BertKT: A Purely Attention-Based, Bidirectional Deep Learning Architecture for Knowledge Tracing

Yueqi Wang[1] , **Zachary Pardos**[2]
[1]University of Electronic Science and Technology of China
[2]University of California, Berkeley
yqwangrex@gmail.com, pardos@berkeley.edu

Abstract

Knowledge tracing—the task of using a machine to model the cognitive mastery of a student through their interactions with assessment items—has been widely studied and was designed for Intelligent Tutoring Systems (ITS) and other purposes requiring inferences about knowledge components and their relationship to one another. In this work, we present a whole new knowledge tracing architecture called Bert Knowledge Tracing (BertKT) that is purely based on self-attention mechanism. The purely attention-based family of models have significant advantages over the previously widely-used RNN models such as Deep Knowledge Tracing (DKT) model, for their enhanced ability to capture long-range knowledge dependencies and the ease to train. We test BertKT on 2 of the largest educational benchmark datasets for student performance prediction, the result shows a non-trivial increase in cross-validation accuracy of BertKT's prediction compared to DKT and Bayesian Knowledge Tracing (BKT).

1 Introduction

Predicting future response performance on a skill given past response performance on that skill is the central predictive problem in the knowledge tracing. A student's problem-solving history might include but are not limited to the past problem identities (abbreviated as problem ids in the following of this paper) students tried, the corresponding responses (refers to the correctness of responses for the following of this paper), the problems' affiliated problem set ids, related skill ids and problem-difficulty rankings. Among these features, problem ids and responses are most widely used since such information is generally available in most typical computer-aided learning platforms. Given the past problems students solved and the corresponding responses, an Intelligent Tutoring System (ITS) could be designed to model a student's knowledge status and then utilized later by educators to predict students' future performance (correctness of responses) on new problems. As students interact with the coursework, educators could understand quantitatively and thus more accurately the students' knowledge-mastery situation by monitoring the model's predictions of their responses.

One of the most classic models regrading knowledge tracing is the Bayesian Knowledge Tracing (BKT), in which student mastery of hidden knowledge points is modeled as binary variables and the probabilities of these variables are updated by a Hidden Markov Model (HMM), [Corbett and Anderson, 1994]. Such models have a strong disadvantage of insufficient hidden state representations as binary variables, thus reducing the accuracy of the final predictions of responses. Variations of the BKT model were designed to include guessing-and-slipping situations [d Baker *et al.*, 2008] and personalized, prior-knowledge modeling [Yudelson *et al.*, 2013], but none of these improvements overcomes the previously mentioned problems of the BKT family of models. In 2015, the very first knowledge tracing model leveraging deep neural networks called Deep Knowledge Tracing (DKT) was introduced to enable richer hidden-concept learning using an RNN/LSTM neural network [Piech *et al.*, 2015]. However, the original DKT model bears some weaknesses such as inconsistent skill tagging and knowledge state transition[Xiong *et al.*, 2016] [Yeung and Yeung, 2018]. Moreover, the vanilla DKT model uses student skill ids as input data, which are not always available in typical knowledge tracing settings compared with problem level information[Sonkar *et al.*, 2020]. Subsequent works also question the effectiveness of deep-learning oriented KT methods themselves [Khajah *et al.*, 2016] [Ding and Larson, 2019]. Due to the inherently sequential nature of RNN/LSTM, parallelization is disabled and training such models could become computationally inefficient as the records of students' interaction with the course work accumulate and the problem sequences get longer. There are some improvements regarding DKT by including rich features [Zhang *et al.*, 2017] and building hierarchical, feature-item relationships [Wang *et al.*, 2019]. However, to the best of our knowledge, no optimized knowledge tracing model prior to our work had been proposed to solve the aforementioned challenges.

The inventions of Transformer [Vaswani *et al.*, 2017] and BERT [Devlin *et al.*, 2018] models in the field of Natural Language Processing (NLP) make available parallel training and enhanced transfer learning. These 2 achievements have inspired us to introduce the purely attention-based architecture to the educational field. Despite all the advantages of

this architecture, applying a BERT model to solve knowledge tracing tasks poses non-trivial challenges of model abstraction, input-feature integration, pre-training-task designing and single-step cross-validation conducting.

