Distributed Memory Networks for Knowledge Tracing

Nathan Frederick Elazar

A subthesis submitted in partial fulfillment of the degree of
Bachelor of Advanced Computing (Honours) at
The Department of Computer Science
Australian National University

May 2018

Typeset in Palatino by TEX and LATEX 2ε.

Except where otherwise indicated, this thesis is my own original work.

Nathan Frederick Elazar
24 May 2018

Acknowledgements

I would like to thank my supervisor, Dr. Qing Wang, for the amazing support she has given me throughout this project. From guiding my research efforts, to editing my draft thesis, the amount of times she has helped me out more times than I can count.

Abstract

Knowledge tracing is the task of modelling how a student acquires knowledge of various topics over time as they interact with a sequence of learning exercises. Knowledge tracing models are commonly employed by computer aided learning environments to recommend exercises to students in real time, so as to help students practice problems that are the most beneficial to their learning. Recently, deep neural networks have been used to build knowledge tracing models. The most successful such model and the current state of the art for knowledge tracing is the Dynamic Key-Value Memory Network (DKVMN) model, which has been shown to be able to predict student performance better than all previous knowledge tracing models. In this work, we show that DKVMN is incapable of modelling a student which acquires knowledge at a varying rate, and develop a new deep neural network architecture for knowledge tracing called Distributed Memory Networks (DMN) which does not suffer from this limitation. We then propose an extension to our model, called Accumulative Distributed Memory Networks (ADMN), which allows it to exploit the repetitive nature of common computer aided learning environments to learn longer term dependencies in input data. We evaluate our models on 4 public datasets of student exercise solving logs; ASSISTments2009, ASSISTments2015, STATICS2011, JUNYI2015, and find that both of our models significantly outperform the previous state of the art on all of them, with ADMN achieving the best performance overall. Furthermore, we show how our models can be used to automatically discover dependencies between exercises while being trained to predict student performance in an end-to-end manner, something which no previous KT model has been able to do.

Contents

Chapter 1

Introduction

Education is one of the most important resources a society can provide for its citizens, and despite the modern technological innovations revolutionising the way in which all kinds of media are consumed, the education sector remains largely unchanged [Collins and Halverson 2018]. The availability of information provided by the Internet, along with data drive machine learning algorithms have given rise to many new online platforms which allow people to find and consume media. Netflix, an extremely popular movie streaming platform, has become an integral part of film culture [Hallinan and Striphas 2016]. Youtube, a video sharing website which sees thousands of users uploading video content daily, has allowed for individuals to create their own media and communities, in a way unlike historically centralized television [Hartley 2008] [Sajib et al. 2018]. Crucial to the success of such online media sharing platforms are the recommendation systems that power them. Recommendation systems identify trends and patterns in user consumption data to recommend new content to each user, personally tailored to their interests. These personalized recommendations allow each user to find content that is specifically relevant or interesting to them, among the vast amounts of media that would otherwise be impossible to navigate [Lu et al. 2015]. In comparison to this, education is still remarkably traditional: every student in a class is given the same problems to work on in the same order and taught in the same way. However, it is known from research in the field of behavioural psychology that different students may have significantly different learning preferences, from the way in which content is explained, to the topics that are covered [Dunn et al. 2002]. It then seems reasonable that the student experience could benefit from the use of recommendation systems to distribute personalized learning plans and educational media.

1.1 Intelligent Tutoring Systems

The concept of using computer programs to tutor students is not new, indeed there has been significant research into developing AI programs to teach students dating back as early as 1980 [Nwana 1990]. These intelligent tutoring systems, although there have been many different ones created, commonly share the same general characteristics. An intelligent tutoring system has access to a pool of practice exercises, usually

provided by human experts, from a wide range of different topics. The system will present exercises to a student, and record the answers given by the student. By analyzing a student's history, along with the behaviour of other students, the system will attempt to select exercises which help the student learn the most: usually this means the exercise should be not too difficult and not too easy. Depending on the sophistication of the intelligent tutoring system it may offer other features, such as providing hints to students when prompted [Nwana 1990]. Recently some new online

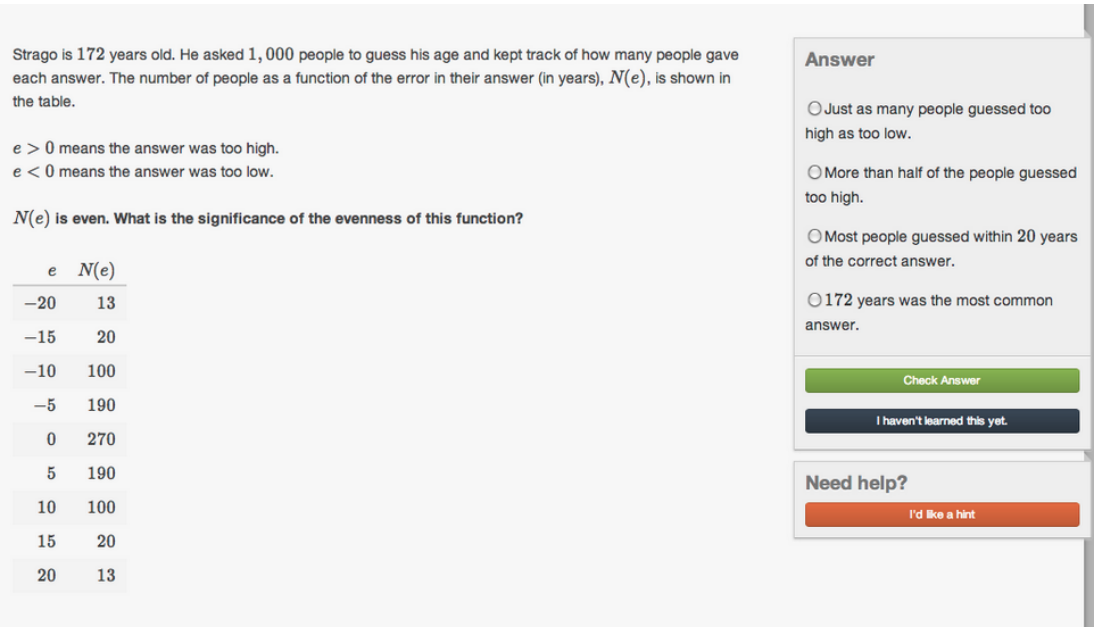

Figure 1.1: Example of a high-school mathematics exercise provided by Kahn Academy

education platforms have appeared which have become moderately popular. Most notably, Khan Academy[1] is a website that allows students to practice problem solving exercises across a wide range of topics, including mathematics, economics, computer programming and more. As of 2018, Khan Academy sees millions of users monthly. Despite Khan Academy's undoubted usefulness, it and other online platforms like it do not provide in-depth feedback or detailed learning plans tailored to individual students [Thompson 2011].

1.2 Knowledge Tracing

For the purpose of building useful computer aided learning environments, it is crucial to understand how students acquire new knowledge. This motivates the study of knowledge tracing, the aim of which is to model a student's mastery or knowledge of various topics, and how this knowledge changes over time [Corbett and Anderson 1994]. Reliable knowledge tracing models have a variety of useful applications

[1] https://www.khanacademy.org/

in improving the effectiveness of autonomous tutoring systems. It is common for a knowledge tracing model to be used to decide which exercise a student should be given next: by evaluating the the student's would-be knowledge after they attempt any given exercise, exercises can be ranked by how much the student's knowledge is expected to improve after attempting them. This implicitly means understanding when a student has sufficiently mastered an exercise and at which point they will no longer benefit from practicing it. Similarly, knowledge tracing models can be used to discover which exercises are the most beneficial for learning overall, by predicting which exercises generally lead to the largest improvements in student performance. This can be used by human experts to help design new exercises for students [Piech 2016]. The first widely popular knowledge tracing model was a simple probabilistic model called Bayesian Knowledge Tracing (BKT) [Corbett and Anderson 1994]. BKT was successfully applied to discover insights into designing problem solving exercises, and providing personalized exercise recommendations for students in real time. Since its inception there have been many advances in the field of knowledge tracing, with the most successful models employing powerful machine learning techniques such as deep neural networks. The current state of the art model is a neural network architectures specifically designed for KT, called Dynamic Key-Value Memory Networks (DKVMN). However, DKVMN does not achieve significantly higher performance than more general purpose deep neural network models [Zhang et al. 2017].

1.3 Research Objectives

The goal of this research project is to investigate the capabilities of knowledge tracing models to predict students' learning behaviour, with the intention of using these models to provide personalized recommendations for topics or exercises for students to practice. To this end, we will

- Develop a method for knowledge tracing that can be used to accurately predict how a student will perform on exercises given their history of interactions.
- Investigate how the method can be used to automatically discover related concepts and dependencies among exercises.
- Conduct experiments on real world data collected from operational intelligent tutoring systems to valid our methods.

1.4 Contributions

The main contributions of this work are as follows:

- We show that the current current state-of-the-art knowledge tracing model is not capable of modelling knowledge growth at a varying rate, and propose a new model, called Distributed Memory Networks, which uses distributed writing to

an external memory matrix to overcome this limitation. We evaluate our model on 4 public knowledge tracing datasets and find it significantly outperforms the previous state of the art.

- We provide a further extension to our model, called Accumulative Distributed Memory Networks, to exploit the repetitive nature of exercises in knowledge tracing and show that this improves the models predictions.
- We show how our models automatically discover groups of similar exercises.
- We show how our models can be altered to read and write to external memory separately, and show that this allows them to automatically discover dependencies between exercises. As far as we know, no previous knowledge tracing models are capable of this, and instead exercise dependencies are commonly annotated by human experts.

Chapter 2

Related Works

In this chapter we review the related works from the fields of Knowledge Tracing (KT) and Recommendation Systems (RS) and how these ideas have led to the modern state of the art in real-time interactive tutoring systems.

2.1 Knowledge Tracing

Knowledge Tracing (KT) is the task of modelling a student's knowledge growth over time, based on their interactions with an Intelligent Tutoring System (ITS). It is crucial for an ITS to be able to successfully model a student's learning progress in order to facilitate several intelligent tutoring features, such as exercise recommendation and providing personalized feedback. KT can also be used to analyze ITS data to inform decisions of human education experts. When it was first proposed as a problem, a probabilistic model was used. Recently, neural networks have been demonstrated to perform better, however the use of neural networks makes the model opaque and uninterpretable for humans.

2.1.1 Bayesian Knowledge Tracing

Bayesian Knowledge Tracing (BKT) was one of the earliest attempts at KT, it was first introduced by [Corbett and Anderson 1994]. BKT assumes a simple probabilistic model for student learning: a student's current knowledge state is represented by a set of binary variables, one for each exercise. Each binary variable can either be 1, representing the student has previously mastered the exercise, or else 0 representing the student has yet to master this exercise. The probability that a student successfully answers the next exercise is computed simply based on whether or not the student has mastered the exercise. The parameters of the model can be inferred from a dataset using a standard optimization algorithm, and expectation maximization is used in the original paper. Since its initial conception many improvements have been added to the simple BKT model, including personalized parameters for each student, which substantially improve the models prediction accuracy [Yudelson et al. 2013].

2.1.2 Deep Knowledge Tracing

Following the recent success of deep neural networks in the field of natural language processing, [Piech et al. 2015] proposed to use a recurrent neural network model for the task of KT, called Deep Knowledge Tracing (DKT). They showed that their DKT model did in-fact significantly outperform BKT. However, later papers using the more sophisticated variants of BKT with personalized student parameters showed that the performance improvement from DKT was marginal at best [Xiong et al. 2016]. As a neural network model, DKT also has the advantage that it can process any vector input, unlike BKT which only works for discrete exercise tags. This makes it easy to incorporate other information about a student's interactions into the model. For example, [Liang et al. 2017] have used other features, such as the time it takes a student to answer and the number of times a student has attempted the exercise before, along with the correctness of the student's response. They found that this additional information significantly improved the DKT model's predictions. One down-side to the recurrent neural network model is that the student's knowledge is represented by a single vector: a student's mastery of every different topic is compacted into a single dense vector, which is completely uninterpretable to human experts. In contrast, BKT explicitly uses a set of variable to indicate a student's mastery of each exercise.

2.1.3 Dynamic Key-Value Memory Networks

In order to create a neural network model that explicitly models a student's knowledge across different topics separately, [Zhang et al. 2017] propose Dynamic Key-Value Memory Networks (DKVMN). Unlike in DKT, the DKVMN model uses multiple vectors to represent a student's knowledge. Each exercise is assumed to belong to a one of multiple *latent concepts*, and the model summarizes the student's mastery of each concept in a separate vector. The model will automatically learn which exercise belongs to which concept, and will place similar exercises in the same concept. This allows the model to automatically discover groups of related exercises, which can be useful for analyzing and designing exercises. In their paper, [Zhang et al. 2017] found that DKVMN showed minor performance improvements compared to DKT.

2.2 Recommendation Systems

The goal of a recommendation system is to, given a set of users and a set of items, recommend items that users are likely to want to purchase. One way of finding recommendations is to look for items which are commonly bought by the same user, this approach is known as collaborative filtering. Alternatively, features of the items themselves (such as a text description of a book, or a poster image for a movie) can be used to characterise the items, after which traditional machine learning techniques can be used. It is also possible to combine both of these approaches, to exploit both sources of information [Zhang et al. 2016].

2.2.1 Collaborative Filtering

The premise of Collaborative Filtering (CF) is that users can be described by which items they purchase and items can be described by which users purchase them. This is commonly summarised as "people who bought x also bought y". Any algorithm that only makes use of this implicit relationship between users and items is referred to as a CF algorithm. One of the most successful CF algorithms is matrix factorization, which seeks to represent each item and each user with a vector, so that the inner product of a user with an item is the measure of how likely that user is to buy that item [Koren et al. 2009].

