Master Thesis Econometrics & Management Science

Predicting Purchase Decisions Using Autoencoders in a High-Dimensional Setting

Luuk van Maasakkers (414156)

Supervisors:

prof. dr. D. Fok

prof. dr. B. Donkers

December 19, 2018

Erasmus University Rotterdam

Erasmus School of Economics

Abstract

To be able to serve customers with personalized product recommendations or shopping lists, it is important for retailers to accurately predict the purchase behaviour of their customers. In this study, we propose an autoencoder-based model for next basket prediction. The goal of next basket prediction is to accurately predict the set of items a customer will buy at his next moment of purchase. Our model consists of three main steps. First, we train an autoencoder, which encodes high-dimensional baskets into low-dimensional codes. Next, this code is used to predict the code of the next basket, which is decoded in the third step to obtain basket predictions. Our model clearly outperforms the benchmark model in a wide range of performance measure and is able to deal with high dimensionality of the retailer's assortment.

Contents

1 Introduction

For retailers, it is an important challenge to predict the items contained in the next basket that a customer will purchase. Often the only information is the sequence of earlier purchased baskets. *Next basket prediction* offers retailers the opportunity to develop more effective marketing strategies, e.g. serve customers with individual-specific product recommendations or personalized promotions. By doing so, a retailer can improve customer satisfaction and stimulate customers' purchases, resulting in increased future sales. However, the underlying factors influencing a customer's decision process are numerous and can be complex to capture.

Before continuing, note that there is a difference between a product recommendation and a basket recommendation. In the latter case, multiple products are simultaneously recommended to a customer. Basket recommendations are particularly useful in situations where a customer makes regular, recurring purchases, e.g. in a supermarket or grocery. Recommending an entire basket of products can be seen as a service to the customer, as the customer can now fill his usual basket with one click. In contrast, single product recommendations can have a lot of other purposes as well, such as helping the customer to discover new products. Product and basket recommendations are closely related, but depending on the situation, most often either of the two is preferred. In this study, we focus on basket recommendations, i.e. simultaneously recommending a set of multiple products.

A *basket* is defined as a set of products from a retailer's assortment, bought by a certain customer at the same point in time. In this study, we consider the case of a customer purchasing a series of baskets at different points in time. Co-occurency, sequentiality, periodicity and recurrency of purchased items are all relevant factors that characterize the sequence and contents of a basket series. Another aspect that complicates the prediction process is the dimensionality of the retailer's assortment. Nowadays, online retailers can have hundreds, thousands or even millions of items in their assortment. Most standard techniques are unreliable or computationally very expensive when there are so many items that must be taken into account in the decision process of a customer.

In this study, we propose a three-step model to predict next purchase baskets in a high-dimensional setting. In the first step, the high-dimensional basket is encoded in a low-dimensional *code*. This *code* is used to predict the basket code at the next point in time in the second step. This new basket code is decoded in step three, resulting in a predicted basket for the next point in time. The encoding and decoding is done using an

autoencoder. The second step can be separated from the autoencoder, e.g. by estimating a vector autoregression model based on the codes summarizing the baskets. We also consider a model where the second step is integrated with the autoencoder, resulting in a new neural network where weights are optimized to map the codes nonlinearly to their successive baskets. To test the model, we consider customers from an American online grocery store, *Instacart*, whose purchase behaviour over the past year is known (*The Instacart Online Grocery Shopping Dataset*, 2017). Based on the previous basket of a customer, we aim to predict the products that will be purchased next, regardless of the time between last and next order.

The problem in this study shares some common ground with market basket analysis (MBA). However, it is not a classical MBA problem. MBA is an example of affinity analysis and association rule learning, as it aims to uncover relationships between items purchased by customers. In particular, it aims to find combinations of products that frequently co-occur in orders based on basket data. This part of MBA is captured in the autoencoder, as the code layer should capture the relationships between items. However, the main aim of the research is different from MBA, as we want to predict the next basket based on earlier baskets. Standard MBA aims to predict which other items will occur in a basket, given a set of purchased items in the same basket. Although capturing co-occurrences is not our main goal, we will investigate whether the trained autoencoders are capable of capturing co-occurence patterns.

A few recent studies (Rendle et al., 2010; Wang et al., 2015; Yu et al., 2016; Guidotti et al., 2017) have developed next basket prediction models. The approaches in these studies are very different, ranging from personalized Markov chains to recurrent neural networks. The common ground in these studies is that they capture sequential patterns, general user taste and/or co-occurence of products in their predictions. We extend the field of next basket prediction by using an autoencoder-based approach.

We discuss the current literature on this topic in more detail in the next section. Afterwards, we formally formulate the problem, explain the proposed techniques and discuss the learning process. Next, we apply our model to the Instacart dataset. Finally, we present the results and draw conclusions.

2 Literature review

Next basket prediction is a relatively new topic in the literature. A possible reason for this is the recent growth in the usage of web shops by consumers, and, as a result of that, the

increased relevance of the topic and growth in data availability. With customer accounts and loyalty cards, shops can nowadays observe series of many baskets for each customer and try to use this information to their advantage. Besides that, computational advances in recent decades have made it possible to estimate the parameters of complex models that can deal with the high dimensionality of basket data. The dimensionality of basket data is usually high due to the large number of products in a store's assortment, which is especially the case for online shops.

Studies in the field of next basket prediction aim to predict the basket of a certain user at time t based on the purchased baskets at time $t-1$, $t-2$, etc. Note that the time points are not absolute, but relative regarding a user (e.g. his first, second, etc. basket). In general, we can distinguish three components on which these models are based: *general user taste*, *sequential behavior* and *item co-occurence patterns*. In some papers, only the first two components are considered. We also consider the third, as it is specific to the problem of next basket prediction (compared to predicting or recommending single items). A model based on general user taste aims to learn general preferences of a customer and make basket predictions according to this learned taste. For example, someone who has purchased mostly sports clothes in the past will be expected to purchase sports clothes next because it corresponds to his general taste. On the other hand, a model based on sequential behavior aims to find common sequential patterns between consecutive baskets in the data and make predictions according to these patterns. For example, someone who just bought a new computer will be expected to buy computer accessories (e.g. screen, mouse, speakers) next. Finally, a pattern-based model aims to find frequently occurring itemsets in the data, regardless of sequential information. For example, someone who is expected to buy paint is likely to buy a paint brush as well. Note that the three components do not exclude each other and can actually complement each other. Whereas some studies focus on one specific component, others combine multiple in their model. In this section, we discuss a few promising models from recent studies and relate them to our proposed model.

Rendle et al. (2010) propose a factorized personalized Markov chain (FPMC) for next basket prediction. The model combines matrix factorization (MF, to capture general taste of users) and Markov chains (MC, to model sequential behavior) into a Markov chain with user-specific transitions. In this way, sequential data is captured by the transition matrix and user-taste is captured in the user-specific transition rates. As the data in the next basket prediction context is typically sparse (only a small proportion of all products in the assortment is actually bought by a customer each time), standard estimation approaches such as maximum likelihood do not result in good estimates of the personalized transition matrix. Instead, the researchers propose a factorization model for estimation, where each

transition is influenced by transitions of similar users, similar items and similar transitions. As the state space of the Markov chain can become very large in this problem (2^N, where N is the number of products in the assortment), only Markov chains of length 1 are considered and transition probabilities are simplified. The researchers show that the proposed model outperforms standard MF and MC models on an online drug store data set in terms of the accuracy of top ranked products predicted for each customer.