In this work, we introduce a bidirectional, purely attention-based knowledge tracing model called Bert Knowledge Tracing (BertKT), which is well suited to knowledge tracing tasks with parallelization enabled and is more capable of capturing long-distance knowledge dependencies.

The main contributions of this work are:

1. Introducing a purely-attention based, bidirectional deep learning architecture novel to knowledge tracing.
2. A novel way to encode students' learned knowledge contained in problems and responses.
3. An average 9% gain in validation accuracy compared with DKT model and 15.5% for BKT model regarding performance prediction.
4. Enabling transfer learning, parallel training and generalization to more than 1 dataset.

In order to facilitate research in BertKT, we have published core part of our code used for experiments[1].

2 Methodology

In this section, we will talk about series of methodologies needed to design knowledge encoding, attention mechanism and different training tasks.

2.1 Problem Statement and Input Data Description

Problem Statement

Given a student's previously solved problem ids, their responses and new problem ids he or she chooses, the task is to predict the student's responses (correctness) of the chosen problems.

Input Data Description

The Input data is consisted of a number of samples, and each one contains:

1. 100 problems and their ids for 1 student.
2. 100 responses to the 100 problems in 1.. Responses are marked by 1 if correct and 0 otherwise.
3. 100 problem set ids that each one of 100 problems in 1. affiliated to.

The input data structure is visualized in Table 1. The details of how we create these sequences from benchmark datasets are discussed in section 3.

2.2 Knowledge Embedding and Positional Encoding

We consider a single problem id as a unit analogous a word in a sentence for NLP tasks, and we employ word level embeddings for input sequences where each problem id is embedded to a vector of certain length and embedding weights are trained together with the large BertKT model. Same embedding mechanism is applied to responses in section 2.6.

pid0	pid1	...	pid49	pid50	...	pid99
response0	response1	...	response49	response50	...	response99
pset0	pset1	...	pset49	pset50	...	pset99

Table 1: Input data visualization. The input sequences to the BertKT model have 100 problem ids, responses and problem set ids for one student in a row. *pid* stands for *problem id*, *response* for *correctness of the response* marked by 0 or 1 and *pset* for *problem set id*

Since the attention mechanism we use in section 2.3 does not reserve positional information, we apply positional encoding here to record the positions in input sequences. Concretely, we choose the widely-used sine and cosine functions of different frequencies as positional encoding functions shown in equation 1. Sine and cosine are naturally ideal functions to perform positional encoding for the advantage of taking full use of double-precision floating-point numbers' accuracy. For other methods of positional encoding, please refer to Jonas Gehring and Michael Aulis' ICML paper [Gehring *et al.*, 2017].

$$\begin{aligned} PE_{(pos,2i)} &= sin(pos/10000^{2i/d_{model}}) \\ PE_{(pos,2i+1)} &= cos(pos/10000^{2i/d_{model}}) \end{aligned} \tag{1}$$

In function 1, *PE* stands for Positional Encoding, *pos* means the position a problem id occupies in a sequence and *i* means the i^{th} dimension of the positional encoding. *d_model* stands for the dimension of a particular neural network layer and we set it to be equal to the dimension of the embedding layer. After the positional encoding, the encoded positional vector representation of problem ids or responses are linearly added to their embeddings elementwise. The illustration of this process could be viewed at the bottom part of Figure 2.

2.3 BertKT Encoder

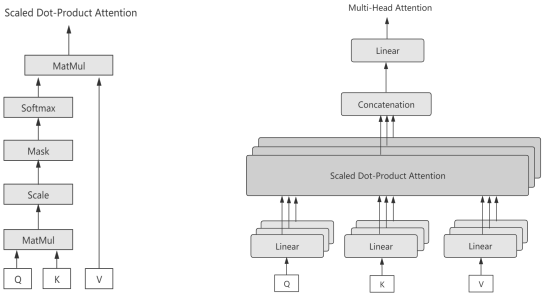
In this section, we will introduce how we create the core part of a BertKT model—the BertKT Encoder, including the self-attention mechanism and other components. We basically utilize a transformer neural network structure which has been common in use, so we refer readers to [Vaswani *et al.*, 2017] for the advantages of such neural networks.