2.2.2 Sequence Recommendation

Recently, recommendation systems are being used in more online applications, such as music recommendation or Youtube video recommendation. In these settings, it is important to consider the order in which a user consumes items and to update models in real-time, since a user's preferences may fluctuate considerably over time. To this end, recommender models based on recurrent neural networks have been used to perform real-time recommendation in online settings [Wu et al. 2017]. Additionally, [Donkers et al. 2017] have designed a recurrent neural network which exploits the assumptions of CF. Their model learns to give personalized predictions for each user by incorporating a user-specific vector embedding into the model input. This kind of user-specific modelling is not used in DKT, where each student is treated as a sequence of interactions.

Chapter 3

Preliminary

This chapter describes the various neural network models that are related to this project, as well as how they are trained and the format of the datasets that are used in their training.

3.1 Feed-forward Neural Networks

Feed-forward Neural Networks (FNN) are a class of machine learning algorithm that use parametric regression to learn a function from data [Specht 1991]. In its standard formulation, a dataset consists of many pairs of *feature* vectors $\mathbf{x}_i$ and *target* vectors $\mathbf{y}_i$.

$$\mathcal{D} = ((\mathbf{x}_i, \mathbf{y}_i))_{i=1}^{D}$$

The goal of training a parametric regression model is to produce a function which maps features to targets such that the relationship between features and targets demonstrated in the dataset is maintained [Hardle and Mammen 1993]. Ideally, the function should *generalize* to feature vectors not present in the dataset in a meaningful way. In a FNN model, the function to be learned is required to be a composition of *activation layers*. Each activation layer is a function of the form

$$A_i(\mathbf{x}) = \sigma_i(T_i\mathbf{x})$$

where σ_i is a non-linear *activation function* applied component-wise, T_i is an affine transformation, and $\mathbf{x}$ is the input vector. Then an n-layer FNN is constructed by simply composing n different activation layers

$$NN = A_n \circ A_{n-1} \circ ... \circ A_1$$

Any affine transformation T_i may be expressed as a matrix multiplication followed by the addition of a constant vector, so that for an input vector $\mathbf{x} \in \mathbb{R}^n$

$$T_i\mathbf{x} = \mathbf{W}_i\mathbf{x} + \mathbf{b}_i$$

for some *transformation matrix* $\mathbf{W}_i \in \mathbb{R}^{m \times n}$ and *bias vector* $\mathbf{b}_i \in \mathbb{R}^m$. Then the entire FNN is parameterized by the elements in each of the transformation matrices and bias vectors (also known as *weights*). The process of *training* a neural network involves finding values for the model parameters so that the resulting function does a good job of mapping features to targets. In order to measure how good any particular combination of parameters is, a *loss* function is used. The loss function maps inputs from $\mathbb{R}^p$ to $\mathbb{R}^+$, where p is the number of model parameters. The output of the loss function is interpreted as a score of how good the input parameters are, with 0 being perfect and larger values being worse. Commonly, the loss function used is the sum of all of the squared errors of the models predictions on the training dataset.

$$Loss(\theta) = \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{D}} (NN_\theta(\mathbf{x}) - \mathbf{y})^2$$

To find good values for the model parameters, an optimization technique called Gradient Descent (GD) is used. GD works by first setting the parameters to small random initial values, then changing the parameters proportionally to the derivative of the negative loss function w.r.t the parameters (also known as the *error gradient*), thereby minimising the loss function. In practice, a variation of GD called *stochastic gradient descent* (SGD) is used. SGD samples a small batch of data (called the *mini-batch*) from the dataset without replacement, then updates the model parameters proportionally to the error gradient of that batch. This process is repeated until all of the data in the dataset has been sampled. Each such cycle through the dataset is called an *epoch*. The model may be trained for multiple epochs, until performance no longer improves significantly. SGD introduces random noise into the optimization process that helps to prevent the algorithm converging to a local minimum, often improving performance [Amari 1993]. There have been a variety of different activation functions used in NN models since their inception, some popular choices have been the Logistic Sigmoid function

$$Sigmoid(x) = \frac{1}{1 + \exp(-x)}$$

the Hyperbolic Tanh function

$$Tanh(x) = 2 \times Sigmoid(2x) - 1$$

and the Rectified Linear Unit.

$$ReLU(x) = max(x, 0)$$

Usually the choice of which activation function to use is made based on the output range of the activation function, as well as how easy it is to train with SGD [Alcantara 2017].

3.2 Recurrent Neural Networks

Recurrent neural networks (RNN) are a modified version of the standard NN model that are designed to work for sequential input data [Mikolov et al. 2010]. In the case of sequential data, each input sample in a dataset consists of a sequence of fixed-dimension input vectors.

$$\mathcal{D} = ((\mathbf{x}_{i,j})_{j=1}^{n}, \mathbf{y})_{i=1}^{D}$$

Since the input sequence may have any length, it is not possible to design a neural network which operates on the entire sequence. Instead, an activation layer is used to learn a mapping from a current *hidden state* $\mathbf{h} \in \mathbb{R}^h$ and an input vector $\mathbf{x} \in \mathbb{R}^d$ to a new hidden state $\mathbf{h}_{t+1}$ [Bengio et al. 1994].

$$A(\mathbf{h}, \mathbf{x}) = \sigma(T[\mathbf{h}, \mathbf{x}])$$

Where $[\mathbf{h}, \mathbf{x}] \in \mathbb{R}^{h+d}$ is the vector concatenation of $\mathbf{h}$ and $\mathbf{x}$. To process an input sequence the RNN begins in some *initial hidden state* $\mathbf{h_0}$, then takes the first element of the sequence as input and computes a new hidden state $\mathbf{h_1}$. This is repeated with each element in turn, until all of the elements in the input sequence have been processed.

$$\mathbf{h}_{t+1} = A(\mathbf{h}_t, \mathbf{x}_t)$$

The resulting hidden state $\mathbf{h_n}$ is used as the vector embedding of the entire sequence, from which the predicted target vector is computed. During training, the initial hidden state $\mathbf{h}_0$ is learned along with the transformation parameters.

3.3 Gated Recurrent Neural Networks

Basic RNNs as described above tend to perform badly, as it is very difficult to train them due to gradients becoming unstable over time-steps [Bengio et al. 1994]. This occurs because at each timestep the current hidden state is multiplied by a matrix W, so the derivative of the n'th hidden state with respect to the first will be dependent on W^n, which will either become very large if the spectral norm of W is greater than 1 (causing *exploding gradients*) or very small if the spectral norm of W is less than 1 (causing *vanishing gradients*) as n increases. This means that the RNN can not effectively learn long-term dependencies in the data, since inputs from previous time-steps do not consistently affect the gradient for the current hidden state [Henaff et al. 2016]. To allow for better training, an alternative form of RNN called *Gated Recurrent Neural Networks* (GRNN) are typically used instead. The key insight of GRNN is that we should have the recurrent function output the *change* in hidden state, instead of the new hidden state, so that

$$\mathbf{h}_{t+1} = \mathbf{h}_t + A(\mathbf{h}_t, \mathbf{x}_t)$$

This simple change leads to a current hidden state that is the sum of the previous hidden states, hence the error gradient of the current hidden state will be contributed to

equally by all previous states [Chung et al. 2014]. However, this is also undesirable for many sequential tasks [Desmet 2002] [Ney et al. 1994]. For example, in natural language processing when predicting the next word in a sentence we would expect that the most recent couple of words should be more important than those hundreds or thousands of time-steps before. To address this, a gating mechanism is introduced which allows the recurrent neural network to selectively erase components of its hidden state at each time-step. The network can learn when to apply its gate, so that it learns to selectively allow relevant information to flow through time while blocking old or irrelevant information [Chung et al. 2014].

3.3.1 Long-Short Term Memory Unit

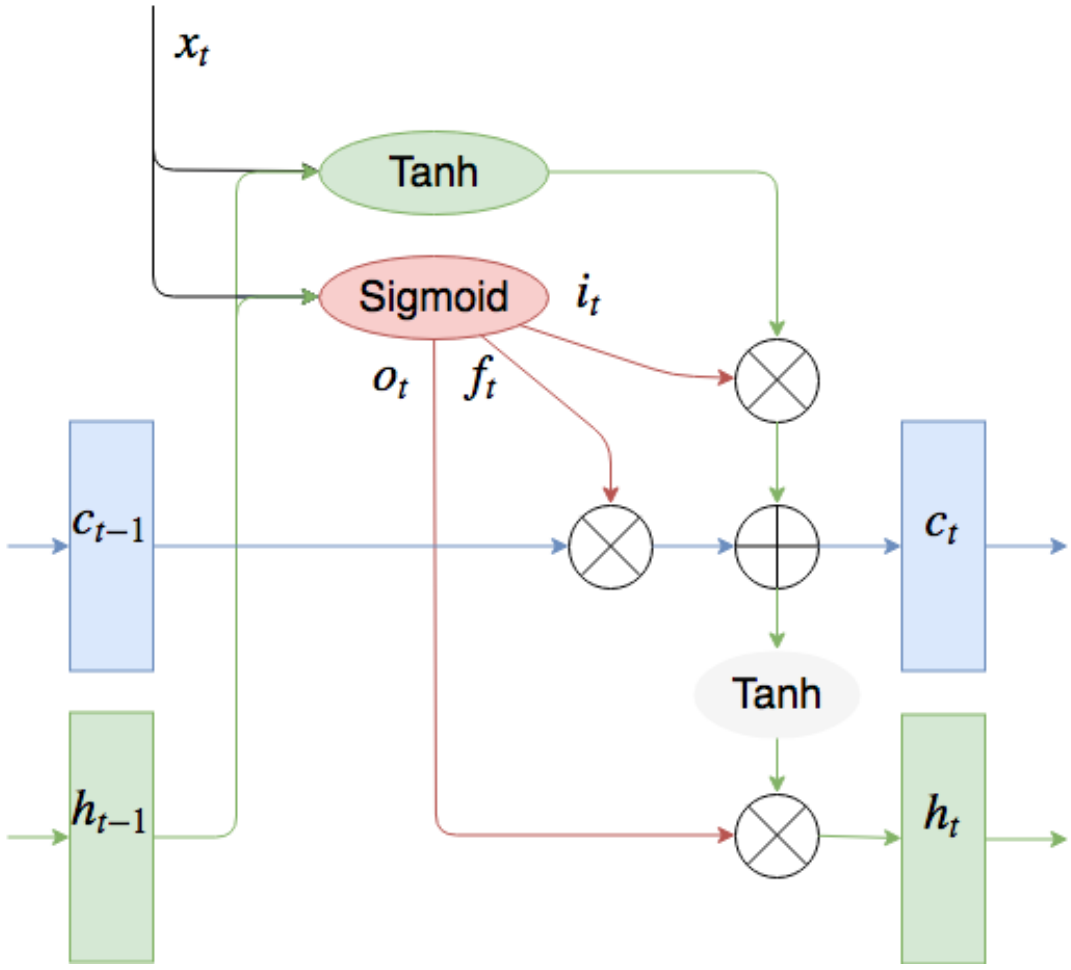

Figure 3.1: LSTM operational diagram. Ellipses with borders represent activation layers, ellipses without borders represent activation functions without affine transformation.

The most popular and oldest variety of GRNN is the Long-Short Term Memory

Unit (LSTM) [Hochreiter and Schmidhuber 1997]. The gating mechanism consists of 3 gates named as *forget* **f**, *input* **i**, *output* **o**. **i** controls which parts of the current input are added to the hidden state, **o** controls which parts of the hidden state are used in the output module of the network, and **f** erases parts of the hidden state. Each of the gates is computed by applying a sigmoid activation layer to the current hidden state output and input. The result is a vector with each component in [0, 1]. To apply the gate to a vector, it is simply multiplied component wise. The result is that a gate with a component close to 0 will erase that component of the vector it acts on.

$$\mathbf{f}_t = Sigmoid(T_f[\mathbf{h}_t, \mathbf{x}_t])$$

$$\mathbf{i}_t = Sigmoid(T_i[\mathbf{h}_t, \mathbf{x}_t])$$

$$\mathbf{o}_t = Sigmoid(T_o[\mathbf{h}_t, \mathbf{x}_t])$$

$$\mathbf{c}_{t+1} = \mathbf{f}_t \odot \mathbf{c}_t + \mathbf{i}_t \odot Tanh(T[\mathbf{h}_t, \mathbf{x}_t])$$

$$\mathbf{h}_{t+1} = LSTM(\mathbf{c}_t, \mathbf{h}_t, \mathbf{x}_t) = \mathbf{o}_t \odot Tanh(\mathbf{c}_{t+1})$$

Here $\mathbf{c}_{t+1}$ is the hidden state of the network, and $\mathbf{h}_{t+1}$ is the output read from memory which is used for prediction. Intuitively, the LSTM hidden state can be thought of as storing information in a memory that persists over time. At each time-step, the three gates control what information is written to memory, erased from memory, and read from memory. During training the network will learn transformations to compute gates that store and maintain the most relevant information about inputs the model has seen so far.

3.3.2 Gated Recurrent Unit

More recently the Gated Recurrent Unit (GRU) has been proposed as a simpler alternative to LSTM [Chung et al. 2014]. The GRU does not need an extra output state, and instead uses the hidden state directly to compute predictions. In addition, the GRU only uses two gating variables *reset* **r** and *update* **z**.

$$\mathbf{r}_t = \sigma(T_r[\mathbf{h}_t, \mathbf{x}_t])$$

$$\mathbf{z}_t = \sigma(T_z[\mathbf{h}_t, \mathbf{x}_t])$$

$$\tilde{\mathbf{h}}_{t+1} = Tanh(T_h[\mathbf{r}_t \odot \mathbf{h}_t, \mathbf{x}_t])$$

$$\mathbf{h}_{t+1} = GRU(\mathbf{h}_t, \mathbf{x}_t) = (1 - \mathbf{z}_t) \odot \mathbf{h}_t + \mathbf{z}_t \odot \tilde{\mathbf{h}}_{t+1}$$

Again, the hidden state is used to store relevant information, except unlike a LSTM the GRU couples the forget and input gates together into the one update gate, which controls erasing and writing to memory.