One problem in the FPMC model is that all components of the model are linearly combined. This implicit linearity assumption can be too restrictive in some cases, as argued by Wang et al. (2015), who propose a hierarchical representation model (HRM). In this two-layer model, items and users are represented by latent vectors of equal size, describing characteristics of the items. In the first layer, items contained in the last purchased basket are aggregated to obtain a pooled basket vector. In the second layer, the user vector (which is based on the entire purchase history of a user) and pooled basket vector of the last transaction are aggregated to obtain a new latent vector, which can be used directly to predict items in the next basket. To introduce nonlinearity in the model, the pooling operation in each of the layers can be chosen to be nonlinear (e.g. *max pooling*, which takes for each element the maximum over all individual vectors). It can be shown that by choosing specific pooling operations and optimization criterions, the HRM reduces to a Markov chain model, matrix factorization model or the FPMC model. Therefore, the model is very general. It is shown that by using *max pooling* in both layers, the HRM can outperform baseline MC, MF and FPMC models.

Yu et al. (2016) describe a dynamic recurrent model (DREAM) based on a recurrent neural network. Similar to the HRM, items are described by latent vectors, which are pooled to obtain a basket latent vector. Next, these basket-representing latent vectors of a user's purchase history are mapped to score vectors through non-linear operations in the recurrent neural network, where each element of the score vector represents one item. A high score indicates that the user is more likely to purchase the corresponding item. The network contains one hidden layer with sigmoid activation function, which can be interpreted as a dynamic representation of the user, as it is based on all earlier purchased baskets and gets updated when a new basket is added. On the other hand, the recurrency of the network takes the sequential behavior of users into account. The researchers show that DREAM outperforms other models such as FPMC and HRM.

Finally, we consider the Temporal Annotated Recurring Sequences (TARS) model by Guidotti et al. (2017). In contrast to the previously discussed studies, the researchers aim to construct a model that is readable and interpretable by humans. The model is mainly

based on item co-occurence and sequentiality. A sequence, as defined by the researchers, consists of two sets of items, where the first set is contained in a basket purchased earlier than the basket containing the second set. A recurring sequence is defined as a sequence that occurs at least a certain amount of times in the purchase history. The temporal annotated recurring sequences can be used to make predictions of new baskets, by taking into account all recurring sequences that are potentially active at the next purchase time. Predictions are personalized and user-specific, meaning that the predictions for a certain customer depend on a model that is built based on only the purchase history of that customer (which is in line with the user-centric vision for data prediction, as noted by the researchers). It is shown that TARS predictions can outperform baseline models, including the three models discussed earlier.

Most of the above studies combine multiple purchase history components (general user taste, sequentality, co-occurence patterns) in their predictions. In our proposed model, we mainly focus on sequential behavior and co-occurence patterns. Although the code layer in the autoencoder can be interpreted as a dynamic user taste at a specific point in time, it does not depend on earlier purchases and is therefore not general. By using autoencoders with multiple hidden layers to summarize baskets, our model allows for highly nonlinear item co-occurence patterns. In this sense, our model is very similar to the DREAM model. In contrast, the FPMC model only allows for linear combinations. In contrast to HRM and DREAM, our model takes raw baskets represented by high-dimensional dummy vectors as inputs. As a consequence, dimensionality and sparsity issues have to be taken into account during training. Finally, some of the earlier mentioned models (e.g. FPMC) are designed to recommend new (i.e. not earlier bought) items to customers. We predict new baskets regardless of whether items are bought before by that customer or not. Each of these two approaches has its practical relevance. For example, for a Netflix user it does not make sense to recommend a series he already watched, as the user probably will not watch the series again. For these types of companies, recommendations are mainly used to help users discover new products. On the other hand, a supermarket customer usually buys the same products repeatedly and does not always have direct need to discover new products. For these types of shops, with regular purchases, recommendations mainly serve as a help to the user to save time and fill their baskets quickly with desired products. Our approach mainly focuses on the second case.

3 Proposed model

In this section, we first mathematically formulate the considered problem of next basket prediction. Afterwards, we describe our proposed approach. In short, our approach

contains three steps. First, we summarize basket data in relatively few dimensions using an autoencoder. We also make some remarks on how the autoencoder can be trained effectively. The goal of this part is to reduce the dimensionality of the baskets, while preserving as much information as possible from the original baskets. Most standard techniques and models can not deal with the high dimensionality of the original baskets, but are well applicable to the low-dimensional basket representations. Secondly, after training the autoencoder, we map the codes on their successors. Thirdly, the predicted codes are decoded to obtain basket predictions. We propose several variations of the model, varying in flexibility and complexity.

3.1 Problem formulation

In the general scenario of next basket predictions, we consider a large set of C customers, each purchasing a series of baskets over time. At each moment of purchase, a customer selects a number of products from an assortment of N products. Together, the chosen products form the basket purchased at that moment in time. The basket of customer c purchased at time t is represented by an N-dimensional dummy vector $\mathbf{b}_t^c$, where the n^{th} element b_{tn}^c is 1 if the n^{th} product in the assortment is purchased at time t and 0 otherwise. Usually, these vectors are very sparse, meaning that only a small number of products is purchased compared to the total number of products in the assortment. The sum of each basket vector is also the number of products contained in that basket. Our goal is to predict the basket of customer c at time t, $\mathbf{b}_t^c$, based on his previously purchased baskets $\mathbf{b}_{t-1}^c$. Note that time is relative in this notation, i.e. only the order of basket matters and not the absolute (inter)purchase times. The length of the observed basket sequence can also be different across customers. We define $\mathcal{B}$ as the set of all observed baskets.

3.2 Basket encoding

The baskets are characterized by their sparsity and high dimensionality. Most models are not designed for sparse data and high dimensionality often causes computational problems. To avoid these problems, we first summarize the basket vectors into low-dimensional vector representations by using an autoencoder. An autoencoder is a feedforward neural network that maps an input to itself through a small (i.e. low-dimensional) hidden layer. As the literature on autoencoders is diverse and terminology has evolved over time, it is difficult to attribute the ideas about autoencoders to specific researchers. In the late 1980s, the terms *auto-associative neural network* and *identity mapping* were first used. In more recent research, the term *autoencoder* is more common.

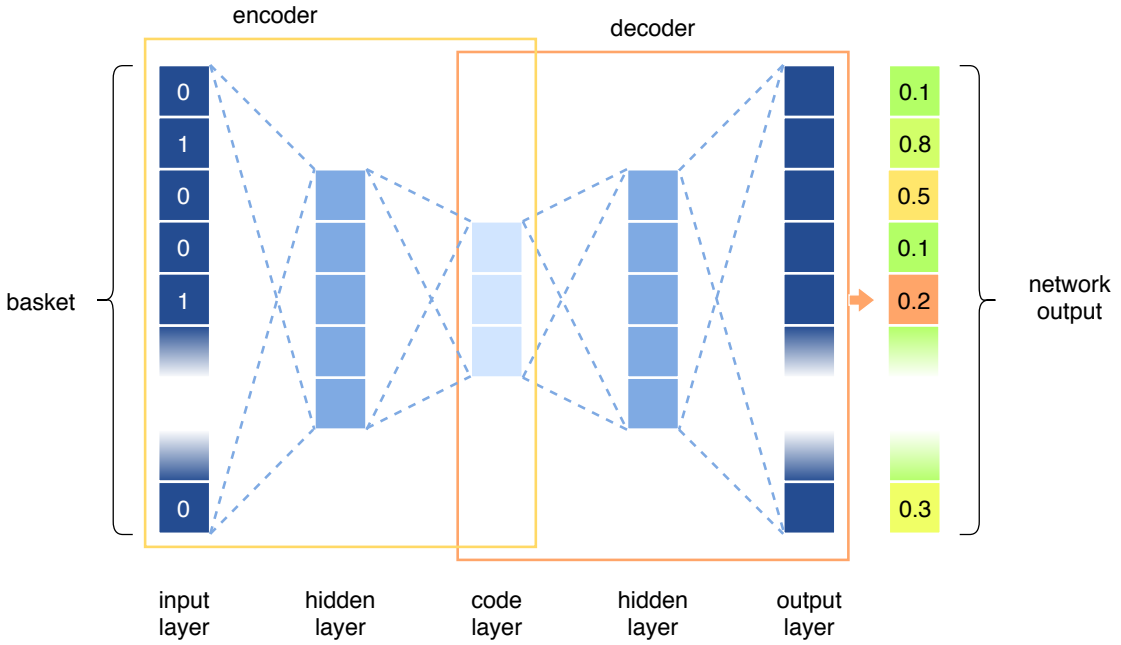


Figure 1: A simple autoencoder with three hidden layers. The smallest hidden layer is called the code layer. The first part of the network encodes the input vector into the code layer, the second part of the network decodes the code layer representation into an estimated output. In each node, except the input nodes, a non-linear operation is performed (see Figure 2).