Attention Mechanism 1: Scaled Dot-Product Attention

Purely attention-based mechanism is the key attribute of BertKT model. First of all, attention mechanism decides what proportion of the problems to attend to when predicting the response for a target problem. Also, pure-attention structure accelerates training process credited to parallelization. Here we choose the scaled dot-product attention in equation 2 to compute unit attention given queries, keys and values. In our settings, all queries, keys and values are created from different linear mappings of the same problem id/response embedding since we need to compute self-attention.

$$Attention(Q, K, V) = softmax_k(\frac{QK^T}{\sqrt{d_k}})V \tag{2}$$

[1] https://drive.google.com/drive/folders/1fp2oP9RgbnJl8A7nPupT87ZzItp9jsvf?usp=sharing



(a) Scaled Dot-Product Attention (b) Multi-head Attention

Figure 1: Self-Attention computation flow. Figure 1a shows the calculation steps of scaled-dot product attention defined by equation 2 where Q, K, V means Query, Key and Value respectively. *MatMul* stands for matrix multiplication and *Scale* corresponds to the scaling factor $\sqrt{d_k}$ in equation 2; For Figure 1b, Q, K and V are partitioned into small contiguous sequences and linear-mapped into different sub-spaces where scaled-dot product attentions are computed separately followed by concatenation.

In equation 2, Q, K, V stand for Query, Key, and Value respectively, and d_k is the dimension of the self-attention layer. Scaled dot-product attention has a series of advantages over other attention mechanisms. "Dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code" and scaling is applied to reduce variance between samples [Vaswani *et al.*, 2017]. In our perspective, another advantage of dot-product attention is that the linear weights are trained together with the entire model, enabling better adaptation to subsequent layers and meanwhile attaining higher accuracy. The detailed theory of scaled dot-product attention is discussed in the original transformer paper [Vaswani *et al.*, 2017]. Additionally, as for the model not to attend to future problems or see itself when computing attention weights, *masks* are applied to block out such information. Figure 1a shows the computation steps of scaled dot-product attention.

Attention Mechanism 2: Multi-Head Attention
In order for the model to learn the attention at full-scale, we create partitions of the input sequences and map them into different sub-spaces. Attention of each partition is then computed separately followed by concatenation. A wrapping-up linear layer is added after the concatenation layer before the module outputs multi-head attention values. "Multi-head attention allows the model to jointly attend to information from different representation" [Vaswani *et al.*, 2017]. The computation of Multi-head Attention could be viewed in Figure 1b.

BertKT Encoder Assembling
Figure 2 shows the structure of BertKT Encoder. The input problem id sequences are embedded using a problem embedding layer and positional encoding is added afterwards, if the Encoder is used for the fine-tuning task, an additional response embedding layer is utilized to process response sequences. Then, Multi-head attention layers are employed to compute self-attention followed by residual connection summation and layer normalization. The normalized layer goes through a feed-forward neural network followed by residual connection and another layer normalization process. Lastly, a fully connected Linear Layer with sigmoid activation is used for wrapping up. The outputs of BertKT Encoder are vectors where the entries contain probabilities of target problem responses or other target values depending on the task.

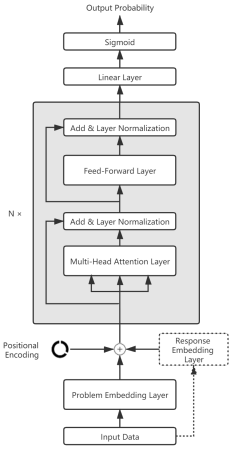


Figure 2: BertKT Encoder Structure Visualization. The shaded part of the legend is the core part of BertKT Encoder and in order for readers to see the interactions of embeddings and positional encoding, we plot out a redundant embedding computation step below the shaded part identical to that of Figure 4.

2.4 Pre-training Task1—Masked Knowledge Model

One crucial attribute of an effective knowledge tracing model is being able to learn rich hidden knowledge status as well as their relationships. In order for our model to master such information, we design a straightforward but rather "difficult" pre-training task and we call it the Masked Knowledge Model (MKM). The idea of this task is inspired by the Masked Language Model (MLM) task designed by Google AI Language Team [Devlin *et al.*, 2018]. In MKM, we would like to randomly mask a proportion of the problem ids of each student, and train the model to predict these ids under the mask. The MKM is considered difficult since we have in total the unique problem ids in the order of 10,000 that are all potential correct predictions. Another reason is that relations between problem ids may not be as strong as that between natural language words. However, we will show later in section 3 that despite such challenges, MKM task still aids greatly the fine-tuning task's training speed and pushes the final cross-validation accuracy to over 80%.