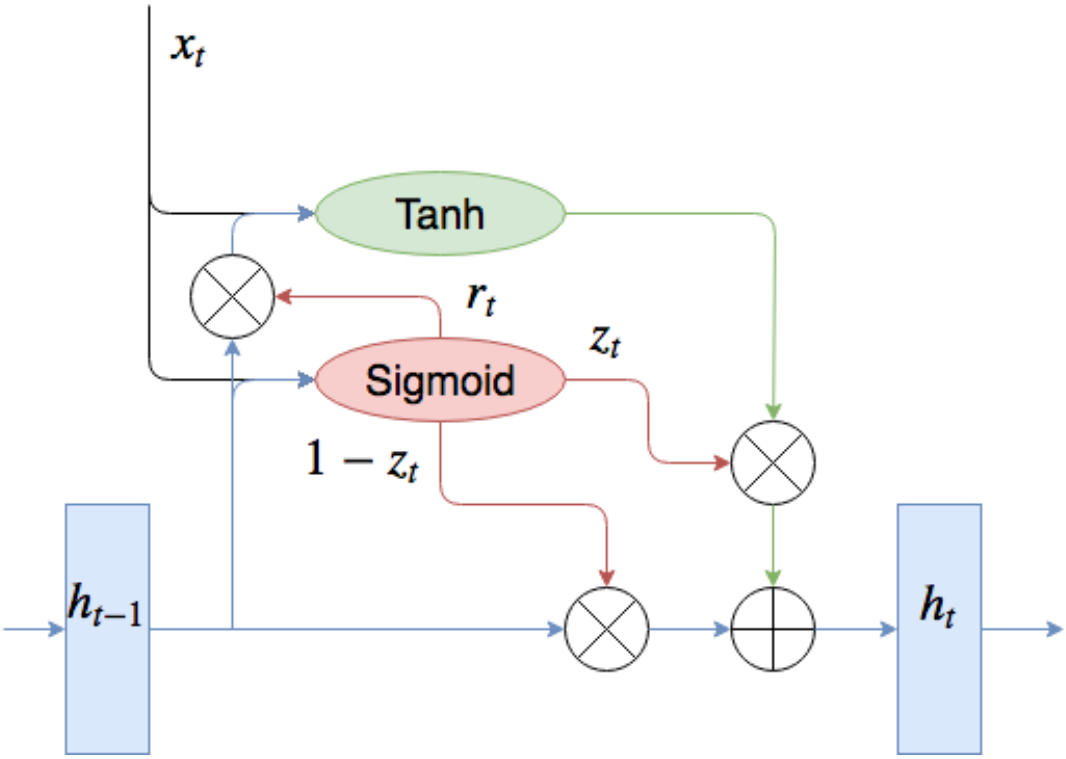

Figure 3.2: GRU operational diagram.

3.4 Memory Augmented Neural Networks

LSTM has proven to be very successful at natural language processing, and other sequential modelling tasks that require long-term dependency modelling. However, an LSTM stores all of its memory in a single hidden vector. This lack of compartmentalization means that the LSTM must store all relevant information in the same dense representation. This causes LSTM to have difficulty accurately remembering very long sequences of more than hundreds of time-steps [Zaremba and Sutskever 2014]. Memory Augmented Neural Networks (MANN) were proposed as a solution to the above problem: they allow the network to keep multiple hidden state vectors and read or write to these vectors separately. Each of the hidden state vectors in a Memory Network is referred to as a *memory slot*, and each memory slot can be thought of as a row in a *memory matrix* [J. Weston 2015]. The most common implementation of MANN uses a simple *attention* mechanism to choose which memory slots to read and write to. Given a current input $\mathbf{x}_t \in \mathbb{R}^d$ and memory matrix $\mathbf{M} \in \mathbb{R}^{N \times d}$, each memory slot is assigned a *correlation weight* which represents how similar the input is to each slot. Typically this is implemented by computing the standard inner product between the input and each memory slot, then passing the result through a Softmax function

$$\mathbf{w}_t = Softmax(\mathbf{M}\mathbf{x}_t)$$

where $Softmax(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$. The softmax function normalises the vector of similarities, so that the components sum to 1 and each component is in [0, 1]. A *summary* vector $\mathbf{r}_t$ is then read from memory by taking a sum of memory slots weighted by correlation.

$$\mathbf{r}_t = \mathbf{w}_t^T \mathbf{M}$$

$\mathbf{r}_t$ is treated as the model output at time-step t from which prediction is performed. Finally, $\mathbf{r}_t$ and $\mathbf{x}_t$ are then taken as input to update the memory matrix, using a simple gating mechanism [Graves et al. 2014] [J. Weston 2015].

$$\mathbf{e}_t = Sigmoid(T_e[r, x])$$

$$\mathbf{a}_t = Tanh(T_a[r, x])$$

$$\mathbf{M}_{t+1} = \mathbf{M}_t \odot (\mathbf{w}_t \mathbf{e}_t^T) + (\mathbf{w}_t \mathbf{a}_t^T)$$

3.5 Dynamic Key-Value Memory Networks

The Dynamic Key-Value Memory Network (DKVMN) is a type of MANN that is specifically designed for the task of KT. The DKVMN uses its external memory to explicitly model a students knowledge of multiple different topics. Each exercise is assumed to belong to a *latent concept*, with each memory slot storing a students current mastery of a different latent concept. This allows the model to update a students mastery of one concept, while leaving unrelated concepts unchanged [Zhang et al. 2017]. In standard MANN models, the current content in each of the memory slots is used to compute the correlation weight vector, which determines which memory slot to read and write to at each time-step. However, in KT the item input at each time-step is an exercise, while the content we wish to read is the *mastery* of that exercise. Because of this, the input vectors and hidden vectors have a different semantic meaning and hence should exist in separate vector spaces. To accommodate this, DKVMN uses two separate memory matrices named *key* $\mathbf{M}^K \in \mathbb{R}^{N \times d_k}$ and *value* $\mathbf{M}^V \in \mathbb{R}^{N \times d_v}$. Key is static, remaining unchanged over time-steps, and is used to compute similarity to the current input to determine which slots will be read and written to. Value is dynamic, it contains the students current mastery vectors and is updated at each time-step [Zhang et al. 2017].

3.6 Knowledge Tracing Problem Definition

For the formal problem definition of KT, we assume that students interact with an ITS to produce a sequence of interaction events. For a given a dataset, we denote $\mathcal{Q}$ be the set of distinct exercise tags present in the dataset, where an exercise's tag is a natural number identifier for it. Each interaction event is a pair of the form (q, r) where $q \in \mathcal{Q}$ is the tag of the exercise the student attempted, and $r \in 0, 1$ is the outcome of the students attempt, with 1 representing success and 0 representing failure. A stu-

dents history is represented as an ordered sequence of such exercise interaction pairs $X = ((q_1, r_1), (q_2, r_2), \ldots, (q_n, r_n)$. The goal of KT is then to predict the outcome of a students next exercise interaction given their history, that is to model the probability $P(r_{n+1} = 1 \mid q_{n+1}, X)$ Since NN require vector input and learn affine transformations of their input, it is not feasible for a NN to be trained with natural number identifiers as input. To facilitate NN models we learn vector embeddings for each exercise, similarly to how natural language processing models learn word vector embeddings [Tang et al. 2014] [Bojanowski et al. 2016]. For every exercise $q_i \in \mathcal{Q}$, we keep a vector embedding $\mathbf{v}_i$ which will be used as input to the model in place of q_i. During training the embedded vectors will be treated just like any other model parameter, and optimized along with the transformation weights.

Chapter 4

Distributed Memory Networks

Recently [Zhang et al. 2017] have proposed a new KT model called Dynamic Key-Value Memory Networks (DKVMN), which specifically models a students knowledge across different *latent concepts*. Each exercise is assumed to belong to a latent concept, and during training the model will learn to assign similar exercises to the same latent concept. In their paper they compare DKVMN against LSTM and a sophisticated version of BKT called BKT+. They find that DKVMN achieves superior performance, making it the state of the art model for KT. In addition to improved performance, it has several other benefits over LSTM including resistance to overfitting, smaller number of parameters and automatic discovery of similar exercises via latent concepts. In this chapter we discuss some of the limitations of the DKVMN model and propose a new model called Distributed Memory Networks (DMN) which does not suffer from these drawbacks.

4.1 Distributed Memory Networks

Although the DKVMN model has achieved a better performance than LSTM, it still has some fundamental limitations. For example, the DKVMN model does not use the current memory content when calculating the erase and add vectors $\mathbf{e}_t$ and $\mathbf{a}_t$. This means that, given the same input, it will always make the same change to memory. However, this does not seem to accurately reflect the way a student learns. The first time a student answers an exercise correctly, there should be a large change in their mastery. Conversely, if a student answers that exercise correctly after having seen it many times before then their mastery should only change by a small amount. DKVMN is incapable of modelling this phenomenon. The reason why the DKVMN model suffers from this flaw is due to its adoption of the ideas used in Memory Augmented Neural Networks (MANN). In the MANN model it is not necessary to use the current memory content when calculating the state update vectors $\mathbf{e}_t$ and $\mathbf{a}_t$, because the correlation weights $\mathbf{w}_t$ are computed based on the current memory content. In DKVMN, however, the correlation weights are computed from the current input and a static matrix $\mathbf{M}_k$, so the dynamic state of the DKVMN model is not used for this computation at all. This problem can be avoided by simply using the memory summary $\mathbf{r}_t$ along with the current input $\mathbf{x}_t$ to compute the update vectors. However, if the

correlation weight $\mathbf{w}_t$ has high values for multiple slots, then they will be updated using each others memories which means it is no longer the case that each memory slot corresponds to one single concept. We therefore propose a new model Distributed Memory Network (DMN) which uses the same read process as DKVMN, however uses a different write process which allows the model to make variable changes to its state at each step, depending on its current memory content, while still maintaining independent memory slots.

4.1.1 Model Overview

The DMN is a type of MANN, it uses an external memory matrix $\mathbf{M}_t \in \mathbb{R}^{N \times d_v}$ to process the sequential history of a student $X = ((q_1, r_1), (q_2, r_2), \ldots, (q_n, r_n))$. At each time-step of the input sequence, the model takes as input a discrete exercise tag $q_t \in \mathcal{Q}$ and *reads* $\mathbf{r}_t \in \mathbb{R}^{d_v}$, the students current mastery of q_t, from $\mathbf{M}_t$. $\mathbf{r}_t$ is then used compute the probability of the student successfully completing the given exercise $p(r_t = 1 \mid q_t, (q_i, r_q)_{i=1}^{t-1})$. It then takes the outcome of the students attempt r_t and uses (q_t, r_t) to *write* an update to $\mathbf{M}_t$.

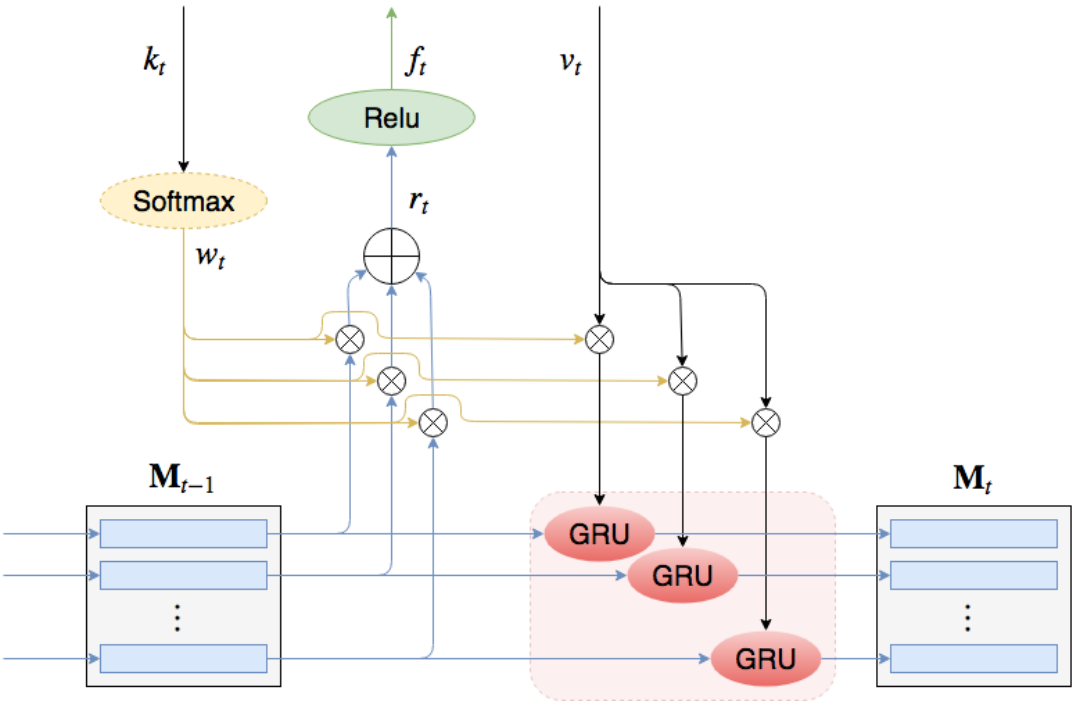

Figure 4.1: DMN operational diagram. Ellipses with dotted borders represent activation layers with a linear transformation (i.e. without the bias term). Note functions with the same colour share the same weights.