3.2.1 Autoencoders

Ackley et al. (1985) introduced the encoder problem, where two sets of units are connected through a small set of hidden units. This set of hidden units may act as a limited capacity bottleneck through which information about the two sets must be squeezed. Rumelhart et al. (1985) presented a similar problem, but this time the input and output units were the same. That is, the required mapping is the identity mapping. The hidden units yield useful internal representations that give explicit information about the structure of the input patterns. Baldi & Hornik (1989) proved that a standard quadratic error loss function for an autoencoder with one hidden layer of linear units has one unique minimum, with optimal weights corresponding to the principal components of the inputs. DeMers & Cottrell (1993) described an autoencoder with additional hidden layers and non-linear activation functions. These autoencoders are no longer equivalent to principal component analysis (PCA), as PCA is restricted to a linear map (Japkowicz et al., 2000). We apply this type of autoencoder in our approach.

An autoencoder is a neural network consisting of an input layer, output layer and one or more hidden layers. For each input variable, the input layer contains exactly one *node* (also known as *neuron* or *unit*). In our case, the input layer contains N nodes, each corre-

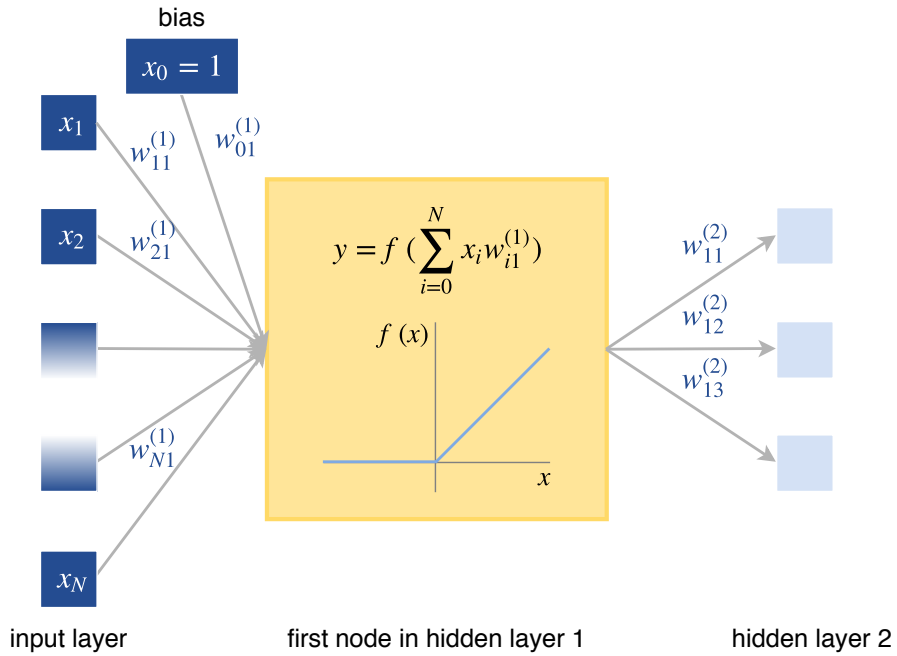


Figure 2: Detailed illustration of the first node of the first hidden layer in the autoencoder from Figure 1. The node first takes a weighted sum of its inputs (including a constant bias) and transforms this weighted sum through a non-linear activation function. The rectifier function ($f(x) = \max(0, x)$), which we use in all hidden layers, is displayed here. The output of the activation function serves as input for the next layer.

sponding to a dummy variable telling whether the corresponding product is purchased in that basket or not. As the input basket is mapped to itself, the output layer also contains N nodes. If the trained autoencoder fits the data well, the outputs of the output layer should be similar to the corresponding inputs. For example, if the first node of the input layer contains a 0, the first node of the output layer should produce an output close to 0. Apart from the in- and output layer, the autoencoder contains a prespecified number of hidden layers, each having a prespecified number of nodes, which can be different across layers. At least one of the hidden layers should contain less than N nodes and the hidden layer with the least number of nodes is called the *code* layer. The information from the input is summarized (i.e. *encoded*) in the low-dimensional code layer and *decoded* afterwards to obtain an output that is as close as possible to the input. Therefore, the first part of the network (up until the code layer) is called the *encoder* part and the second part of the network (from the code layer onwards) is called the *decoder* part. Figure 1 shows an autoencoder with three hidden layers, containing 5, 3 and 5 nodes respectively.

To capture non-linear relations between variables, each node performs a non-linear operation on its inputs and produces an output. In the input layer, no computations are performed. Therefore, the outputs of the input layer are simply the values of the input variables and these outputs serve as inputs for the first hidden layer. Each node in the

hidden layers and output layer takes a weighted sum of its inputs and transforms it non-linearly through an *activation function* to produce an output. The output serves as input for the next layer (or, in case of the output layer, serves as the final output). Figure 2 shows the operations performed in the first node of the first hidden layer. Each input x_i is multiplied with a weight $w_{ij}^{(l)}$, connecting input i with node j in layer l. Additionally, a constant *bias* $w_{0j}^{(l)}$ is added in each node. After summing the weighted inputs, the result is passed through the activation function f. This activation function is specified in advance. The most commonly used activation functions are the *rectifier* function and *sigmoid* function. In our case, we use the rectifier function in all hidden layers, as deeper networks tend to train very poorly with sigmoid activation. The sigmoid function maps its input to the small interval $(0, 1)$. As a result, a large change in input may sometimes result in a very small change in output. If multiple sigmoid layers are stacked on top of each other, a large change in the weights of the first hidden layer can lead to a very small change in the network outputs. Because the gradient of the loss function with respect to these parameters is very small, weights will hardly be updated during training. This problem is known as the vanishing gradient problem (Hochreiter, 1998) and can be avoided by using the rectifier function in hidden layers. A node with rectifier activation is often called a *rectified linear unit* (ReLU). In the last layer, we apply the sigmoid function to obtain outputs between 0 and 1.