Concretely, we mask 15% of problem ids in each sample and replace them with a 'Mask' token, which is trivially set to the total number of unique problems in the dataset plus 3. One masked sample is shown in Table 2.

Figure 3 shows the architecture and data flow for the MKM task and the LPP task introduced in section 2.5.

Student	CLS	pid0	pid1	...	pid49	SEP	pid50	pid51	...	pid99
Student X	16231	24	Mask	...	7804	16232	8023	Mask	...	11435

Table 2: A sample of masked sequences. The *CLS* and *SEP* tokens are introduced in section 2.5 discussing the second pre-training task, readers could skip them when reading section 2.4.

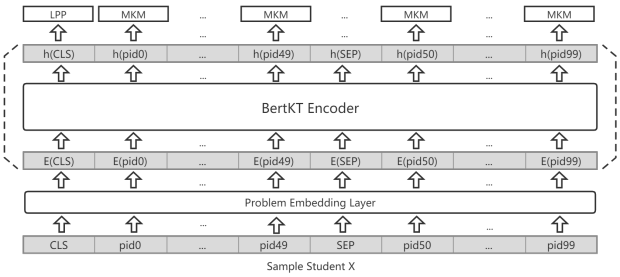


Figure 3: Model architecture and data flow for pre-training tasks—MKM and LPP. The *CLS* and *SEP* tokens are introduced in section 2.5, reader could skip them when reading section 2.4. An input sample of a random student X containing *CLS*, *SEP* tokens and problem ids (pid*i*) are first processed by the problem embedding layer and mapped to vectorized representations called problem embeddings (E(pid*i*)). Then a BertKT Encoder shown in Figure 2 is utilized to computer positional encodings as well as self-attention and outputs the final hidden vectors (h(pid*i*)). The final hidden vector of the *CLS* token contains the probabilities of predictions to the LPP task introduced in section 2.5 and the final hidden vectors corresponding to the masked 15% of problem ids contain the probabilities for the MKM task. The square-bracketed part of the architecture used for pre-training remains the same as that for the fine-tuning shown in Figure 4.

2.5 Pre-training Task2—Learning Pattern Prediction

The idea of pre-training the model to learn sequence level relationships increases drastically the training speed of fine-tuning and improves prediction accuracy simultaneously [Devlin *et al.*, 2018]. In order for the BertKT model to learn sequence level connections, we would like to design a sequence level representation for each sample, mask it with a special token, and train the model to predict it. We introduce this pre-training task as the Learning Pattern Prediction (LPP) task and we design it in the following way.

As shown in Table 2, we have 100 problems for one student in a sample sequence. Each sequence is firstly partitioned into to contiguous parts evenly separated to 50/50 by a *SEP* token. Then, we compute the arithmetic average of the problem set ids of the first 50 problems and do the same to the second 50. Following that, we compare these 2 average values and concatenate integer 1 to the head of this sample if the first value is lower, and 0 otherwise. Finally, we mask the concatenated value with a *CLS* token that stands for classification and train the model to predict it.

Intuitively, if a particular sample sequence has the value 1 under the *CLS* mask, it means that the first 50 problems the student tries bear a lower average problem set id than that of the second 50. In other words, we know this student tries overall more advanced problems marked by greater problem set ids in the second half of their learning process. If the real value of the *CLS* is 0, the student may be circling around problems with their ids close in value, reviewing plenty of basic problems in the second half of their learning process or learning in reverse directions. All these 3 situations could produce a lower average problem set id for the second half of the sequence and are usually considered unscientific learning patterns for most students. After being trained by the LPP task, the BertKT model could distinguish between a scientific or unscientific learning pattern represented by the real value of the CLS token. The architecture and data flow for the LPP task is shown in Figure 3.

After being processed by the 2 pre-training tasks, we assume the model has already acquired certain understandings of the hidden relationships between different knowledge status, both in *fine-grained* problem level and *coarser* problem set level. Later in section 3, we will show quantitatively how the pre-training tasks enhance the training speed and validation accuracy of fine-tuning.