4.1.2 Correlation Weight

Each exercise q_t has its own vector of correlation weights $\mathbf{w}_t \in \mathbb{R}^N$, which encodes how correlated that skill is with each of the memory slots in $\mathbf{M}_t$. We use the same method as [Zhang et al. 2017] to compute correlation weights: the input exercise q_t is first substituted with its vector embedding $\mathbf{k}_t \in \mathbb{R}^{d_k}$. Then the correlation weight is computed by applying Softmax to the inner product of $\mathbf{k}_t$ with each of the slots in a *key* matrix $\mathbf{M}^K \in \mathbb{R}^{N \times d_k}$. Intuitively, this process is the same as the standard memory augmented neural network addressing method, except that a static matrix is used instead of the dynamic memory content. We note however that because the key matrix is static, this operation is precisely equivalent to applying a linear transformation to $\mathbf{k}_t$ and then a Softmax function. That is, a Softmax activation layer without the bias term.

$$\mathbf{w}_t = Softmax(\mathbf{M}^K \mathbf{k}_t)$$

4.1.3 Read Process

Given correlation weights $\mathbf{w}_t$, a *summary* $\mathbf{r}_t$ of the current memory relevent to exercise q_t is computed by taking a weighted sum over memory slots in the value matrix, where the slots are weighted by $\mathbf{w}_t$.

$$\mathbf{r}_t = \mathbf{w}_t^T \mathbf{M}_t$$

To account for different exercises having different difficulties, both $\mathbf{r}_t$ and $\mathbf{k}_t$ are passed through a ReLU[1] activation layer to compute an exercise specific vector $\mathbf{f}_t$.

$$\mathbf{f}_t = ReLU(T_f[\mathbf{r}_t, \mathbf{k}_t])$$

Finally $\mathbf{f}_t$ is passed through a Sigmoid activation layer to compute the predicted probabillity of correctness.

$$p(r_t = 1 \mid q_t, (q_i, r_q)_{i=1}^{t-1}) = Sigmoid(T\mathbf{f}_t)$$

4.1.4 Write Process

After the student answers exercise q_t, the correctness of their response r_t along with q_t is used to update the memory matrix $\mathbf{M}_t$. The tuple (q_t, r_t) replaced by an embedding vector $\mathbf{v}_t$, which captures the students knowledge growth after completing q_t. To compute the updated memory $\mathbf{M}_{t+1}$ we treat each memory slot as a seperate independent RNN with a single hidden state vector. $\mathbf{v}_t$ is split up according to the correlation weight $\mathbf{w}_t$ and distributed across all of the memory slots, so that the ith slot receives $w_t(i)\mathbf{v}_t$ as input. Each memory slot then independently computes its own updated content as in a GRNN. This framework allows us to use any GRNN model to perform

[1] In the original DKVMN paper a Tanh activation is used here, however it is more common in the literature to use ReLU activations outside of the recurrent section [Gulcehre et al. 2016] [Graves et al. 2014].

the updates, in our implementation we choose to use the GRU update method because it is simpler than LSTM, which leads to significant computation time reduction considering the update rule is applied to multiple slots at each time-step.

$$\mathbf{M}_{t+1}(i) = GRU(\mathbf{M}_t(i), w_t(i)\mathbf{v}_t)$$

Here, $\mathbf{M}_t(i)$ is the i'th row of $\mathbf{M}_t$. Note that we use the same GRU function for each slot, that is the parameters of the GRU update are shared across all slots.

4.2 Accumulative Distributed Memory Networks

In the KT setting it is common for students to attempt the same exercise multiple times consecutively. In this case, there is no need to read and write to the distributed memory at each time-step, since it will be the same memory slots being read and written to until the student moves on to another exercise. Instead, we can use a simple GRNN model while the student is attempting the same question, then use a DMN update to the GRNN hidden state when the student changes to a new exercise. A similar model called Hierarchical Recurrent Neural Network (HRNN) has been put forward by [Quadrana et al. 2017] for the purpose of performing recommendations in real-time. They use a session-level RNN and a user-level RNN together: every time a user purchases an item the session-level RNN is updated, and when the user logs-out of the system the hidden state of the session-level RNN is used to update the user-level RNN. When a new session is started, the user-level RNN is used to seed the initial hidden state of the session-level RNN. This allows the user-level RNN to remember long-term behaviour, while the session-level RNN adapts to the users current whims.

In our proposed Accumulative DMN (ADMN) model we use an external memory matrix $\mathbf{M}_t$ in addition to a hidden state vector $\mathbf{h}_t$. We define a boolean variable δ_t to be 0 if $q_t = q_{t-1}$ and 1 if $q_t \neq q_{t-1}$. The updated memory $\bar{\mathbf{M}}_{t+1}$ is computed as in DMN, except the hidden state vector $\mathbf{h}_t$ is used as input instead of $\mathbf{v}_t$.

$$\bar{\mathbf{M}}_{t+1} = DMN(\mathbf{M}_t, \mathbf{w}_{t-1}\mathbf{h}_t^T)$$

And the updated hidden state $\bar{\mathbf{h}}_t$ is precisely the summary vector read from memory as in DMN.

$$\bar{\mathbf{h}}_t = \mathbf{w}_t^T \bar{\mathbf{M}}_t$$

Then if $\delta_t = 1$, the memory matrix and hidden state will be replaced with their updated versions, otherwise they will remain unchanged.

$$\mathbf{M}_{t+1} = \delta_t \times \bar{\mathbf{M}}_{t+1} + (1 - \delta_t) \times \mathbf{M}_t$$

$$\mathbf{r}_t = \delta_t \times \bar{\mathbf{h}}_t + (1 - \delta_t) \times \mathbf{h}_t$$

$$\mathbf{f}_t = ReLU(T_f[\mathbf{r}_t, \mathbf{k}_t])$$

Finally, the hidden state is updated using a standard GRU with the response vector $\mathbf{v}_t$ as input.

$$\mathbf{h}_{t+1} = GRU(\mathbf{r}_t, \mathbf{v}_t)$$

Intuitively, while a student is attempting the same exercise all of their responses will be accumulated in the hidden state vector. When they attempt a new exercise, the accumulated value will be written to memory as in DMN. This allows the model to remember student behaviours over a longer period of time: since the memory is only updated when a student changes exercise, each time-step the model remembers can hold information from many time-steps of the raw data [Chung et al. 2016]. In addi-

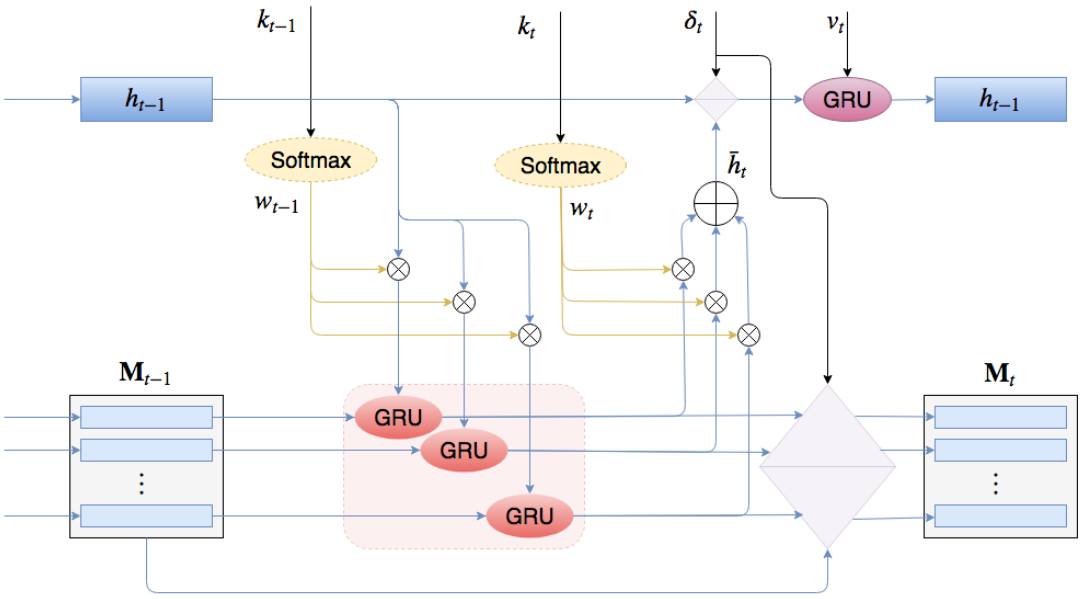

Figure 4.2: ADMN operational diagram. Diamonds represent conditional switch, which chooses one of its inputs depending on boolean input δ.

tion, this model has the effect that memory slots will read from each-others content directly. Consider that a student answers two exercises q_1 and q_2 consecutively which read/write different memory slots. Then after completing exercise q_1 the hidden state will contain the content of the slot that q_1 reads from, which will then be used as input to write to q_2's slot. This is unlike regular DMN, where memory slots are only ever input response vectors $\mathbf{v}_t$.

4.3 Independent Reads

In the DKVMN and DMN models proposed thus far, each exercise both reads from and writes to the same slots, with the same weights. This means that when discovering hidden concepts, the model can only find groups of exercises which are 'similar' to each other or 'dissimilar'. However, exercises often have a hierarchical structure of dependencies, so that students must first master one concept before they can be-

gin learning another. For example, consider a student learning basic arithmetic. They must first understand how to add and multiply positive integers before they can learn to add and multiply rational numbers. In this situation there is a directed relationship between exercises, in order for a model to make predictions about a students mastery of fractions, it would need to know about the students mastery of integer arithmetic. This would mean that memory-augmented models should learn to put both integer and rational arithmetic in the same memory slot. However, if the model is to make predictions for integer arithmetic after the student has attempted some rational arithmetic problems, the joint memory slot will be contaminated by the students performance on rational arithmetic, while the students mastery of integer arithmetic should not change. If on the other hand, the model learns to put integer and rational arithmetic into separate memory slots, then the model could not use information about the students performance on integer arithmetic to predict their performance on rational arithmetic. We therefore propose Independent-read ADMN (IADMN) which allows exercises to read and write to separate memory slots. We use a single addressing matrix $\mathbf{M}^K$, but each exercise is used to create two embedding vectors $\mathbf{k}_r$ and $\mathbf{k}_w$. These are then used to compute two correlation weights, $\mathbf{w}_r$ and $\mathbf{w}_w$.

$$\mathbf{w}_r = Softmax(\mathbf{M}^K \mathbf{k}_r)$$

$$\mathbf{w}_w = Softmax(\mathbf{M}^K \mathbf{k}_w)$$

The model then reads from memory using $\mathbf{w}_r$, and writes using $\mathbf{w}_w$. $\mathbf{f}_t$ is computed using both $\mathbf{k}_r$ and $\mathbf{k}_w$.

$$\mathbf{f}_t = ReLU(T_f[\mathbf{r}_t, \mathbf{k}_r + \mathbf{k}_w])$$

In this model, it is possible for one exercise q_1 to read from the memory slot of another exercise q_2, without q_2 being aware of q_1. This allows for directed relationships between exercises to be discovered.

Chapter 5

Experiments

In this chapter we evaluate and compare our DMN models with two baseline KT models: LSTM and DKVMN. We test all models capability to predict student performance on 4 KT datasets. We additionally use our models to extract exercise dependencies from one of these datasets.

5.1 Datasets

We use 4 publically available KT datasets: ASSISTments2009, ASSISTments2015, STATICS, and JUNYI. These datasets contain interaction logs from students with various online intelligent tutoring systems. An interaction is an event where a student provides an answer to a given exercise. From each interaction, the ID of the exercise which was attempted and the 'correctness' of the student's answer are recorded.

5.1.1 ASSISTments 2009

This dataset [Feng et al. 2009] is gathered from the ASSISTments online tutoring platform, during the year 2009. This platform provides mathematics exercises of around high-school level topics. When it was first released this dataset contained duplicated records, as such some older papers which used this dataset have unreliable results [Xiong et al. 2016]. Since then an updated version has been released without erroneous duplicates, we use the updated version in our experiments. The dataset contains data from two different interactive modes "skill-builder" and "non-skill-builder", in our experiments we use the "skill-builder" data as in this dataset each interaction is labelled with a unique skill ID, which makes preprocessing much simpler. As in [Zhang et al. 2017], we discard records without skill names, after which this dataset contains 4,151 students, 110 exercise tags, and 325,637 interactions.

5.1.2 ASSISTments 2015

This dataset is also gathered from the same ASSISTments online platform, but during the year 2015. Unlike the 2009 version, this dataset only contains records from one interactive mode, all of which are used. In our preprocessing, we removed records

which did not have a 'correctness' value of either 0 or 1, as these records indicate the student requesting a hint. The remaining dataset contains 19,840 students, 100 exercise tags, and 683,801 interactions.

5.1.3 Statics 2011

This dataset [Steif and Bier] [Koedinger et al. 2010] is gathered from a college-level engineering statics course. In the statics system, each exercise consists of solving multiple *steps*. Because this dataset contains a relatively small number of different exercises as is, we instead opt to use each *step* of an exercise as a separate exercise input to our models. That is, the exercise ID and the step ID together act as the input identifier, so that different steps of the same exercise will be treated as separate input IDs. With this change in mind, the dataset contains 333 students, 1,223 exercise tags and 189,297 interactions.

5.1.4 Junyi Academy 2015

This dataset [Chang et al. 2015] is taken from Junyi Academy [1], a website similar to Khan Academy [2] which provides exercises from a wide range of topics. This dataset was taken from the use-logs of the Junyi website in 2015, the full dataset contains logs from a quarter of a million students, and over 25 million interactions. To reduce the amount of computation time needed, in our experiments we randomly selected 10,000 students to use from this dataset, in our selected sub-sample there are 720 exercises and 2,122,314 interactions.