In contrast to the activation functions, the weights $w_{ij}^{(l)}$ are not predetermined, but updated during training to minimize a certain loss function. This loss function is chosen in advance and measures the error between the final output of the network and the original input. Although the mean squared error is most commonly used as activation function, we use binary cross-entropy because of the characteristic 0/1-nature of the data. The loss function is specified as

$$L(\mathcal{W}) = -\frac{1}{|\mathcal{B}|}\frac{1}{N}\sum_{\mathbf{b}_t^c \in \mathcal{B}}\sum_{n=1}^{N}(b_{tn}^c \log(\hat{b}_{tn}^c) + (1 - b_{tn}^c)\log(1 - \hat{b}_{tn}^c)), \tag{1}$$

where $\mathcal{W}$ is the set of all network weights and $\hat{b}_{tn}^c$ represents the estimated value of the n^{th} element of $\mathbf{b}_t^c$, resulting from the network with weights $\mathcal{W}$. The meaning of all other terms can be found in Section 3.1. If b_{tn}^c is 1, the negative logarithm of $\hat{b}_{tn}^c$ is added to the loss. This means that if $\hat{b}_{tn}^c$ is close to 1, loss is low, whereas loss is very high if $\hat{b}_{tn}^c$ is close to 0. The binary cross-entropy loss function is the same as the summed negative log likelihood of N Bernoulli distributions with probabilities $\hat{\mathbf{b}}_t^c$ and outcome $\mathbf{b}_t^c$. Therefore, minimizing this loss function with respect to $\mathcal{W}$ is equivalent to maximizing the summed likelihood of these N Bernoulli distributions.

3.2.2 Backpropagation

Backpropagation (Hecht-Nielsen, 1992) is the most commonly used algorithm to train weights of feedforward neural networks. Before training, weights are randomly initialized with draws from a uniform distribution on a very small interval around 0. In each iteration, a batch of data is passed through the network and the loss function is evaluated. Starting in the output layer, the obtained loss is then "backpropagated" through the network, while updating weights contributing to the loss along the loss function gradient. Weights are thus updated using gradient descent. There are several variants of this optimization algorithm. In the case of standard gradient descent, the loss function is summed over all observations and afterwards weights are updated. As each weight update requires all observations to be passed through the network, this procedure is very slow in general.

As an alternative, gradient descent can be approximated by updating weights incrementally. That is, every time one observation has been propagated through the network, weights are updated according to the loss on that particular observation. This procedure is called *stochastic gradient descent* (SGD), as the gradient estimate is based on a single training sample and therefore a "stochastic approximation" of the "true" loss gradient. SGD is usually a lot faster than standard gradient descent, because one update require a lot less computations. Moreover, it can sometimes avoid falling into local minima (Mitchell, 1997). However, SGD also has its downsides. Because the approximated gradient can differ from the true gradient, weights may sometimes be updated in the wrong direction. As a consequence, the path to the global minimum is not as direct as in standard gradient descent.

A third option is using mini-batch gradient descent, where weights are updated after a relatively small set of observation has been passed through the network. This is computationally more efficient than pure SGD and still has a relatively high model update frequency compared to standard gradient descent, reducing the risk of falling into local minima. The size of the mini-batches must be determined in advance. Choosing a too large batch size is usually not a good idea, as large-batch methods tend to converge to sharp minima (local minima with a very big gradient in a small region around them). As these sharp minima are usually not the global minima, this leads to poorer generalization (Keskar et al., 2016). After some trial and error, we found that training with mini batches of size 100 worked well in our case. For much higher batch sizes, weight updates sometimes led to a sudden increase in the value of the loss function, both in and out of sample.

Next, we derive the backpropagation algorithm for our autoencoder, with sigmoid activation in the output layer and rectifier activation in all hidden layers. To do so, we

use the following notation:

- N_l: the number of nodes in layer l. Note that N_L, the number of nodes in the output layer, equals N, the length of a basket vector.
- $x_{ij}^{(l)}$: the i^{th} input of node j in layer l. $l = 1$ corresponds to the first hidden layer. Note that $x_{0j}^{(l)}$, the bias, is always one.
- $w_{ij}^{(l)}$: weight connecting node i in layer $l-1$ with node j in layer l.
- $z_j^{(l)}$: the weighted sum computed in unit j in layer l, equal to $\sum_{i=0}^{N_{l-1}} x_{ij}^{(l)} w_{ij}^{(l)}$.
- $o_j^{(l)}$: the output of unit j in layer l, equal to $f(z_j^{(l)})$, where f is the activation function in layer l.
- $Downstream(j^{(l)})$: the set of nodes that take the output of node j in layer l as their input.

As we use a mini-batch approach, each update is based on the gradient of the loss function computed over the baskets in that batch. For batch $\mathcal{S} \subset \mathcal{B}$, the loss function evaluated over the baskets in this batch becomes

$$L_{\mathcal{S}}(\mathcal{W}) = -\frac{1}{|\mathcal{S}|}\frac{1}{N}\sum_{\mathbf{b}_t^c \in \mathcal{S}}\sum_{n=1}^{N}(b_{tn}^c \log(\hat{b}_{tn}^c) + (1-b_{tn}^c)\log(1-\hat{b}_{tn}^c)). \tag{2}$$

The gradient of the loss function with respect to $\mathcal{W}$ specifies the direction of steepest increase, so weights must be updated in the opposite direction. For standard gradient descent, the update rule becomes:

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \frac{\partial L_{\mathcal{S}}}{\partial w_{ij}^{(l)}}, \tag{3}$$

where η is the gradient descent learning rate. For now, we stick to this constant learning rate, as it simplifies the derivations.

By the chain rule,

$$\frac{\partial L_{\mathcal{S}}}{\partial w_{ij}^{(l)}} = \frac{\partial L_{\mathcal{S}}}{\partial z_j^{(l)}}\frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}}. \tag{4}$$

By the definition of $z_j^{(l)}$, the last term always equals $x_{ij}^{(l)}$. We can split up the first term even further with the chain rule to get

$$\frac{\partial L_{\mathcal{S}}}{\partial w_{ij}^{(l)}} = \frac{\partial L_{\mathcal{S}}}{\partial o_j^{(l)}}\frac{\partial o_j^{(l)}}{\partial z_j^{(l)}}\frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}}. \tag{5}$$

The second term is the gradient of the activation function. For the rectifier function, this is $I[z_j^{(l)} > 0]$. Note that the rectifier function is not differentiable at zero, but this is not a problem in practice. It is unlikely for the rectifier input to get an input of exactly 0. In most software implementations, the gradient at 0 is simply set to 0. It can be shown that the gradient of the sigmoid function σ is $\sigma(z_j^{(l)})(1-\sigma(z_j^{(l)}))$, which equals $o_j^{(l)}(1-o_j^{(l)})$.

The first term can be easily evaluated for the output nodes. For these nodes,

$$\frac{\partial L_{\mathcal{S}}}{\partial o_j^{(L)}} = \frac{\partial}{\partial \hat{b}_{tj}^c} - \frac{1}{|\mathcal{S}|}\frac{1}{N}\sum_{\mathbf{b}_t^c \in \mathcal{S}}\sum_{n=1}^{N}(b_{tn}^c \log(\hat{b}_{tn}^c) + (1-b_{tn}^c)\log(1-\hat{b}_{tn}^c)) \tag{6}$$

$$= -\frac{1}{|\mathcal{S}|}\frac{1}{N}\sum_{\mathbf{b}_t^c \in \mathcal{S}}\frac{\partial}{\partial \hat{b}_{tj}^c}(b_{tj}^c \log(\hat{b}_{tj}^c) + (1-b_{tj}^c)\log(1-\hat{b}_{tj}^c)) \tag{7}$$

$$= -\frac{1}{|\mathcal{S}|}\frac{1}{N}\sum_{\mathbf{b}_t^c \in \mathcal{S}}\left(\frac{b_{tj}^c}{\hat{b}_{tj}^c} - \frac{1-b_{tj}^c}{1-\hat{b}_{tj}^c}\right), \tag{8}$$

where the second equation follows from the fact that the gradient is 0 for all $n \neq j$. Because the activation function in the output layer is the sigmoid, we get:

$$\frac{\partial L_{\mathcal{S}}}{\partial w_{ij}^{(L)}} = -\frac{1}{|\mathcal{S}|}\frac{1}{N}\sum_{\mathbf{b}_t^c \in \mathcal{S}}\left(\frac{b_{tj}^c}{\hat{b}_{tj}^c} - \frac{1-b_{tj}^c}{1-\hat{b}_{tj}^c}\right)\hat{b}_{tj}^c(1-\hat{b}_{tj}^c)x_{ij}^{(L)} \tag{9}$$

$$= -\frac{1}{|\mathcal{S}|}\frac{1}{N}\sum_{\mathbf{b}_t^c \in \mathcal{S}}(b_{tj}^c(1-\hat{b}_{tj}^c) - (1-b_{tj}^c)\hat{b}_{tj}^c)x_{ij}^{(L)} \tag{10}$$

$$= -\frac{1}{|\mathcal{S}|}\frac{1}{N}\sum_{\mathbf{b}_t^c \in \mathcal{S}}(b_{tj}^c - \hat{b}_{tj}^c)x_{ij}^{(L)}. \tag{11}$$