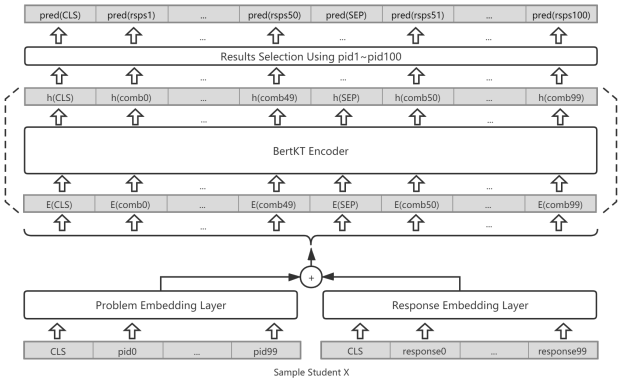


Figure 4: Model architecture and data flow for Fine-tuning. The *prior problem ids*(pid*i*) and *prior responses*(response*i*) are embedded using 2 separate embedding layers followed by element-wise summation to form combined vector representations E(comb*i*). Then, the combined embeddings are processed by the BertKT Encoder shown in Figure 2 which outputs the final hidden layer containing response probabilities (pred_rsps*i*). Responses to subsequent problems are selected by the *Result Selection Layer* using post problem ids. The square-bracketed part of the model is identical to that of Figure 3 saved during pre-training.

2.6 Fine-tuning—Predicting Student Responses

Knowing student prior solved problem ids and their corresponding responses, the goal of the fine-tuning task is to predict responses given new problem ids. In this paper, we use the problems No.0 to 99 as well as their responses as the prior information and treat problems No.1 to 100 as the posterior or the target problems to do response predictions on. A recap of the input sequence visualization is shown in Table 1.

In the fine-tuning task, we use a separate embedding layer to encode the student responses shown in line 2 of Table 1 where the value of response*i* is binary. Then, the embeddings of problem ids and responses are summed elementwise to

form combined embeddings followed by a BertKT Encoder identical to that in the pre-training part, except that labels are changed to ground-truth values of responses. Since each of the final hidden vectors contains probabilities that responses of all problems to be correct given the prior, a *Result Selection Layer* is added to select the target problem's response indexed by post/target problem id. This is implemented by creating a selection matrix containing post problem ids followed by multiplying it to the final hidden vectors. To point out, *pid*s are numbered in an order consistent with the student's problem-solving order and the value of *pid1* could be problem 2 and *pid2* could be problem 5 actually, depending on which problem a student tried chronologically. Also, the CLS and SEP tokens retained are simply place holders in the fine-tuning task with no actual meanings. Figure 4 shows the architecture and data flow for fine-tuning.

Fine-tuning Variation—Switch to Skill Ids

For experimental purposes, we change the input problem id sequences to their related skill id sequences and compare the results to that of DKT.

We discover that using the skill ids as input could increase the validation accuracy by 11% for DKT model and 2% for BertKT model. Given the high validation accuracy of BertKT discussed in section 3, the 2% increase in accuracy does not affect the model's overall performance. Oppositely, it means that BertKT could have already learned the hidden knowledge state common to both skill id and problem id when using the latter as input. We include this experiment to keep consistency with the previous work of DKT [Piech *et al.*, 2015] and to argue that despite in their work, AUC of the prediction could reach over 80%, this result might be partially guaranteed by using the more accurate skill id information. In our experiment by switching to problem ids as input, cross-validation accuracy is far lower for the DKT compared with using skill ids. Also, a recent work introducing question-centric DKT [Sonkar *et al.*, 2020] shows the AUC of question-centric DKT on ASSISTment 2009 dataset is significantly smaller than the original DKT due to overfitting, which implies that skill level information is far more representative to fit a complex model than the same amount of question level information.

3 Experiment

In this section, we will analyze the results for BertKT model and compare it to 2 of the most popular knowledge tracing models—Bayesian Knowledge Tracing (BKT) and Deep Knowledge Tracing (DKT). Our model is trained and tested on the 2009-2010 ASSISTment dataset[2] [Feng *et al.*, 2009] and KDD Cup 2010 Algebra I 2005-2006 development dataset[3].