5.2 Implementation Details

For all of the datasets, we remove students with 3 or less interactions, as we figured such little information would be insufficient to properly model a students learning. Our models do not make predictions for the correctness of the first interaction a student has, but make predictions for every interaction afterwards. Datasets were split so that 60%/20%/20% of students were placed into training/validation/testing portions. After each epoch of training, models were evaluated on the validation data, the epoch which achieved the highest validation score was then evaluated on the testing dataset. Our models were trained using the Adam optimization algorithm [Kingma and Ba 2014] with a learning rate of 0.005, and using cross-entropy loss. Additionally, all gradients were clipped to have a norm of at most 50, which is known to improve RNN training [Pascanu et al. 2013]. All trainable parameters were initialized using Glorot uniform initialization [Glorot and Bengio 2010]. A mini-batch size of 32 was used on all datasets except for STATICS, where a mini-batch size of 8 was used. We

[1] Junyi Academy (http://www.junyiacademy.org/) is established in 2012 on the basis of the open-source code released by Khan Academy

[2] https://www.khanacademy.org/

Test AUC

Dataset	LSTM	DKVMN	DMN	ADMN	IADMN
ASST2009	0.81772 ± 1.13e-3	0.81717 ± 3.36e-4	0.82413 ± 7.56e-4	**0.82506 ± 5.11e-4**	**0.82505 ± 5.19e-4**
ASST2015	0.72845 ± 5.11e-4	0.72743 ± 4.06e-4	0.73346 ± 3.27e-4	**0.73471 ± 2.83e-4**	**0.73440 ± 3.89e-4**
STATICS	0.82729 ± 1.27e-3	0.82635 ± 1.64e-3	0.83329 ± 6.21e-4	0.83361 ± 9.75e-4	**0.83453 ± 5.86e-4**
JUNYI	0.79988 ± 4.39e-4	0.79820 ± 1.39e-3	**0.80786 ± 3.87e-4**	**0.80752 ± 3.41e-4**	0.80634 ± 8.39e-4

Table 5.1: Model AUC mean ± standard deviation score on testing datasets over 6 runs. Models with best performance in **bold**.

found that when using a mini-batch size of 32 on STATICS, models performed poorly, due to lack of stochastic noise.

5.3 Prediction Accuracy

To evaluate the predictive capabilities of our models, we train each model to perform KT on each of the datasets. To measure their performance, we use the Area Under the Receiver Operating Characteristic Curve (AUC) metric [Ling et al. 2003]. AUC is a reliable metric, even when there is an imbalance in the frequency that predicted classes appear, and it is predominately used by other papers in the area of KT, so our results can be compared readily. An AUC score of 0.5 implies a model is as good as flipping a coin, a score of 1.0 implies a model is always correct. Models were trained 6 times with different random initializations, their mean performance on the testing set is shown, along with standard deviation. Previously, [Zhang et al. 2017] found that their DKVMN model outperformed an LSTM on the ASST2009, ASST2015 and STATICS datasets. However, our findings are contrary to this: as can be seen in Table 5.1, we find that a regular LSTM slightly outperforms DKVMN. Our reported DKVMN scores are more or less the same as those reported by [Zhang et al. 2017], however our LSTM performs significantly better than theirs. We hypothesise this discrepancy is due to our training models with Adam optimization, while in previous papers a standard SGD optimizer was used. Optimization methods which use second derivative information have been shown to achieve significantly better results for training RNN [Bengio et al. 2013], and the Adam optimizer does make use of second derivative information. We do find, as expected, that the DMN model significantly outperforms both the LSTM and DKVMN models on all datasets. The ADMN model does see modest improvements over DMN on the ASST2009 and ASST2015 datasets, however it does not improve performance on STATICS and JUNYI. In the STATICS dataset, each exercise ID corresponds to a unique *step* in solving a problem, and as such there are no repeated exercise tags in this dataset. Then we should expect that the performance of ADMN does not vary at all from DMN on this dataset. Interestingly, this is not the case for JUNYI: the JUNYI dataset has the longest spanning consecutive exercises of any dataset, when a student repeats exercises they will on average repeat the same exercise 10 times in a row. This leads us to believe that the model was relying

too much on its single hidden state, without fully utilizing the benefits of the external memory. We therefore conclude that the ADMN works best when there are medium length streaks of consecutive exercises, of length around 5.

5.4 Model Size Comparison

Previously, [Zhang et al. 2017] has found that DKVMN requires fewer parameters than LSTM in order to achieve optimal accuracy. In order to see if our DMN models still retain this property, we show model performances for various settings of the model hyper-parameters. Here N is the number of memory slots. For the memory augmented models, to reduce the number of settings to search, we restrict the dimension of the external memory matrix and key matrix to be the same $d_k = d_v = d$.

Model	ASST2009			ASST2015			STATICS			JUNYI		
	d	N	test AUC	d	N	test AUC	d	N	test AUC	d	N	test AUC
LSTM	20	-	0.81667	20	-	0.72842	20	-	0.82729	20	-	0.79917
	40	-	0.81743	40	-	0.72845	**40**	-	**0.82751**	**40**	-	**0.79988**
	80	-	0.81772	**80**	-	**0.72858**	80	-	0.82538	80	-	0.79870
	120	-	**0.81777**	120	-	0.72739	120	-	0.82468	120	-	0.79858
	160	-	0.81738	160	-	0.72667	160	-	0.82313	160	-	0.79645
DKVMN	20	20	0.81623	20	20	0.72682	20	20	0.82591	**20**	**20**	**0.79820**
	40	20	0.81724	40	20	0.72657	40	20	0.82482	40	20	0.79610
	60	20	0.81685	60	20	0.72641	60	20	0.82479	60	20	0.79493
	20	30	0.81566	**20**	**30**	**0.72743**	20	30	0.82614	20	30	0.79703
	40	30	0.81691	40	30	0.72674	**40**	**30**	**0.82635**	40	30	0.79597
	60	**30**	**0.81729**	60	30	0.72660	60	30	0.82393	60	30	0.79537
	20	40	0.81597	20	40	0.72702	20	40	0.82521	20	40	0.79713
	40	40	0.81692	40	40	0.72611	40	40	0.82480	40	40	0.79439
	60	40	0.81717	60	40	0.72565	60	40	0.82439	60	40	0.79414
DMN	20	20	0.82360	20	20	0.73335	20	20	0.83323	20	20	0.80685
	40	20	0.82419	40	20	0.73306	40	20	0.83229	40	20	0.80725
	60	**20**	**0.82438**	60	20	0.73315	60	20	0.83329	60	20	0.80716
	20	30	0.82305	20	30	0.73328	20	30	0.83331	20	30	0.80724
	40	30	0.82408	40	30	0.73346	40	30	0.83322	40	30	0.80732
	60	30	0.82413	60	30	0.73301	60	30	0.83343	60	30	0.80724
	20	40	0.82342	20	40	0.73336	20	40	0.83307	20	40	0.80681
	40	40	0.82388	**40**	**40**	**0.73357**	**40**	**40**	**0.83375**	**40**	**40**	**0.80786**
	60	40	0.82379	60	40	0.73334	60	40	0.83302	60	40	0.80734
ADMN	20	20	0.82333	**20**	**20**	**0.73471**	20	20	0.83345	20	20	0.80732
	40	20	0.82502	40	20	0.73442	**40**	**20**	**0.83361**	**40**	**20**	**0.80758**
	60	20	0.82506	60	20	0.73397	60	20	0.83289	60	20	0.80673
	20	30	0.82280	**20**	**30**	**0.73471**	20	30	0.83320	20	30	0.80677
	40	**30**	**0.82532**	40	30	0.73441	40	30	0.83308	40	30	0.80752
	60	30	0.82452	60	30	0.73404	60	30	0.83256	60	30	0.80673
	20	40	0.82257	20	40	0.73467	20	40	0.83320	20	40	0.80688
	40	40	0.82385	40	40	0.73439	40	40	0.83356	40	40	0.80709
	60	40	0.82474	60	40	0.73399	60	40	0.83302	60	40	0.80684
IADMN	20	20	0.82278	**20**	**20**	**0.73440**	20	20	0.83395	20	20	0.80631
	40	20	0.82463	40	20	0.73386	40	20	0.83439	40	20	0.80634
	60	20	0.82463	60	20	0.73366	60	20	0.83428	60	20	0.80587
	20	30	0.82308	20	30	0.73438	20	30	0.83430	20	30	0.80613
	40	30	0.82453	40	30	0.73404	**40**	**30**	**0.83453**	40	30	0.80632
	60	**30**	**0.82505**	60	30	0.73370	60	30	0.83370	60	30	0.80582
	20	40	0.82333	20	40	0.73430	20	40	0.83354	20	40	0.80593
	40	40	0.82415	40	40	0.73401	40	40	0.83391	**40**	**40**	**0.80653**
	60	40	0.82458	60	40	0.73370	60	40	0.83438	60	40	0.80549

Table 5.2: Model mean AUC score on testing dataset for various settings of d and N. Best performing parameter settings in **bold**.

As can be seen in table 5.2, we do find that the memory based models tend to need a smaller hidden dimension d than the LSTM in order to achieve optimal performance on the ASST2009 and ASST2015 datasets. On the STATICS dataset, all models achieve optimal performance for $d = 40$, this could simply mean that the dataset is relatively simple, so even the LSTM is able to model it with a small hidden dimensionality. It should be noted that the memory models do additionally need an external memory matrix, however the only parameters needed to facilitate this are the weights of the *key* matrix $M^K \in \mathbb{R}^{N \times d}$, whereas the hidden dimension size influences the number of parameters needed in all of the transformations used to compute gates and updates and outputs. We find that, overall, all of the memory models need around the same hidden dimensionality to achieve optimal performance. We do find, however that the ADMN and IADMN variations need a smaller value of N than DMN. We believe this is due to these models having a hidden state vector which can store short-term information, and so these models need less memory slots. We also observe that the DMN, ADMN, and IADMN models are robust to the choice of these parameters, as they can vary significantly while the model performance remains for the most part unchanged.

5.5 Over-fitting Resistance

In their original DKVMN paper, [Zhang et al. 2017] found that DKVMN is significantly more resistant to over-fitting compared to LSTM. Over-fitting describes a situation where a model learns to perform very well on the training dataset, while performing worse on unseen testing data. Here we compare the training and validation AUC of LSTM, DKVMN, and DMN during training to show that DMN is also resistant to over-fitting the training data.

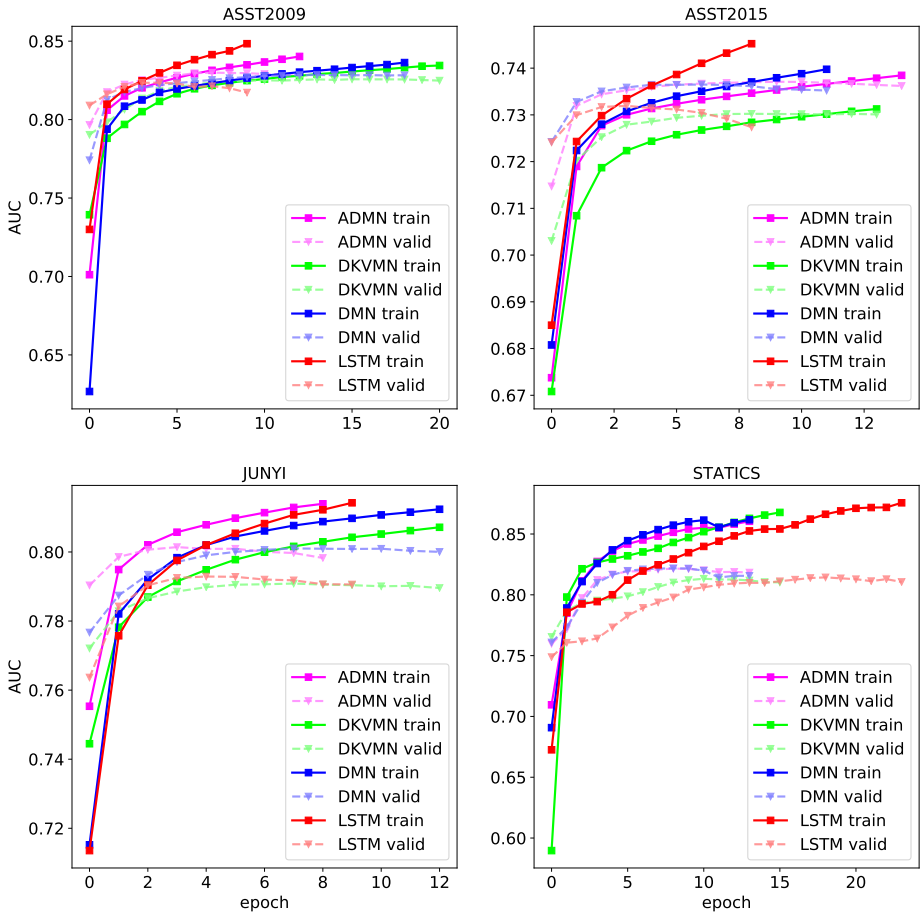

Figure 5.1: Training vs Validation AUC over epochs on each dataset

In Figure 5.1 we see that on the ASST2009 and ASST2015 datasets the LSTM model does indeed over-fit, as after only 3 or 4 epochs its training and validation AUC scores begin to rapidly diverge. On the other hand, all of the memory based models fair much better, maintaining similar training and validation AUC scores until convergence. On the other two datasets we find that all models over-fit, although it can definitely be seen that DMN and ADMN over-fit less than DKVMN and LSTM.

5.5.1 Memory Usage

In order to visualize how our memory based models utilizate their external memory, here we show the learned correlation weights for our models when trained on each of the datasets.