Thus, the update rule for weights in the output layer becomes

$$w_{ij}^{(L)} \leftarrow w_{ij}^{(L)} + \eta\frac{1}{|\mathcal{S}|}\frac{1}{N}\sum_{\mathbf{b}_t^c \in \mathcal{S}}(b_{tj}^c - \hat{b}_{tj}^c)x_{ij}^{(L)}. \tag{12}$$

For nodes in the hidden layers, the first term in Equation 4 must take into account the indirect ways through which weights influence the value of the loss function. Note that $z_j^{(l)}$ can only influence the loss through the node's direct successors, i.e. its downstream.

Therefore, we get:

$$\frac{\partial L_{\mathcal{S}}}{\partial z_j^{(l)}} = \sum_{d \in Downstream(j^{(l)})} \frac{\partial L_{\mathcal{S}}}{\partial z_d^{(l+1)}} \frac{\partial z_d^{(l+1)}}{\partial z_j^{(l)}} \tag{13}$$

$$= \sum_{d \in Downstream(j^{(l)})} \frac{\partial L_{\mathcal{S}}}{\partial z_d^{(l+1)}} \frac{\partial z_d^{(l+1)}}{\partial o_j^{(l)}} \frac{\partial o_j^{(l)}}{\partial z_j^{(l)}} \tag{14}$$

$$= \sum_{d \in Downstream(j^{(l)})} \frac{\partial L_{\mathcal{S}}}{\partial z_d^{(l+1)}} w_{jd}^{(l+1)} I[z_j^{(l)} > 0]. \tag{15}$$

The update rule for hidden layer weights thus becomes

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta I[z_j^{(l)} > 0] x_{ij}^{(l)} \sum_{d \in Downstream(j^{(l)})} \frac{\partial L_{\mathcal{S}}}{\partial z_d^{(l+1)}} w_{jd}^{(l+1)}, \tag{16}$$

where the unknown gradient can be computed iteratively, starting in the output layer and going backwards.

Algorithm 1 Adam optimization algorithm (Kingma & Ba, 2014). Default settings for the algorithm are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations are element-wise. g_t^2 indicates the element-wise square of g_t.

Require: α: stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: exponential decay rates for moment estimates
Require: ϵ: small constant to avoid division by zero
Require: $f(\theta)$: stochastic objective function with parameters θ
Require: θ_0: initial parameter vector
  $m_0 \leftarrow 0$ (initialize first moment estimate)
  $v_0 \leftarrow 0$ (initialize second moment estimate)
  $t \leftarrow 0$ (initialize iteration counter)
  while θ_t has not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (compute gradient w.r.t. stochastic objective in iteration t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (update biased second moment estimate)
    $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (compute bias-corrected first moment estimate)
    $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (compute bias-corrected second moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (update parameters)
  end while
  return θ_t (parameter estimates)

We have just derived the weight update rules in case of a constant learning rate η. Using a constant learning rate is, however, not always efficient. A learning rate that is too small leads to slow convergence, whereas a learning rate that is too large may cause the

weights to "jump over" their optimum, hindering convergence. In the *Adam* optimization algorithm (Kingma & Ba, 2014), a per-parameter adaptive learning rate is introduced that is based on the first and second moment of the gradient, i.e. its mean and uncentered variance (see algorithm 1). The algorithm estimates the first and second moment of the gradient by calculating an exponentially weighted average of the gradient and squared gradient, with decay rates β_1 and β_2 respectively. That is, the moment estimates are a weighted average over all earlier computed gradients, but higher weights are assigned to the more recently computed gradients. As the moment estimates are biased towards the initial estimate 0 (especially the early estimates), the algorithm computes a bias-corrected estimate by dividing through $(1 - \beta_1^t)$ and $(1 - \beta_2^t)$ respectively. Instead of updating parameters in the direction of the single gradient computed in one iteration, as in standard gradient descent, parameters are updated in the direction of the estimated first moment of the gradient, taking into account all earlier computed gradients. Moreover, the step size is adapted based on the estimated second moment of the gradient. A larger second moment estimate corresponds to a smaller step, because this indicates greater uncertainty about whether the direction of the first moment estimate corresponds to the direction of the true gradient.

Steps are repeated until a certain stopping condition is met. This could be stopping after the obtained loss on a separate validation set stops decreasing (or even starts increasing) or stopping after a fixed number of *epochs*. In one epoch, all observations are propagated through the network exactly once. In the case of (mini-batch) SGD, each epoch consists of a number of *iterations*. Weights are updated exactly once in each iteration. We stop training after 100 epochs. In all settings, the loss evaluated on the validation set did not decrease anymore by then.

Empirically, *Adam* has been proven to outperform other adaptive learning methods on a wide range of problems. Therefore, we apply it when training the autoencoder. In our application, the objective function is the binary crossentropy $L(\mathcal{W})$, with stochastic loss $L_{\mathcal{S}}(\mathcal{W})$ in each iteration. The parameters we update are the network weights, which are randomly initialized with values close to zero. This is done with the *glorot uniform initializer*, introduced by Glorot & Bengio (2010). In short, it uniformly draws weights from a small interval around zero, with limits depending on the size of the two layers that are connected by the weights (see algorithm 2). The underlying idea is that the input of each activation function initially has the same, relatively low variance. We use it because it is known for its fast convergence.

We use the default settings of the *Adam* algorithm. When weights do not converge

