttRNN		96.20	96.86
	FT-Cp-AttRNN	92.64	97.06	97.93
	t-ProgModel		97.50	**98.89**
	c-ProgModel		**98.19**	98.20
	AttRNN (upper bound)	66.32	89.01	89.67
	FT-AttRNN		85.93	85.46
Search Creative Work	FT-Lr-AttRNN		84.69	84.45
	FT-Cp-AttRNN	66.32	87.25	86.36
	t-ProgModel		88.21	88.25
	c-ProgModel		**88.79**	**88.83**
	AttRNN (upper bound)	89.30	95.68	97.34
	FT-AttRNN		93.40	94.53
Search Screening Event	FT-Lr-AttRNN		91.87	93.56
	FT-Cp-AttRNN	89.30	93.81	94.56
	t-ProgModel		**95.01**	**96.90**
	c-ProgModel		93.62	94.31

Training: We implemented ProgModel model using TensorFlow 1.4.0 and conducted the experiments on NVIDIA Tesla M40. At each batch t, we train all models until their convergence. *We observe that ProgModel takes around 10 epochs due to less parameters and transferred knowledge in* M^t *while AttRNN retraining usually needs 100 epochs and various fine tuning competitors need around 30-50 epochs.*

3.2 Main Results

Table 2 and Table 3 show the F1 score of slot filling performance comparison results on ATIS dataset and each domain of Snips dataset. The results show that ProgModel consistently outperforms AttRNN in all domains, where the improvement gain is up to 4.24% in ATIS and 3.03% in Snips. As expected, ProgModel continuously improves performance with more and more new batches of training data, even though it is only trained on new data at each batch. Among all competitors, FT-Cp-AttRNN achieves the closest performance to ProgModel by using much larger model size (shown in Section 3.4). In comparison, both FT-AttRNN and FT-Lr-AttRNN frequently suffer from catastrophic forgetting. The values in pink show that the performance of FT-AttRNN and FT-Cp-AttRNN drops up to 3.82% and 5.38% respectively. As a result, their F1 scores are significantly reduced in the end. At last, we observe that ProgModel is quite close to upper bound performance (Note that this is only for reference rather than comparison since upper bound performance assumes the availability of all training data while ProgModel does not).

3.3 Ablation Study

We further look into each competitor to better understand the advantage of our method. Since FT-AttRNN is only trained on new data, it is oftentimes overwhelmed by new knowledge and results in forgetting the old knowledge. On the other hand, FT-Lr-AttRNN has difficulty to learn new knowledge since it cannot jump out of local optimum due to a small learning rate. As a result, the performance of FT-Lr-AttRNN is even lower than FT-AttRNN most of the time. To make it even worse, the learning rate is very hard to tune at each batch. As we can see, it is non-trivial to achieve both goals, learn new knowledge and remember old knowledge.

FT-Cp-AttRNN performs slightly better than FT-AttRNN and FT-Lr-AttRNN . FT-Cp-AttRNN can be treated as a naive solution to achieve both goals by almost duplicating the model again and again. However, in addition to larger model size and longer training time, it still suffers from efficiently transfer previous knowledge and leads to catastrophic forgetting from time to time.

In comparison, ProgModel outperforms all above competitors since it provides a systematic mechanism to achieve both goals. The training of our designed context gate helps to determine which knowledge to transfer at each batch.

At last, we observe that c-ProgModel performs better than t-ProgModel in ATIS. This has two reasons: First, the utterances in different groups of ATIS are quite structurally similar such that c-IDE further enhances the correct slot label distribution by combing all outputs together. Second, many slots are not included in last group of ATIS dataset. In Snips, t-ProgModel outperforms c-ProgModel in most domains since the utterances in different groups have significant different structures and vocabularies. Thus, inference outputs could be conflicted and their combination leads to worse performance.

Table 4: Average Model Size (MB)

Domain	Batch	Approach FT-AttRNN	 FT-Cp-AttRNN	 ProgModel
ATIS	0	18.0	18.0	18.0
	1		36.9	21.9
	2		56.2	25.5
	3		75.5	29.4
	4		94.8	33.3
Snips	0	25.3	25.3	25.3
	1		50.7	31.8
	2		76.2	38.1

3.4 Model Size Results

Table 4 reports the model size comparison between FT-AttRNN (FT-AttRNN and FT-Lr-AttRNN), FT-Cp-AttRNN and ProgModel (t-ProgModel and c-ProgModel). With more and more training data at each batch, the increase of ProgModel size is significantly slower than that of FT-Cp-AttRNN since the capability of knowledge transfer in ProgModel avoids the full model copy. Thus, our approach also better trades off the model size and performance. The small fluctuation of ProgModel model size expansion is due to the different size of vocabulary in each batch of training utterances. Each expanded model M^t will only keep the embedding of vocabulary in new training data U^t. One may concern that ProgModel will become large over time. In practice, it will be expanded only when the new data grows too large to handle by current model. Moreover, a new base model can be periodically reinitialized to reset the model size.