- For ASSISTment 2009-2010 dataset, we select students with more than 100 problems solved to form sequences. Among those students, we extract chronologically their first 100:
  1. "Problem_id"s.
  2. "Correct"s of the problems in 1. as responses.
  3. Problems' corresponding "base_sequence_id"s as problem set ids.
- For KDD Cup 2010 Algebra I 2005-2006 Development dataset, we select students with more than 100 problems solved to form sequences. Among those students, we extract chronologically their first 100:
  1. "Problem Name"s, which are later numbered as problem ids.
  2. "Correct First Attempt"s as responses.
  3. Problems' corresponding "Problem Hierarchy"s as problem set ids.

Each of the 2 datasets is split into 0.7 to 0.3 in the number of samples for training and validation respectively. Some important hyperparameters we choose for the BertKT model are listed as follows:

1. The number of BertKT Encoder core layers N = 4
2. Attention layer dimension = 64
3. Feed-forward layer dimension = 256
4. The number of heads for multi-head attention = 8

For the BKT model implementation, we have used and made modifications to the original pyBKT implementation by Xu, Johnson and Pardos[4] [Xu *et al.*, 2015].

All models are trained on a single NVIDIA Tesla P100 cloud GPU, and the training time for BertKT model on either of 2 aforementioned datasets is less than 10 minutes. For the evaluation metric, we choose binary validation accuracy and all models are validated using a 0.3/0.7 train-test hold out validation method. The results we show in the following subsections are the best of all our results produced.

3.1 Pre-training and Fine-tuning Analysis

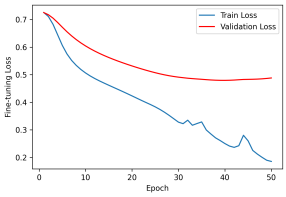
In this subsection, we compare the training speed and validation accuracy of the BertKT fine-tuning process with or without the pre-training. When computing the loss and accuracy, we use Keras *BinaryLoss* and *BinaryAccuracy* metrics with a threshold of 0.5, which means predictions with probabilities greater and equal to 0.5 are set to 1 and 0 otherwise. We train all pre-training and fine-tuning tasks exhaustively until convergence.

Figure 5 shows the fine-tuning loss and accuracy without pre-training. Figure 5b demonstrates that the validation accuracy finally converges to nearly 0.79 after about 40 epochs. As comparison, Figure 6b shows that when pre-training is incorporated before fine-tuning, the validation accuracy reaches over 0.8 after fine-tuning for 1-2 epochs and converges to 0.81 after roughly 15 epochs (Average accuracy over all training

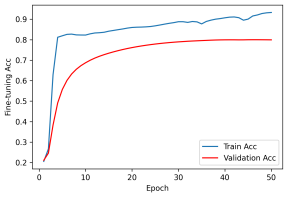
[2]https://sites.google.com/site/assistmentsdata/home/assistment-2009-2010-data

[3]Stamper, J., Niculescu-Mizil, A., Ritter, S., Gordon, G.J., & Koedinger, K.R. (2010). Algebra I 2005-2006. Development data set from KDD Cup 2010 Educational Data Mining Challenge. Find it at http://pslcdatashop.web.cmu.edu/KDDCup/downloads.jsp

[4]https://github.com/CAHLR/pyBKT

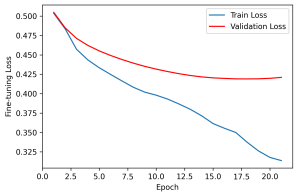


(a) Fine-tuning loss I

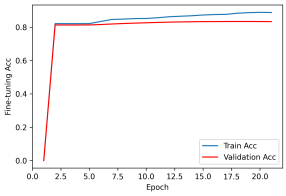


(b) Fine-tuning accuracy I

Figure 5: Fine-tuning loss and accuracy *without* pre-training for one of the training processes. The validation loss and accuracy converge basically after 40 epochs without pre-training. Validation accuracy eventually converges to 0.79.



(a) Fine-tuning loss II



(b) Fine-tuning accuracy II

Figure 6: Fine-tuning loss and accuracy *after* pre-training for one of the training processes. The validation loss and accuracy converge basically after 15 epochs with pre-training enabled and validation accuracy reaches over 0.8 after fine-tuning for 1 epoch and finally converges to 0.81. Compared to Figure 5, the number of training epochs are reduced from roughly 40 to 15 and validation accuracy is increased by 2%.

processes is shown in table 3). For one random training process, we could see there is a 2% increase in validation accuracy when pre-training participates and the number of training epochs is significantly reduced from roughly 40 to 15, saving efforts of training from scratch the next time we train the model for newly designed tasks. Also as mentioned before, the data for pre-training in a self-supervised way is relatively cheap to collect compared with that for the fine-tuning task.