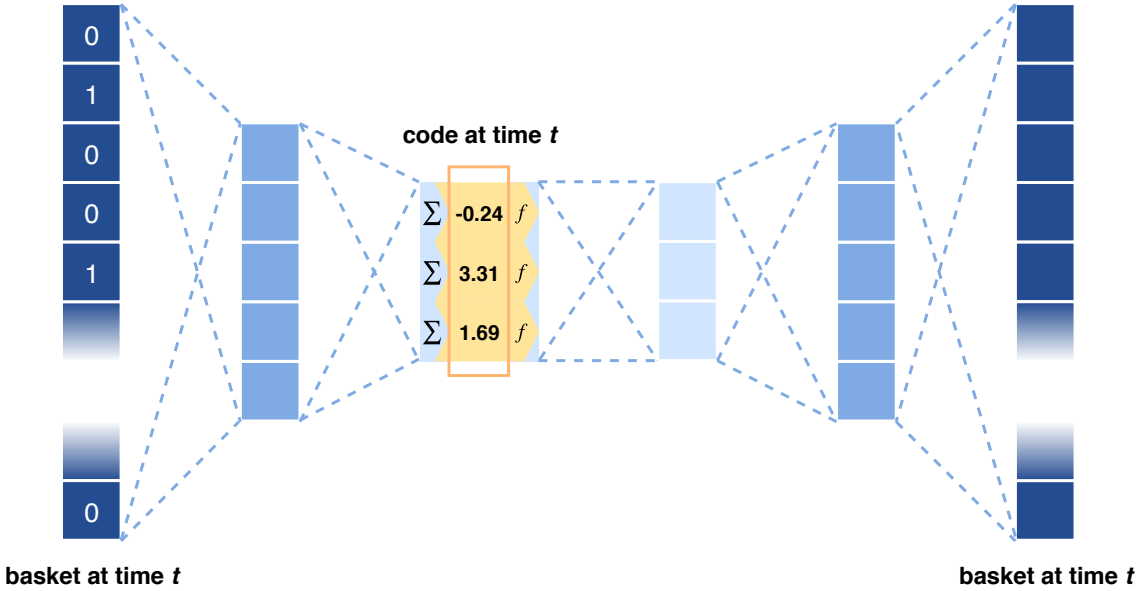


Figure 3: Example of an autoencoder with 4 hidden layers, where the middle two layers are equal in size. We consider the first of these two layers as the code layer. The code elements are the weighted sums of inputs in these nodes, before passing them through the rectifier function. In this example, the input basket is summarized in the code $[-0.24, 3.31, 1.69]$.

well, we use a step size of $\alpha = 0.0001$. The complete backpropagation algorithm with *Adam* optimizer for our autoencoder, with sigmoid activation in the output layer and rectifier activation in all hidden layers, is described in algorithm 2.

3.2.3 Evaluating basket codes

After training the autoencoder, baskets can be transformed into codes. To obtain the code evaluation of a basket, it is propagated through the trained network until it reaches the code layer. There, we define the weighted sum of inputs in each node as the elements of the code, before passing the weighted sum through the rectifier function. If we would consider the node outputs as code elements, negative code elements would be left-censored at zero. This is not desirable, as there is no more distinction between values just below zero and values far below zero. Moreover, regular regression techniques are not applicable to censored data in the prediction step. Figure 3 gives an example of an autoencoder with four hidden layers. Note that the two middle layers are of equal size, similar to the networks we train in section 4.3. The first of these two middle layers is chosen to be the code layer, although this choice is somewhat arbitrary. To decode the code, it is first passed through the rectifier function, after which it is propagated through the remaining hidden layers.

Algorithm 2 Backpropagation algorithm for our autoencoder. We use the default settings $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. When weights do not converge well, we choose $\alpha = 0.0001$. We stop training after 100 epochs, i.e. after each observation has been backpropagated 100 times. In all cases, out of sample loss has already stopped decreasing by then.

Create a feed-forward autoencoder with $L+1$ layers (the input layer has index 0 and the output layer has index L). The number of nodes in layer l is denoted by N_l (note that $N_L = N$, the size of a basket vector). All hidden layers have rectifier activation, the output layer has sigmoid activation.
Initialize network weights $\mathcal{W}$:
for l in $\{1, ..., L\}$ **do**
  for i in $\{1, ..., N_{l-1}, \}$ **do**
    for j in $\{1, ..., N_l\}$ **do**
      Draw initial weight $w_{ij0}^{(l)}$ from uniform distribution on interval
        $[-\sqrt{6/(N_{l-1}+N_l)}, \sqrt{6/(N_{l-1}+N_l)}]$ (glorot uniform initializer)
      $m_{ij0}^{(l)} \leftarrow 0$ (initialize first moment gradient estimate)
      $v_{ij0}^{(l)} \leftarrow 0$ (initialize second moment gradient estimate)
    end for
  end for
end for
$k \leftarrow 0$ (initialize iteration counter)
while stopping condition has not been met **do**
  $k \leftarrow k + 1$
  Draw a random batch $\mathcal{S} \subset \mathcal{B}$ without replacement from all training baskets. A basket can only reappear in a new batch once all other baskets have appeared in a batch.
  Initialize $g_{ijk}^{(l)} = 0$ for all network weights
  for each $\mathbf{b}_t^c$ in $\mathcal{S}$ **do**
    Propagate basket through the network with non-updated weights $\mathcal{W}_{k-1}$ to obtain output $\hat{\mathbf{b}_c^t}$. While passing an observation through the network, node inputs $x_{ijk}^{(l)}$ and weighted sums $z_{jk}^{(l)}$ are computed.
    Propagate the error backwards through the network:
    for l in $\{L, L-1, ..., 1\}$ **do**
      for i in $\{1, .., N_{l-1}\}$ **do**
        for j in $\{1, ..., N_l\}$ **do**
          if l equals L **then**
            $\delta_j^{(l)} \leftarrow -\frac{1}{|\mathcal{S}|}\frac{1}{N}(b_{tjk}^c - b_{tjk}^c)$ (error term of ouput node j)
          else
            $\delta_j^{(l)} \leftarrow I[z_{jk}^{(l)} > 0] \sum_{d=1}^{N_{l+1}} \delta_d^{(l+1)} w_{jd,k-1}^{(l+1)}$ (error term of node j in layer l)
          end if
          $g_{ijk}^{(l)} \leftarrow g_{ijk}^{(l)} + \delta_j^{(l)} x_{ijk}^{(l)}$ (update gradient of weight $w_{ij}^{(l)}$)
        end for
      end for
    end for
  end for

Update weights:
for l in $\{L, L-1, ..., 1\}$ **do**
  for i in $\{1, .., N_{l-1}\}$ **do**
    for j in $\{1, ..., N_l\}$ **do**
      $m_{ijk}^{(l)} \leftarrow \beta_1 \cdot m_{ij,k-1}^{(l)} + (1-\beta_1) \cdot g_{ijk}^{(l)}$ (biased first moment estimate)
      $v_{ijk}^{(l)} \leftarrow \beta_2 \cdot v_{ij,k-1}^{(l)} + (1-\beta_2) \cdot g_{ijk}^{2(l)}$ (biased second moment estimate)
      $\hat{m}_{ijk}^{(l)} \leftarrow m_{ijk}^{(l)} / (1-\beta_1^k)$ (bias-corrected first moment estimate)
      $\hat{v}_{ijk}^{(l)} \leftarrow v_{ijk}^{(l)} / (1-\beta_2^k)$ (bias-corrected second moment estimate)
      $w_{ijk}^{(l)} \leftarrow w_{ij,k-1}^{(l)} - \alpha \cdot \hat{m}_{ijk}^{(l)} / (\sqrt{\hat{v}_{ijk}^{(l)} + \epsilon})$ (update weights)
    end for
  end for
end for
end while
return $\mathcal{W}_k$ (parameter estimates)