4 Conclusion

In this paper, we proposed a novel ProgModel model with the capability of efficient continual learning for semantic slot filling in SLU. ProgModel is designed to expand progressively at each batch of new training data with a new context gate for knowledge transfer. The model can be trained progressively without needing to store old training data. We showed that ProgModel need much shorter training time to significantly outperform baseline approaches and close to the upper bound performance.

References

Yue Deng, KaWai Chen, Yilin Shen, and Hongxia Jin. 2019. Adversarial multi-label prediction for spoken and visual signal tagging. In *ICASSP*, pages 3252–3256.

Robert M. French. 1999. Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, 3(4):128 – 135.

Chih-Wen Goo, Guang Gao, Yun-Kai Hsu, Chih-Li Huo, Tsung-Chieh Chen, Keng-Wei Hsu, and Yun-Nung Chen. 2018. Slot-gated modeling for joint slot filling and intent prediction. In *NAACL-HLT*, pages 753–757.

Daniel Guo, Gokhan Tur, Wen-tau Yih, and Geoffrey Zweig. 2014. Joint semantic utterance classification and slot filling with recursive neural networks. In *IEEE SLT*, pages 554–559.

Dilek Hakkani-Tür, Gökhan Tür, Asli Celikyilmaz, Yun-Nung Chen, Jianfeng Gao, Li Deng, and Ye-Yi Wang. 2016. Multi-domain joint semantic frame parsing using bi-directional rnn-lstm. In *INTERSPEECH*, pages 715–719.

Charles T. Hemphill, John J. Godfrey, and George R. Doddington. 1990. The atis spoken language systems pilot corpus. In *Proceedings of the Workshop on Speech and Natural Language*, HLT, pages 96–101.

Gakuto Kurata, Bing Xiang, Bowen Zhou, and Mo Yu. 2016. Leveraging sentence-level information with encoder lstm for semantic slot filling. In *EMNLP*, pages 2077–2083.

Sang-Woo Lee, Jin-Hwa Kim, Jaehyun Jun, Jung-Woo Ha, and Byoung-Tak Zhang. 2017. Overcoming catastrophic forgetting by incremental moment matching. In *NIPS*, pages 4652–4662.

Zhizhong Li and Derek Hoiem. 2016. Learning without forgetting. In *ECCV*, pages 614–629.

Bing Liu and Ian Lane. 2016. Attention-based recurrent neural network models for joint intent detection and slot filling. In *INTERSPEECH*, pages 685–689.

Baolin Peng, Kaisheng Yao, Li Jing, and Kam-Fai Wong. 2015. Recurrent neural networks with external memory for spoken language understanding. In *NLPCC*, pages 25–35.

Avik Ray, Yilin Shen, and Hongxia Jin. 2018. Learning out-of-vocabulary words in intelligent personal agents. In *IJCAI*, pages 4309–4315.

Avik Ray, Yilin Shen, and Hongxia Jin. 2019. Iterative delexicalization for improved spoken language understanding. In *Interspeech*.

Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. 2016. Progressive neural networks. *arXiv preprint arXiv:1606.04671*.

Yilin Shen, Avik Ray, Abhishek Patel, and Hongxia Jin. 2018a. CRUISE: cold-start new skill development via iterative utterance generation. In *ACL, System Demonstrations*, pages 105–110.

Yilin Shen, Yu Wang, Abhishek Patel, and Hongxia Jin. 2019. Sliqa-i: Towards cold-start development of end-to-end spoken language interface for question answering. In *ICASSP*, pages 7195–7199.

Yilin Shen, Xiangyu Zeng, Yu Wang, and Hongxia Jin. 2018b. User information augmented semantic frame parsing using progressive neural networks. In *INTERSPEECH*, pages 3464–3468.

Snips. 2017. https://github.com/snipsco/nlu-benchmark/tree/master/2017-06-custom-intent-engines.

Yu Wang, Yilin Shen, and Hongxia Jin. 2018. A bi-model based RNN semantic frame parsing model for intent detection and slot filling. In *NAACL-HLT*, pages 309–314.

Kaisheng Yao, Baolin Peng, Yu Zhang, Dong Yu, Geoffrey Zweig, and Yangyang Shi. 2014. Spoken language understanding using long short-term memory neural networks. In *SLT*, pages 189–194.

Xiaodong Zhang and Houfeng Wang. 2016. A joint model of intent determination and slot filling for spoken language understanding. In *IJCAI*, pages 2993–2999.