The pre-training tasks usually have a much lower accuracy ranging from 0.3-0.5. The main reason we give for this is the high difficulty of the 2 pre-training tasks—MKM and LPP. Empirically, predicting masked problem ids with a vocabulary size of over 10,000 is considered difficult without additional information provided. Similar reasons hold for the LPP task. However, these 2 "merciless" pre-training tasks "force" our model to learn effective knowledge relationships in problem ids and problem set ids. Despite that such hidden features could not be tested explicitly, our results show a promising improvement of training speed and validation accuracy governed by MKM and LPP.

3.2 A Comparison between BertKT, DKT and BKT

For all knowledge tracing models presented here, we train exhaustively until the validation loss and accuracy converge. Then, we take the average value of loss/accuracy for each model after running each for 10 times.

	Train Acc			Val Acc		
Dataset	BKT	DKT	BertKT	BKT	DKT	BertKT
ASSISTment	0.75	0.96	0.89	0.65	0.71	0.83
KDD Cup	0.72	0.82	0.78	0.63	0.70	0.76

Table 3: A Comparison between BertKT, BKT and DKT with problem ids as input. For the ASSISTment 2009-2010 dataset, the validation accuracy of BertKT reaches 0.83, which is 12% higher than that of DKT and 18% higher than BKT; For KDD Cup 2010 Algebra I 2005-2006 dataset, the BertKT's validation accuracy reaches 0.76, gaining a 6% increase than the DKT and 13% than the BKT.

	ASSISTment		KDD Cup	
Metrics	DKT	BertKT	DKT	BertKT
Validation Acc	0.82	0.85	0.75	0.79

Table 4: A Comparison between BertKT and DKT with skill ids as input. BertKT model's validation accuracy is 3% higher than DKT for the ASSISTment dataset and 4% for the KDD Cup dataset; Averaging on the 2 datasets, using skill ids increases validation accuracy of DKT by 8% and only 2.5% for the BertKT due to its robustness compared to Table 3.

Table 3 shows a performance comparison between BKT, DKT and BertKT model. For the ASSISTment 2009-2010 dataset, the BertKT's validation accuracy reaches 0.83 and shows a 12% increase compared to the DKT model and 18% compared to BKT. For the KDD Cup dataset, the increases are 6% and 13% respectively. Overall, the average increase of BertKT's validation accuracy on 2 benchmarks over DKT is 9% and 15.5% over BKT. We credit this improvement to the design of 2 pre-training tasks and the purely attention-based deep learning architecture of the BertKT model with a stronger capability to learn long-distance dependencies.

Experiment with skill id

In the previous work introducing the DKT model [Piech *et al.*, 2015], the authors experimented on skill id as input, which is far more accurate than the problem id we use in this work. We argue that the skill id is not always available to a standard Intelligent Tutoring System (ITS) and problem id is used far more frequently in practice. However, using skill id still serves as a great research option and increases the prediction accuracy when it is available. In order to maintain consistency with the previous works, we also experiment on 2009-2010 ASSISTment dataset replacing problem id with skill id for DKT and BertKT. Other settings remain unchanged.

Table 3 and Table 4 show that using the skill id brings on average an 8% increase of validation accuracy to the DKT and 2.5% to BertKT than using problem ids. We credit this growth to the accuracy of skill id information. Empirically, skill id is a more accurate metrics to represent knowledge status and it could be tested on more than 1 problem. So, the entire skill id set (in the order of 100 in size) is much smaller than the problem id set (in the order of 10,000) and skills are more representative. Thus, when the skill ids are given explicitly as the input data, the student knowledge pattern is clearer and stronger dependencies could be learned, result-

ing in higher prediction accuracy. Given the sharp increase of DKT's validation accuracy and the robustness of BertKT when switching to skill ids, we believe that BertKT is better generalized to model student responses based on problem ids that are not as easy enough for DKT to learn well.