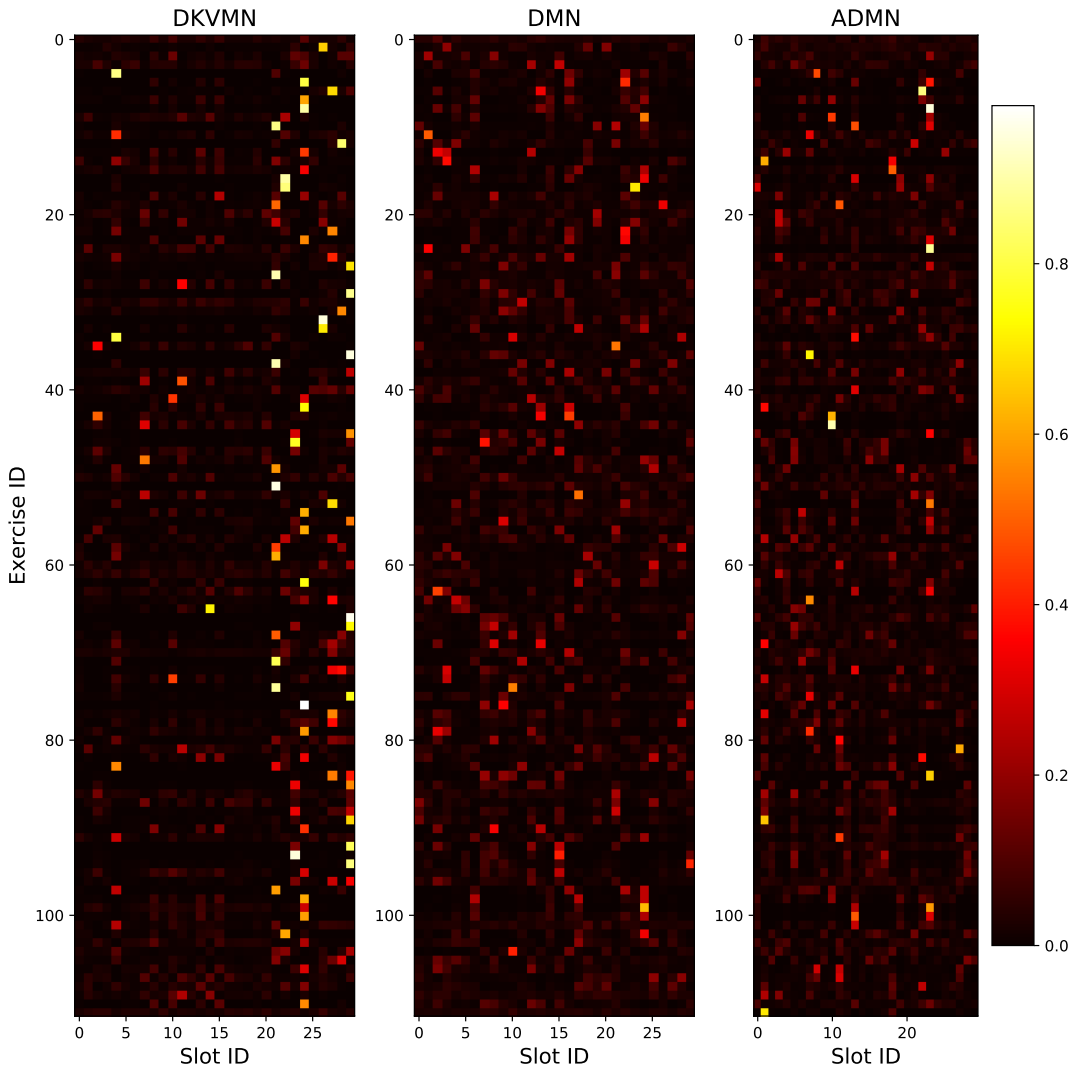

Figure 5.2: Memory slot heatmaps for models trained on ASST2009 with N=30, $d = 60$. The colour of a square represents the value of correlation between the corresponding exercise and memory slot.

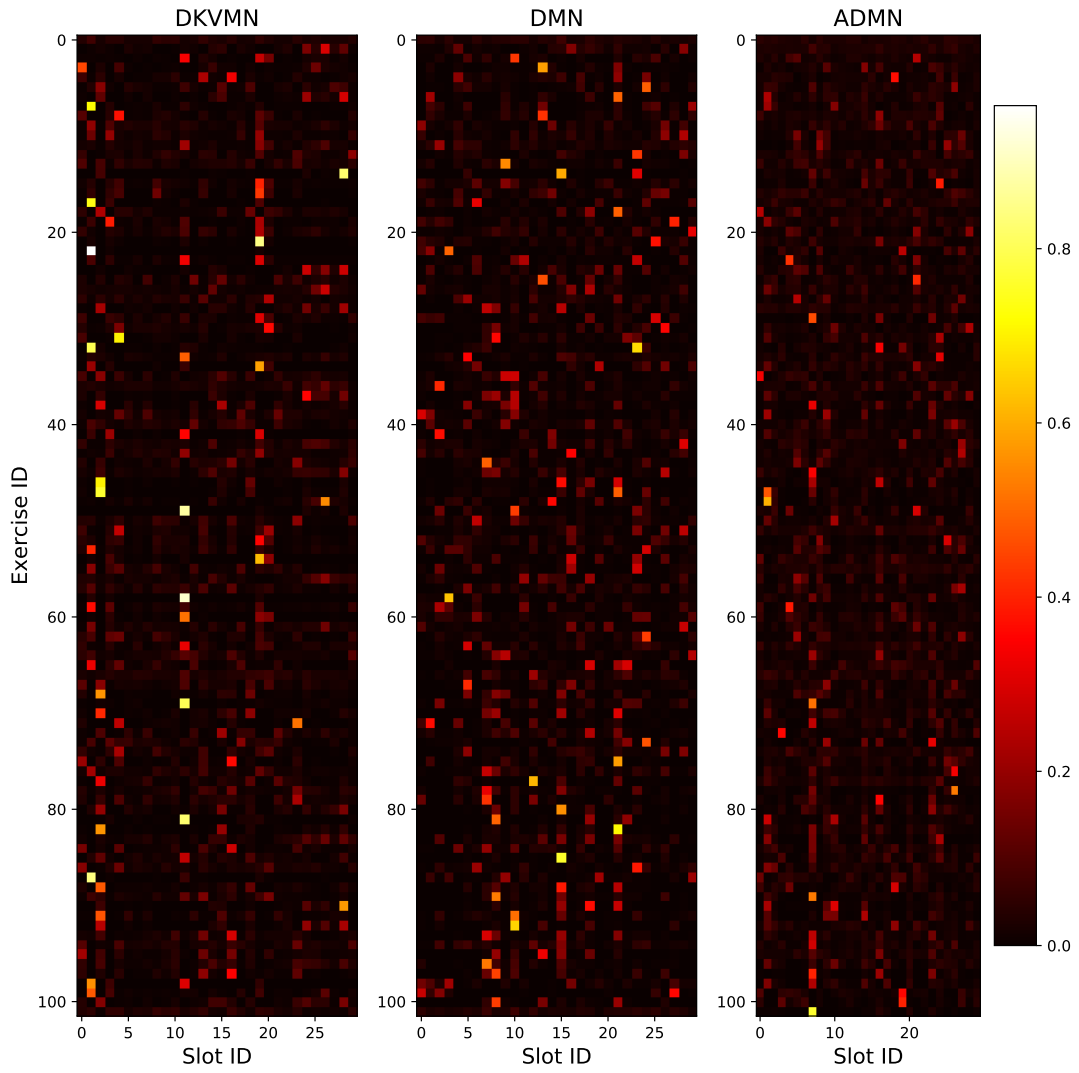

Figure 5.3: Memory slot heatmaps for models trained on ASST2015 with N=30, d = 20.

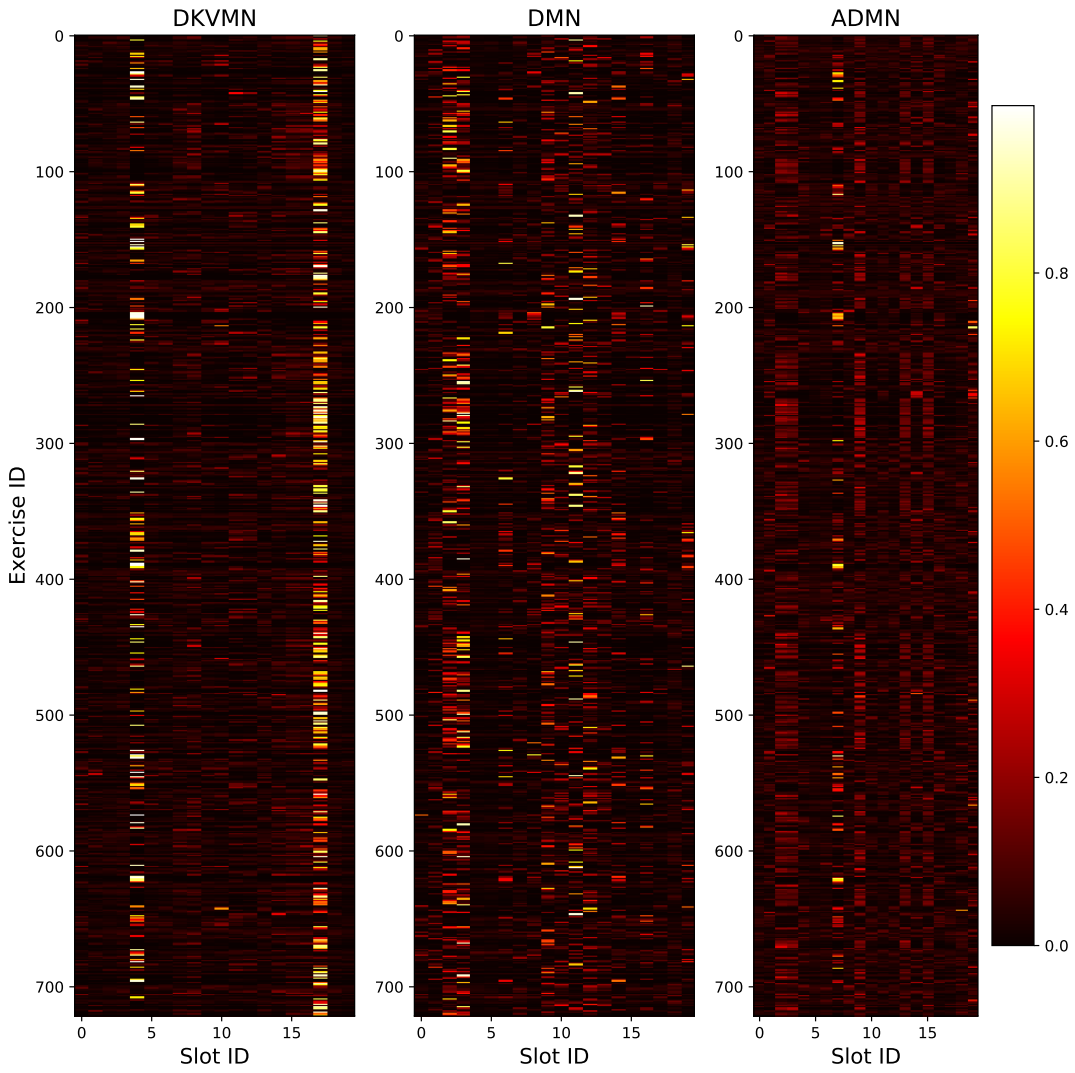

Figure 5.4: Memory slot heatmaps for models trained on JUNYI with N=20, d =40.

In Figures 5.2, 5.3 and 5.4 It can be observed that in the correlations learned by the DKVMN model, exercises are considerably more dense, with the majority of exercises strongly correlated with slots 20-30. Conversely, in the DMN and ADMN model each exercise is correlated with many slots. This could arise due to the fact that the DMN model applies an affine transformation to the correlation weight scaled inputs, whereas DKVMN scales down the result of an activation layer. This means when a DMN or ADMN memory slot is written to with a weight of 0, it is still possible for the slot content to change because of the bias in the activation layer. On the other hand, a DKVMN slot which is written to with a weight of 0 will not change at all. Therefore, the DMN is less incentivised to write exercises strongly to a small number of slots.