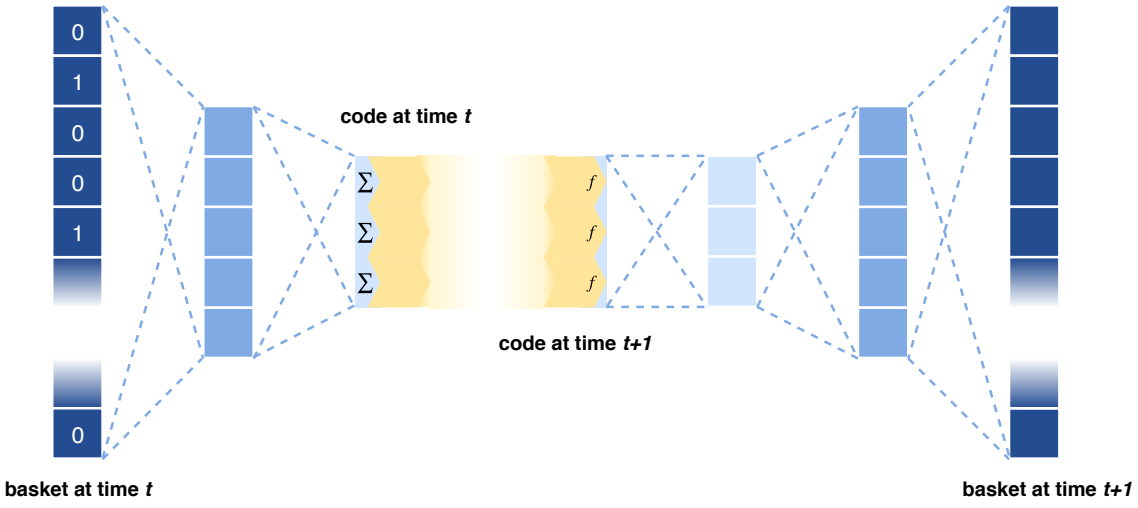


Figure 4: Example of a prediction model, based on an autoencoder with 4 hidden layer. The input basket vector at time t is encoded in a code, which is used to predict the code of the basket at time $t+1$. This code is decoded to obtain a basket prediction for time $t+1$. The exact prediction process that maps the old code to the new code differs for each variation of the model.

3.3 Predicting the next basket

The second and third step of our approach are the prediction of the next code and decoding it to obtain a basket prediction. As both steps can be integrated into one, we discuss them simultaneously in this section. Figure 12 summarizes the entire model. We consider three variations of the model, differing in how they make next basket predictions:

1. The new code is predicted using a vector autoregression (VAR) model (Lütkepohl, 2005) with one lag, where the new code elements are the dependent variables. The old code elements are used as regressors. Afterwards, the predicted code is separately decoded with the decoder part of the earlier trained autoencoder.

2. A new network is trained, with the old code as input and new basket as output. Thus, in contrast to model 1, the code is directly mapped to the next basket. The network consists of the original decoder with an added hidden layer at the start. This added hidden layer has identity activation ($f(x) = x$), such that it resembles the VAR model of the model 1. The weights for this layer are trained, whereas the decoder weights are kept fixed. The main difference with model 1 is that network weights are now trained to minimize the binary cross-entropy of the output basket, whereas in model 1 the regression coefficients are estimated to minimize the squared error on the next code.

3. The same network as in model 2 is trained, but this time the decoder weights can be flexibly updated as well. This relaxes the assumption of model 1 and 2 that successive codes are linearly related. Even if the original baskets would be linearly related, it is quite restrictive to assume that the nonlinear summaries of these baskets are linearly related as well. By allowing decoder weights to be updated during training, the model becomes more flexible and will probably gain predictive power. After all, the parameter space to search for the global minimum of the loss function becomes bigger. As a consequence, the structure of the original decoder is lost in the predictions and there is no longer a true distinction between step 2 and 3. In model 2 and 3, we train the new network for 50 epochs and save the model that performs best in terms of validation loss (i.e the binary cross-entropy evaluated on the predictions of the validation baskets).

3.4 Performance measures

The final predictions are evaluated on a test set using several measures. First, the binary cross-entropy (Equation 1) is evaluated on the test set to measure the accuracy of the basket prediction as a whole. Next, we measure the accuracy of the predicted basket size

with the mean squared error:

$$MSE(\hat{\mathbf{B}}) = \frac{1}{|\mathcal{B}_T|} \sum_{\mathbf{b}_t^c \in \mathcal{B}_T} (\sum_{i=1}^{N} (\hat{b}_{tn}^c - b_{tn}^c))^2, \tag{17}$$

where $\mathcal{B}_T$ denotes the baskets in the test set and $\hat{\mathbf{B}}$ denotes the matrix containing the predictions for these baskets. This measure indicates whether the network outputs are jointly on an accurate scale and whether the size of the basket can be predicted accurately with the predicted model outputs.

Finally, we measure the performance of the model in a product recommendation task. For a test basket containing N_{t+1}^c products, the accuracy of each model is defined as the percentage of the N_{t+1}^c bought products that is contained in the top N_{t+1}^c products with highest predicted output. This accuracy is averaged over all test baskets and reported. In reality, the top ranked products are usually the product that end up being recommended to a customer. It is important that these recommendations are good, i.e. that they reflect the actual preferences of the customer. The higher the accuracy of the top ranked products, the better the recommendations.

4 Application

4.1 The Instacart data set

The used dataset is from the American grocery delivery service Instacart (*The Instacart Online Grocery Shopping Dataset*, 2017). It contains information about approximately 3.4 million orders made by 206,209 customers in 2017. Each of these customers made at least four orders. The company offers 49,688 different products, organized in 134 aisles, which are in turn organized in 21 departments. Each order (or basket) contains one or more of these products. The exact date and time of the order is unknown, but day of the week and time of the day are given. Also, the order in which products are added to the basket is given. Figure 5 shows, for the 10 most frequently bought products in the data, the proportion of baskets containing the product, given whether or not the previous basket contained the product. We observe that the probability of a product being chosen in a basket is a lot higher when the previous basket contained the product, indicating strong first-order autocorrelation.

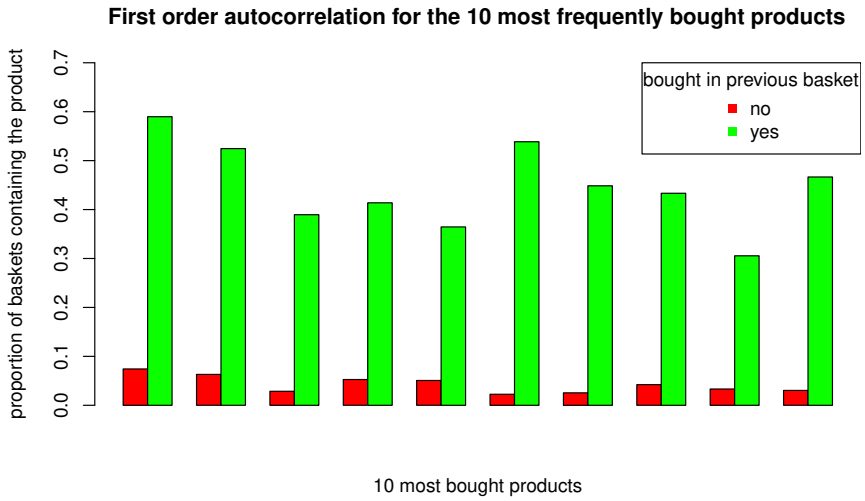


Figure 5: Barplot of the ten most frequently bought products, in random order. For each product, the proportion of baskets in which the product is contained is displayed, given whether the previous basket did (green) or did not (red) contain the product.

Instacart distributed the data on customers in a test set (75,000 customers) and training set (131,209 customers). For customers in the training set, all purchased products are given for all orders, including the last order. For customers in the test set, the purchased products are known for all orders except the last. This information is reserved by Instacart for machine learning competitions. To train and evaluate our model, we consider a smaller version of the data, which we obtain after preprocessing the data as described in the next section.

4.2 Preprocessing the data

Figure 6 shows the distribution of purchase frequency across the products in the training set. Note that this distribution is right-skewed with a long tail, especially note that the interval sizes on the right are a lot larger. The majority of the products is bought no more than 500 times, but the most popular product (*Bananas*) is bought nearly 500,000 times in total.