4 Conclusion and Future Works

In previous sections, we have shown the improvement in training speed and accuracy by the BertKT Architecture. Moreover, saved models and knowledge representation vectors could be employed in the future for transfer learning, saving efforts of training from scratch. We hope BertKT could serve as a basement and a bridge for varying educational data mining tasks, since the trained model could be saved and updated by more pre-training tasks designed in subsequent works. We also hope that cross-platform knowledge encoding and management could be designed to enhance uniform use of our model. By making efforts in the such aspects, modeling students' knowledge and predicting their performance could become ever faster and easier than before, generating greater educational impact on a wider range of fields.

Acknowledgments

We greatly appreciate insights and advice received from *INFO C260F: Machine Learning in Education* at UC Berkeley. Also, we are grateful to the GSI of this class, Rajvardhan Oak, for technical supports.

References

[Corbett and Anderson, 1994] Albert T Corbett and John R Anderson. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User modeling and user-adapted interaction*, 4(4):253–278, 1994.

[d Baker *et al.*, 2008] Ryan SJ d Baker, Albert T Corbett, and Vincent Aleven. More accurate student modeling through contextual estimation of slip and guess probabilities in bayesian knowledge tracing. In *International conference on intelligent tutoring systems*, pages 406–415. Springer, 2008.

[Devlin *et al.*, 2018] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[Ding and Larson, 2019] X Ding and E Larson. Why deep knowledge tracing has less depth than anticipated. In *Proceedings of the 12th International Conference on Educational Data Mining*, 2019.

[Feng *et al.*, 2009] Mingyu Feng, Neil Heffernan, and Kenneth Koedinger. Addressing the assessment challenge with an online system that tutors as it assesses. *User Modeling and User-Adapted Interaction*, 19(3):243–266, 2009.

[Gehring *et al.*, 2017] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1243–1252. JMLR. org, 2017.

[Khajah *et al.*, 2016] Mohammad Khajah, Robert V Lindsey, and Michael C Mozer. How deep is knowledge tracing? In *Proceedings of the 9th International Conference on Educational Data Mining*, pages 94–101, 2016.

[Piech *et al.*, 2015] Chris Piech, Jonathan Bassen, Jonathan Huang, Surya Ganguli, Mehran Sahami, Leonidas J Guibas, and Jascha Sohl-Dickstein. Deep knowledge tracing. In *Advances in neural information processing systems*, pages 505–513, 2015.

[Sonkar *et al.*, 2020] Shashank Sonkar, Andrew E Waters, Andrew S Lan, Phillip J Grimaldi, and Richard G Baraniuk. qdkt: Question-centric deep knowledge tracing. In *Proceedings of the 13th International Conference on Educational Data Mining*, pages 677–681, 2020.

[Vaswani *et al.*, 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[Wang *et al.*, 2019] Tianqi Wang, Fenglong Ma, and Jing Gao. Deep hierarchical knowledge tracing. In *Proceedings of the 12th International Conference on Educational Data Mining*, 2019.

[Xiong *et al.*, 2016] Xiaolu Xiong, Siyuan Zhao, Eric G Van Inwegen, and Joseph E Beck. Going deeper with deep knowledge tracing. In *Proceedings of the 9th International Conference on Educational Data Mining*, 2016.

[Xu *et al.*, 2015] Y. Xu, M. J Johnson, and Z. A. Pardos. Scaling cognitive modeling to massive open environments. In *Workshop on Machine Learning for Education at the 32nd International Conference on Machine Learning (ICML). Lille, France.*, 2015.

[Yeung and Yeung, 2018] Chun-Kit Yeung and Dit-Yan Yeung. Addressing two problems in deep knowledge tracing via prediction-consistent regularization. In *Proceedings of the Fifth (2018) ACM Conference on Learning @ Scale*, pages 1–10, 2018.

[Yudelson *et al.*, 2013] Michael V Yudelson, Kenneth R Koedinger, and Geoffrey J Gordon. Individualized bayesian knowledge tracing models. In *International conference on artificial intelligence in education*, pages 171–180. Springer, 2013.

[Zhang *et al.*, 2017] Liang Zhang, Xiaolu Xiong, Siyuan Zhao, Anthony Botelho, and Neil T Heffernan. Incorporating rich features into deep knowledge tracing. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, pages 169–172, 2017.