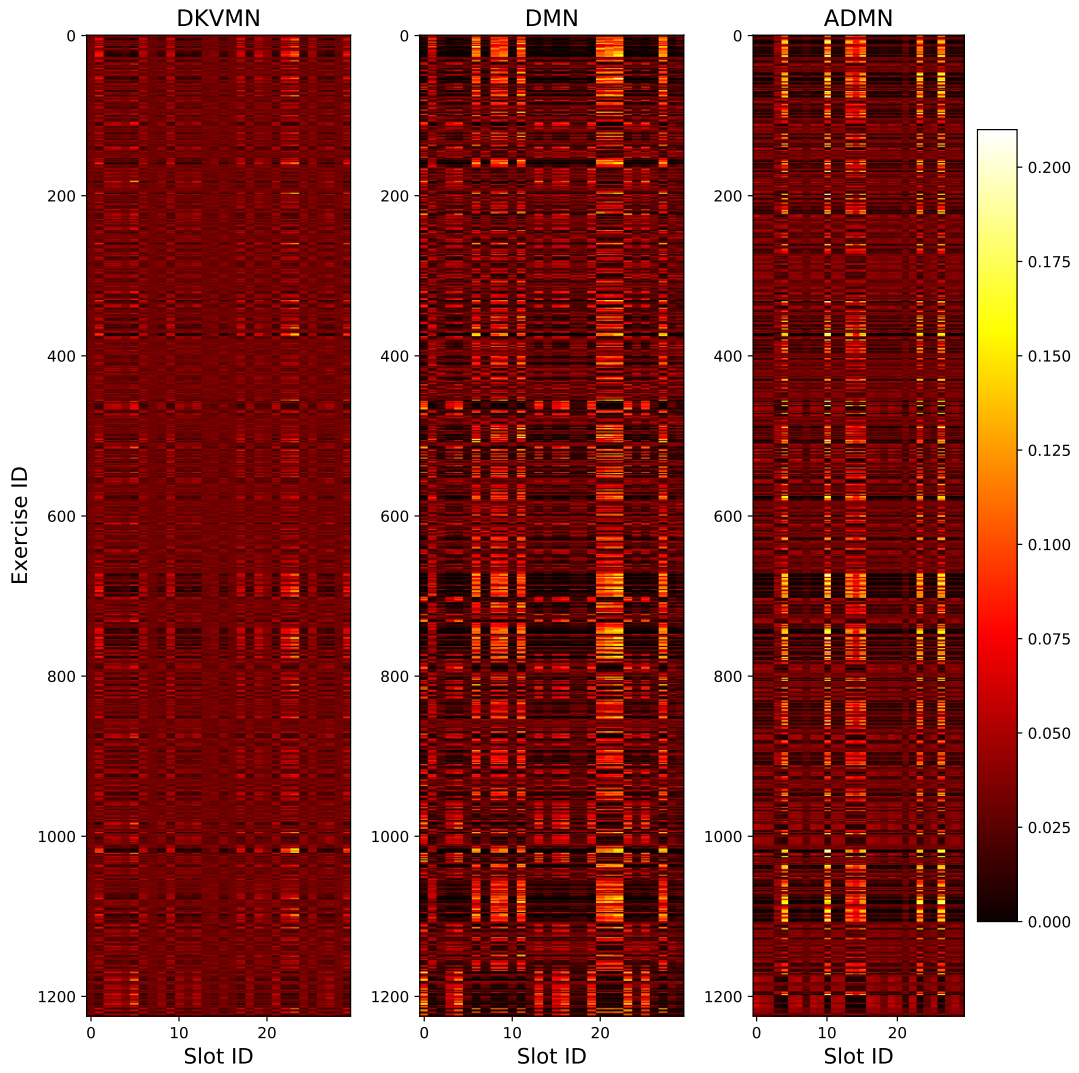

Figure 5.5: Memory slot heatmaps for models trained on STATICS with N=30, $d = 40$.

Interestingly, Figure 5.5 shows somewhat the opposite on STATICS, with DKVMN learning dispersed correlations. Both DMN and ADMN learn dispersed correlations for most exercises, however a lot of exercises have considerably more dense correlations, being mainly condensed into about 10 slots with near zero correlations everywhere else. We also observe that the DMN and ADMN both learned condensed correlations for the *same* exercises, even though they were trained completely independently. The fact that these models opt to only write these exercises to a small portion of the memory slots leads us to believe that these exercises are not very useful in predicting student performance.

5.6 Exercise Clustering

During training, the memory augmented models learn a correlation weight vector for every exercise, and since exercises with similar correlations will share memory slots, the models are encouraged to put similar exercises together in the same slot. This means that the correlation weight vectors can be used to identify clusters of similar exercises, as similar exercises will have similar correlation weights. To verify this we use our DMN model to identify related exercises in the ASST2009 dataset. We choose to use the ASST2009 dataset for two reasons: it has a relatively small number of exercises, which makes displaying all of them simpler, and this dataset contains the names of each exercise (unlike the 2015 version, which only has a number identifier for each exercise).

Figure 5.6: Exercises from the ASST2009 dataset plotted according to correlation weight vectors. A DMN model was trained with $N = 10$ memory slots, and each exercises correlation weight vector was extracted. Vectors were reduced to 2 dimensions using t-SNE, a dimensionality reduction algorithm that maintains original data structure, commonly used to visualize high dimensional data points [Maaten and Hinton 2008]. Exercises are coloured according to which memory slot they have the largest correlation with. e.g. All green exercises have a correlation vector whose largest component is the first component of the vector.

2 Area Irregular Figure	14 Proportion	1 Area Trapezoid
3 Probability of Two Distinct Events	30 Ordering Fractions	11 Histogram as Table or Graph
19 Multiplication Fractions	37 Ordering Positive Decimals	17 Scatter Plot
29 Counting Methods	39 Volume Rectangular Prism	33 Ordering Integers
31 Circumference	45 Algebraic Simplification	34 Conversion of Fraction Decimals Percents
36 Unit Rate	47 Percent Discount	44 Square Root
66 Write Linear Equation from Graph	55 Divisibility Rules	49 Complementary and Supplementary Angles
69 Least Common Multiple	79 Solving Inequalities	52 Congruence
85 Surface Area Cylinder	18 Addition and Subtraction Positive Decimals	58 Solving for a variable
92 Rotations	26 Equation Solving Two or Fewer Steps	61 Estimation
93 Reflection	46 Algebraic Solving	62 Ordering Real Numbers
106 Finding Slope From Situation	59 Exponents	65 Scientific Notation
20 Addition and Subtraction Integers	63 Scale Factor	78 Rate
24 Addition and Subtraction Fractions	80 Unit Conversion Within a System	83 Area Parallelogram
60 Division Fractions	90 Multiplication Whole Numbers	98 Intercept
81 Area Rectangle	91 Polynomial Factors	99 Linear Equations
97 Choose an Equation from Given Information	16 Probability of a Single Event	100 Slope
103 Recognize Linear Pattern	21 Multiplication and Division Integers	101 Angles - Obtuse, Acute, and Right
107 Parts of a Polyomial, Terms, Coefficient, Monomial, Exponent, Variable	23 Absolute Value	104 Simplifying Expressions positive exponents
108 Recognize Quadratic Pattern	38 Rounding	8 Mean
109 Finding Slope From Equation	53 Interior Angles Figures with More than 3 Sides	13 Equivalent Fractions
40 Order of Operations All	71 Angles on Parallel Lines Cut by a Transversal	27 Order of Operations +,-,/,* () positive reals
50 Pythagorean Theorem	73 Prime Number	41 Finding Percents
56 Reading a Ruler or Scale	82 Area Triangle	51 D.4.8-understanding-concept-of-probabilities
64 Surface Area Rectangular Prism	102 Distributive Property	72 Write Linear Equation from Ordered Pairs
67 Percents		75 Volume Sphere
76 Computation with Real Numbers		96 Interpreting Coordinate Graphs
94 Translations		105 Finding Slope from Ordered Pairs
95 Midpoint		4 Table
6 Stem and Leaf Plot		5 Median
9 Range		7 Mode
10 Venn Diagram		15 Fraction Of
12 Circle Graph		22 Addition Whole Numbers
28 Calculations with Similar Figures		25 Subtraction Whole Numbers
35 Percent Of		32 Box and Whisker
48 Nets of 3D Figures		42 Pattern Finding
77 Number Line		43 Write Linear Equation from Situation
86 Volume Cylinder		54 Interior Angles Triangle
87 Greatest Common Factor		57 Perimeter of a Polygon
88 Solving Systems of Linear Equations		68 Area Circle
89 Solving Systems of Linear Equations by Graphing		70 Equation Solving More Than Two Steps
		74 Multiplication and Division Positive Decimals
		84 Effect of Changing Dimensions of a Shape Prportionally
		110 Quadratic Formula to Solve Quadratic Equation

Table 5.3: Exercise ID and corresponding name. Exercises are grouped according the same colouring in Figure 5.6.

From Figure 5.6 and Table 5.3 we can see that the DMN model does indeed discover groups of related exercises. For example, the orange memory slot appears to contain exercises related to basic arithmetic, including q_{20} ="Addition and Subtraction of Integers", q_{24} ="Addition and Subtraction of Fractions" and q_{60} ="Division Fractions". The purple memory slot contains exercises related to ordering of numbers, including q_{30} ="Ordering Fractions", q_{34} ="Ordering Positive Fractions" and q_{79} ="Solving Inequalities". The light blue memory slot contains exercises related to basic geometry, including q_{54} ="Interior Angle Triangle", q_{57} ="Perimeter of a Polygon" and q_68 ="Area Circle".

5.7 Exercise Dependencies

Understanding the dependencies between exercises is vital for designing curriculum and learning plans, both in education institutions and online intelligent tutoring systems. Usually, the task of annotating exercise relationships is done by human experts, and recently [Chang et al. 2015] have proposed a model that learns to predict the relationships between exercises based on human annotations. In this way, a small collection of exercise relationships can be manually annotated, and then the rest can be inferred by their model. However, our proposed IDMN model learns separate read and write weights for each exercise, which can then be used to infer dependencies between exercises. This allows to automatically discover exercise relationships without any human annotation, in an end-to-end way. In our IDMN model, any given memory slot will contain a summary of the student's mastery of all the exercises which write to that slot. Then each slot can be viewed as containing information from a subset of exercises. Then the model should learn to, for a given input exercise q, read from memory slots which contain information about q's dependent exercises. In this setup it is possible for q to read from the memory slots of its dependent exercises, however it is not necessary for those dependent exercises to read from q's slot. We therefore calculate the dependency of q_1 on q_2 as

$$d(q_1, q_2) = \mathbf{w}_{q_1}^{r\ T} \mathbf{w}_{q_2}^{w} \tag{5.1}$$

Intuitively, for each slot we multiply the weight with which q_2 writes to the slot and the weight with which q_1 reads from the slot, then sum over all slots.

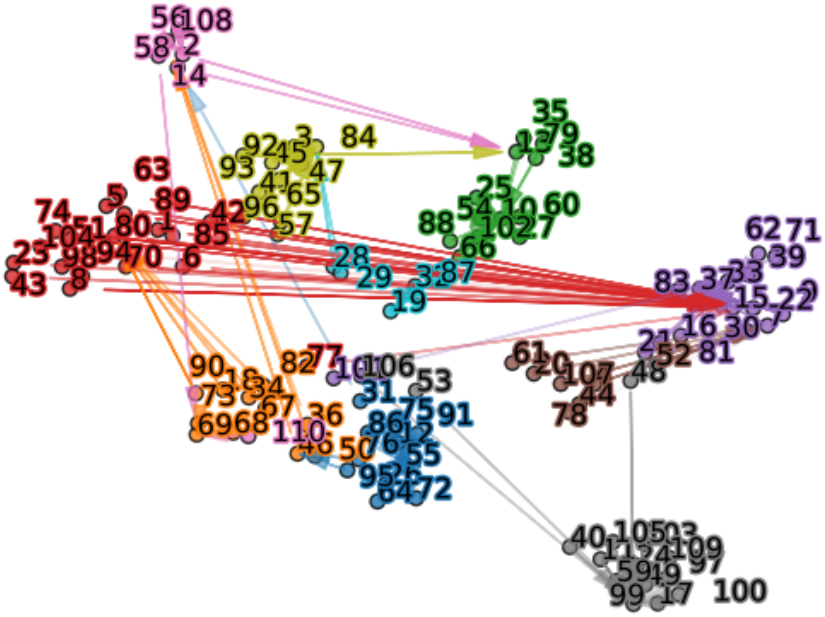

Figure 5.7: ASSISTments2009 exercise dependencies as learned from the IDMN model with $N = 30$ slots and $d = 60$. As in Figure 5.6, exercises are plotted and coloured according to their read correlation weight vectors. For each exercise q, an arrow is drawn from q to the exercise for which q has the greatest dependency, as calculated in eq. (5.1). The arrows are shaded according to the value of the dependency, with transparent arrows representing small dependencies and opaque arrows representing large dependencies.

12 Circle Graph	7 Mode	11 Histogram as Table or Graph
26 Equation Solving Two or Fewer Steps	9 Range	17 Scatter Plot
31 Circumference	15 Fraction Of	24 Addition and Subtraction Fractions
55 Divisibility Rules	16 Probability of a Single Event	40 Order of Operations All
64 Surface Area Rectangular Prism	21 Multiplication and Division Integers	48 Nets of 3D Figures
72 Write Linear Equation from Ordered Pairs	22 Addition Whole Numbers	49 Complementary and Supplementary Angles
75 Volume Sphere	30 Ordering Fractions	53 Interior Angles Figures with More than 3 Sides
76 Computation with Real Numbers	33 Ordering Integers	59 Exponents
86 Volume Cylinder	37 Ordering Positive Decimals	97 Choose an Equation from Given Information
91 Polynomial Factors	39 Volume Rectangular Prism	99 Linear Equations
95 Midpoint	62 Ordering Real Numbers	100 Slope
18 Addition and Subtraction Positive Decimals	71 Angles on Parallel Lines Cut by a Transversal	103 Recognize Linear Pattern
34 Conversion of Fraction Decimals Percents	81 Area Rectangle	105 Finding Slope from Ordered Pairs
36 Unit Rate	83 Area Parallelogram	106 Finding Slope From Situation
46 Algebraic Solving	101 Angles - Obtuse, Acute, and Right	109 Finding Slope From Equation
50 Pythagorean Theorem	4 Table	3 Probability of Two Distinct Events
67 Percents	20 Addition and Subtraction Integers	41 Finding Percents
68 Area Circle	44 Square Root	45 Algebraic Simplification
69 Least Common Multiple	52 Congruence	47 Percent Discount
73 Prime Number	61 Estimation	57 Perimeter of a Polygon
82 Area Triangle	78 Rate	65 Scientific Notation
90 Multiplication Whole Numbers	107 Parts of a Polyomial, Terms, Coefficient, Monomial, Exponent, Variable	84 Effect of Changing Dimensions of a Shape Prportionally
10 Venn Diagram	2 Area Irregular Figure	92 Rotations
13 Equivalent Fractions	14 Proportion	93 Reflection
25 Subtraction Whole Numbers	56 Reading a Ruler or Scale	96 Interpreting Coordinate Graphs
27 Order of Operations +,-,/,* () positive reals	58 Solving for a variable	19 Multiplication Fractions
35 Percent Of	108 Recognize Quadratic Pattern	28 Calculations with Similar Figures
38 Rounding	110 Quadratic Formula to Solve Quadratic Equation	29 Counting Methods
54 Interior Angles Triangle		32 Box and Whisker
60 Division Fractions		87 Greatest Common Factor
66 Write Linear Equation from Graph		
79 Solving Inequalities		
88 Solving Systems of Linear Equations		
102 Distributive Property		
1 Area Trapezoid		
5 Median		
6 Stem and Leaf Plot		
8 Mean		
23 Absolute Value		
42 Pattern Finding		
43 Write Linear Equation from Situation		
51 D.4.8-understanding-concept-of-probabilities		
63 Scale Factor		
70 Equation Solving More Than Two Steps		
74 Multiplication and Division Positive Decimals		
77 Number Line		
80 Unit Conversion Within a System		
85 Surface Area Cylinder		
89 Solving Systems of Linear Equations by Graphing		
94 Translations		
98 Intercept		
104 Simplifying Expressions positive exponents		

Table 5.4: Exercise ID and corresponding name. Exercises are grouped according the same colouring in Figure 5.7.

From Figure 5.7 and Table 5.4 it can be seen that all of the red exercises, including q_8 ="Mean", q_{80} ="Unit Conversion Within a System" and q_{63} ="Scale Factor" are strongly dependent on exercise q_{15} ="Fraction of". This dependency is clearly one directional, as none of the purple exercises have strong dependencies on any other

exercises. This shows that, as we hypothesized, there do exist strong directed dependencies between exercises, and our model is capable of automatically discovering them.

Chapter 6

Conclusion

In this work we have explored two novel neural network architectures, DMN and ADMN, which exploit assumptions specific to the task of KT. Our experiments show that these specialised models which are built for KT achieve better performance than all previously published KT models, making them state of the art. Furthermore, we have shown how they can be used to discover groups of similar exercises, and dependencies between exercises, automatically from interaction logs, a task that is currently performed by human experts. We have shown that tailoring deep neural network models for the task at hand can significantly improve predictive capabilities. Perhaps even more importantly, we have demonstrated how we can use these models to capture specific information that is useful and interpretable for humans. There are still many areas where KT models are lacking, but we hope that by continuing to improve our methods it will be possible to create ITS which can truly deliver personalized learning plans for each individual students needs.

6.1 Future Works

While our work has improved the capability of state of the art KT, there are still many areas which could be improved in order to produce ITS which are even more beneficial to enhancing student learning.

6.1.1 Improving Accumulative Distributed Memory Networks

We found that while our ADMN model improved performance compared to DMN on some datasets, on other datasets which did not have repetitive exercise occurrences, ADMN did not perform better. This is because we simply check if the current exercise is the same as previous exercise to decide when to write to memory. However, there exist more sophisticated methods for creating hierarchical memory models. For example, [Chung et al. 2016] show how a hierarchical LSTM can learn to produce a discrete binary variable that decides when one layer will pass its hidden state into a higher layer. Similarly, it should be possible to create a variation of ADMN that learns when to write the accumulated state to external memory, instead of simply checking if the current exercise matches the previous exercise.

6.1.2 Personalization Parameters

Recently, personalization parameters have been shown to improve the performance of recurrent neural network models for recommendation systems [Donkers et al. 2017]. These models learn a vector embedding of every individual user in a dataset, which is incorporated as input when making recommendations for a user. This allows them to learn about a users personal tastes and respond accordingly. It seems to us that such a model should also be beneficial for the task of KT.

6.1.3 Incorporating External Topic Knowledge

For most common education topics, there exists vast amounts of readily available expert knowledge on the internet. For example, [Wilson and Nichols 2015] have used knowledge graphs, which describe the relationships between various topics of educational exercises, as data for a KT model. In the future we would like to investigate ways of incorporating knowledge graphs and similar forms of external topic information into our KT models.

6.1.4 Dynamic Memory Addressing

In our proposed models, the correlation weights which are used to read and write from memory are computed solely based on the current exercise. This is done so that the relationships between exercises can be inferred from looking at which memory slots an exercise is put in. If the addressing weights were computed from a dynamic memory content, then it would become extremely difficult to interpret the meaning of each memory slot. Nevertheless, [Gulcehre et al. 2016] have shown that a dynamic addressing scheme can improve model performance. We would like to investigate a method to allow the model to dynamically choose which memory slots to read and write to for each exercise, while still maintaining model interpretability.

6.1.5 Using Detailed Answers as Input Data

Since its inception nearly 25 years ago, most of the research effort in KT has been dedicated to modelling student interactions with binary outcomes: the student is either successful or unsuccessful. However, in practice there are many possible answers that a student can give for an exercise, and there are many different ways to be wrong. Currently, KT models do not use the exact answer a student gives as input because it is extremely difficult to convert a students raw textual answer into a vector input. Nevertheless, in the future we would like to investigate ways to make use of a detailed description of the precise *answer* a student gives as input for a KT model. We believe this would not only improve the predictive capability of KT models, but also significantly improve the ability for KT models to give personalized feedback and instruction, something which ITS are currently incapable of.

Bibliography

ALCANTARA, G. 2017. Empirical analysis of non-linear activation functions for deep neural networks in classification tasks. *arXiv preprint arXiv:1710.11272*. (p. 10)

AMARI, S.-I. 1993. Backpropagation and stochastic gradient descent method. *Neurocomputing 5*, 4-5, 185–196. (p. 10)

BENGIO, Y., BOULANGER-LEWANDOWSKI, N., AND PASCANU, R. 2013. Advances in optimizing recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on* (2013), pp. 8624–8628. IEEE. (p. 25)

BENGIO, Y., SIMARD, P., AND FRASCONI, P. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks 5*, 2, 157–166. (p. 11)

BOJANOWSKI, P., GRAVE, E., JOULIN, A., AND MIKOLOV, T. 2016. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*. (p. 16)

CHANG, H.-S., HSU, H.-J., AND CHEN, K.-T. 2015. Modeling exercise relationships in e-learning: A unified approach. In *EDM* (2015), pp. 532–535. (pp. 24, 36)

CHUNG, J., AHN, S., AND BENGIO, Y. 2016. Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704*. (pp. 21, 41)

CHUNG, J., GULCEHRE, C., CHO, K., AND BENGIO, Y. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*. (pp. 12, 13)

COLLINS, A. AND HALVERSON, R. 2018. *Rethinking education in the age of technology: The digital revolution and schooling in America*. Teachers College Press. (p. 1)

CORBETT, A. T. AND ANDERSON, J. R. 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User modeling and user-adapted interaction 4*, 4, 253–278. (pp. 2, 3, 5)

DESMET, W. 2002. Mid-frequency vibro-acoustic modelling: challenges and potential solutions. In *Proceedings of ISMA*, Volume 2 (2002), pp. 12. Citeseer. (p. 12)

DONKERS, T., LOEPP, B., AND ZIEGLER, J. 2017. Sequential user-based recurrent neural network recommendations. In *Proceedings of the Eleventh ACM Conference on Recommender Systems* (2017), pp. 152–160. ACM. (pp. 7, 42)

DUNN, R., BEAUDRY, J. S., AND KLAVAS, A. 2002. Survey of research on learning styles. *California Journal of Science Education 2*, 2, 75–98. (p. 1)

FENG, M., HEFFERNAN, N., AND KOEDINGER, K. 2009. Addressing the assessment challenge with an online system that tutors as it assesses. *User Modeling and User-Adapted Interaction 19*, 3, 243–266. (p. 23)

GLOROT, X. AND BENGIO, Y. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (2010), pp. 249–256. (p. 24)

GRAVES, A., WAYNE, G., AND DANIHELKA, I. 2014. Neural turing machines. *arXiv preprint arXiv:1410.5401*. (pp. 15, 19)

GULCEHRE, C., CHANDAR, S., CHO, K., AND BENGIO, Y. 2016. Dynamic neural turing machine with soft and hard addressing schemes. *arXiv preprint arXiv:1607.00036*. (pp. 19, 42)

HALLINAN, B. AND STRIPHAS, T. 2016. Recommended for you: The netflix prize and the production of algorithmic culture. *New Media & Society 18*, 1, 117–137. (p. 1)

HARDLE, W. AND MAMMEN, E. 1993. Comparing nonparametric versus parametric regression fits. *The Annals of Statistics*, 1926–1947. (p. 9)

HARTLEY, J. 2008. Youtube, digital literacy and the growth of knowledge. (p. 1)

HENAFF, M., SZLAM, A., AND LECUN, Y. 2016. Recurrent orthogonal networks and long-memory tasks. *arXiv preprint arXiv:1602.06662*. (p. 11)

HOCHREITER, S. AND SCHMIDHUBER, J. 1997. Long short-term memory. *Neural computation 9*, 8, 1735–1780. (p. 13)

J. WESTON, A. B., S. CHOPRA. 2015. Memory networks (2015). (pp. 14, 15)

KINGMA, D. P. AND BA, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. (p. 24)

KOEDINGER, K. R., BAKER, R. S., CUNNINGHAM, K., SKOGSHOLM, A., LEBER, B., AND STAMPER, J. 2010. A data repository for the edm community: The pslc datashop. *Handbook of educational data mining 43*. (p. 24)

KOREN, Y., BELL, R., AND VOLINSKY, C. 2009. Matrix factorization techniques for recommender systems. *Computer 42*, 8. (p. 7)

LIANG, Z., XIONG, X., ZHAO, S., BOTELHO, A., AND HEFFERNAN, N. T. 2017. Incorporating rich features into deep knowledge tracing. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale* (2017), pp. 169–172. ACM. (p. 6)

LING, C. X., HUANG, J., ZHANG, H., ET AL. 2003. Auc: a statistically consistent and more discriminating measure than accuracy. In *IJCAI*, Volume 3 (2003), pp. 519–524. (p. 25)

LU, J., WU, D., MAO, M., WANG, W., AND ZHANG, G. 2015. Recommender system application developments: a survey. *Decision Support Systems 74*, 12–32. (p. 1)

MAATEN, L. V. D. AND HINTON, G. 2008. Visualizing data using t-sne. *Journal of machine learning research 9*, Nov, 2579–2605. (p. 34)

MIKOLOV, T., KARAFIÁT, M., BURGET, L., ČERNOCKỲ, J., AND KHUDANPUR, S. 2010. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association* (2010). (p. 11)

NEY, H., ESSEN, U., AND KNESER, R. 1994. On structuring probabilistic dependences in stochastic language modelling. *Computer Speech & Language 8*, 1, 1–38. (p. 12)

NWANA, H. S. 1990. Intelligent tutoring systems: an overview. *Artificial Intelligence Review 4*, 4, 251–277. (pp. 1, 2)

PASCANU, R., MIKOLOV, T., AND BENGIO, Y. 2013. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning* (2013), pp. 1310–1318. (p. 24)

PIECH, C., BASSEN, J., HUANG, J., GANGULI, S., SAHAMI, M., GUIBAS, L. J., AND SOHL-DICKSTEIN, J. 2015. Deep knowledge tracing. In *Advances in Neural Information Processing Systems* (2015), pp. 505–513. (p. 6)

PIECH, C. J. 2016. *Uncovering Patterns in Student Work: Machine Learning to Understand Human Learning*. PhD thesis, Stanford University. (p. 3)

QUADRANA, M., KARATZOGLOU, A., HIDASI, B., AND CREMONESI, P. 2017. Personalizing session-based recommendations with hierarchical recurrent neural networks. In *Proceedings of the Eleventh ACM Conference on Recommender Systems* (2017), pp. 130–137. ACM. (p. 20)

SAJIB, M. S. R., MALIK, M. A. I., ISLAM, M. A., AND HALDER, S. K. 2018. Video recommendation system for youtube considering users feedback. *Global Journal of Computer Science and Technology*. (p. 1)

SPECHT, D. F. 1991. A general regression neural network. *IEEE transactions on neural networks 2*, 6, 568–576. (p. 9)

STEIF, P. AND BIER, N. Oli engineering statics-fall 2011, february 2014. *URL https://pslcdatashop. web. cmu. edu/DatasetInfo*. (p. 24)

TANG, D., WEI, F., YANG, N., ZHOU, M., LIU, T., AND QIN, B. 2014. Learning sentiment-specific word embedding for twitter sentiment classification. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Volume 1 (2014), pp. 1555–1565. (p. 16)

THOMPSON, C. 2011. How khan academy is changing the rules of education. *Wired Magazine 126*, 1–5. (p. 2)

WILSON, K. AND NICHOLS, Z. 2015. The knewton platform. *A General-Purpose Adaptive Learning Infrastructure, Knewton*. (p. 42)

WU, C.-Y., AHMED, A., BEUTEL, A., SMOLA, A. J., AND JING, H. 2017. Recurrent recommender networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining* (2017), pp. 495–503. ACM. (p. 7)

XIONG, X., ZHAO, S., VAN INWEGEN, E., AND BECK, J. 2016. Going deeper with deep knowledge tracing. In *EDM* (2016), pp. 545–550. (pp. 6, 23)

YUDELSON, M. V., KOEDINGER, K. R., AND GORDON, G. J. 2013. Individualized bayesian knowledge tracing models. In *International Conference on Artificial Intelligence in Education* (2013), pp. 171–180. Springer. (p. 5)

ZAREMBA, W. AND SUTSKEVER, I. 2014. Learning to execute. *arXiv preprint arXiv:1410.4615*. (p. 14)

ZHANG, F., YUAN, N. J., LIAN, D., XIE, X., AND MA, W.-Y. 2016. Collaborative knowledge base embedding for recommender systems. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining* (2016), pp. 353–362. ACM. (p. 6)

ZHANG, J., SHI, X., KING, I., AND YEUNG, D.-Y. 2017. Dynamic key-value memory networks for knowledge tracing. In *Proceedings of the 26th International Conference on World Wide Web* (2017), pp. 765–774. International World Wide Web Conferences Steering Committee. (pp. 3, 6, 15, 17, 19, 23, 25, 26, 28)