Before training the autoencoder, we first cluster the products in the data set. We do this for two reasons. Firstly, training an autoencoder with almost 50,000 in- and outputs (the number of unique products in the original data set) is computationally expensive and requires a lot of memory. Reducing the number of products in advance brings computation time and required memory down to an acceptable level. Secondly, it is very difficult to design a model that is capable of predicting products that are extremely infrequently bought in the training set in a sensible way. We avoid this problem if we cluster similar,

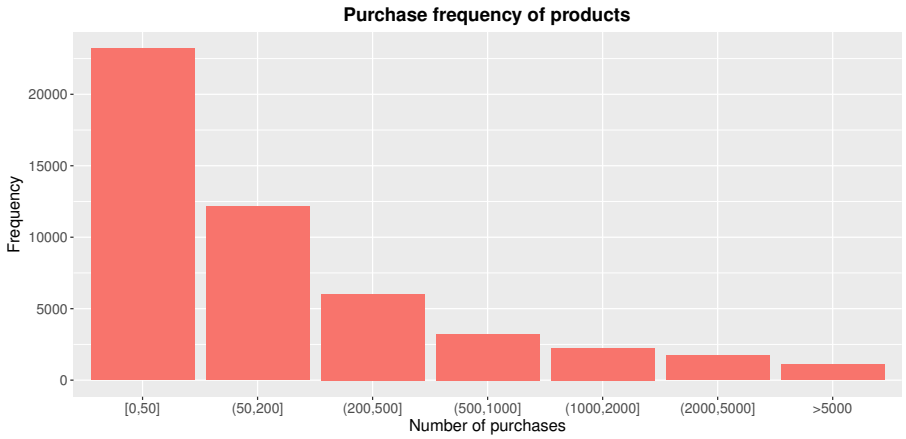


Figure 6: Histogram of purchase frequencies over all products in the training set. Note that the intervals of purchase frequencies are not equally sized.

infrequently bought products into a product group that is cumulatively bought a reasonable number of times.

Two products are considered to be similar to each other when they are bought together with the same products from the same aisle. For example, if *Braeburn Apples* and *Fireside Apples* would both appear a lot in baskets together with *Bananas*, *Kiwis* and *Strawberries* (which all from the same aisle, *Fresh fruits*), they are considered similar. Note that *Braeburn Apples* and *Fireside* do not necessarily need to appear a lot together themselves in baskets to be considered similar. Note also that we only cluster within each aisle, as patterns across aisles are to be found by the autoencoder and pre-clustering based on these patterns could bias this process.

For each aisle a, an $N_a \times N_a$ co-occurence matrix $\mathbf{C}_a$ can be constructed, where N_a denotes the number of items in aisle a. Let $\mathcal{B}_i$ denote the set of baskets in which product i appears. The off-diagonal elements of $\mathbf{C}$ are then defined by

$$c_{ij} = \frac{|\mathcal{B}_i \cap \mathcal{B}_j|}{|\mathcal{B}_i|} \quad \text{for } i, j = 1, \ldots, N_a, \quad i \neq j \tag{18}$$

The values of the diagonal elements c_{ii} are debatable. To make sure that its value is not zero, but also not too high (and influential) compared to other values in the same row, it is set to $\dfrac{1}{|\mathcal{B}_i|}$.

Next, an N_a by N_a cosine similarity matrix $\mathbf{S}$ is constructed over the rows of $\mathbf{C}$, where the elements of $\mathbf{S}$ are given by

$$s_{ij} = \frac{\mathbf{c}_i \bullet \mathbf{c}_j}{||\mathbf{c}_i|| ||\mathbf{c}_j||}, \tag{19}$$

where $\mathbf{c}_i$ is the ith row of $\mathbf{C}$. For grouping products, we only consider the rows of $\mathbf{C}$ corresponding to products (or groups of products) which are bought less than 500 times in the training data. Columns of all products are included, so an infrequently bought product can be grouped with a frequently bought product. The products in this reduced $\mathbf{C}$ matrix corresponding to the highest off-diagonal element are grouped. The basket sets and co-occurence matrix are updated based on this new grouping, as well as the cosine similarity matrix. The process is repeated until there are no (groups of) products left with less than 500 purchases.

Finally, we end up with 9,407 product groups. This still results in relatively high-dimensional baskets, but reduces computation time and memory requirements. We randomly select 5% of users (10,310) and only consider their series of baskets, also for computational reasons. The characteristics of the data set after preprocessing are summarized in Table 1. We divide the orders in a training, validation and test set. For each user, all baskets except the last are assigned to the training set. If the last basket was unknown (i.e. reserved for machine learning competitions), the second to last basket is considered the last. Next, users are randomly separated in two equally sized halfs of 5,155. For the first half, the last basket is assigned to the validation set, which is later used to determine the optimal autoencoder settings. For the second half of the users, the last basket is assigned to the test set, which is finally used to evaluate the performance of our model.

Table 1: Summary of the Instacart data set after preprocessing. A *triple* (c, n, t) is an observation of customer c buying product n at time t. With products, we mean the product clusters obtained after preprocessing.

dataset	# cust.	# products	# baskets	# triples	avg. # baskets per cust.	avg. basket size
training set	10,310	9,407	155,699	1,580,119	15.10	10.15
validation set	5,155	9,407	5,155	55,001	1.00	10.67
test set	5,155	9,407	5,155	53,353	1.00	10.35
all	10,310	9,407	166,009	1,688,473	16.10	10.17

4.3 Model training

We train the autoencoder with the baskets in the training set. To determine the optimal number of layers and nodes in the network, we use the baskets in the validation set. Each of our tested autoencoders has at least two equally-sized hidden layers, of which the first one is the code layer. We do this as it makes it easy to reuse trained weights of smaller networks as initialization weights for larger networks, if necessary. Good initializiation of a network is important, as it can speed up training and avoid falling into local minima. To bring some structure to the set of possible settings, additional hidden layers are added in pairs of two on the outsides of the network (one just after the input layer and one just before the output layer). Moreover, starting from the middle layers, the number of nodes

Table 2: All tested settings for the autoencoder. Settings marked with "-" are not considered, as it would result in hidden layers with over 640 nodes

code size	**number of hidden layers**				
	2	4	6	8	10
5					
10					
20					
40					
80					-
160				-	-
320			-	-	-
640		-	-	-	-

in each additional layer is doubled each time, which gives the autoencoder a characteristic hourglass shape. The underlying idea is that the first hidden layer makes a first summary of the input data, which is in turn summarized more densely in the second hidden layer, etc.

Sticking to these conditions, only two remaining factors influence the shape of the autoencoder: the code size (i.e. number of nodes in the two middle layers) and the number of hidden layers (which is always even in this case). We consider code sizes {5, 10, 20, ..., 320, 640} and 2 to 10 hidden layers. For example, an autoencoder with code size 20 and 6 hidden layers has 80, 40, 20, 20, 40 and 80 nodes in each hidden layer respectively. The in- and output layer, both with size 9407, are always part of the autoencoder. We do not consider layers with more than 640 nodes. Therefore, the autoencoder with code size 640 can only have 2 hidden layers. We restrict the number of nodes for computational reasons. An autoencoder with 2 layers of 640 nodes already has $(9407+1)*640+(640+1)*640+(640+1)*9407 = 12,461,247$ trainable network weights. Additional layers with more nodes would make the number of parameters unmanageable. All considered settings are summarized in Table 2.

For each setting, the autoencoder is trained for 100 epochs. After each epoch, the loss is evaluated on the validation baskets. Only the parameter configuration that performs best on the validation set is stored. For example, if the validation loss attains its minimum after 40 epochs, this network is stored and not the model obtained after 100 epochs. In most cases, the minimum of the (out of sample) validation loss does not correspond with the minimum of the (in sample) training loss. This is due to overfitting: after a certain number of epochs, the network has captured most general patterns and starts fitting to the random noise in the training data, worsening out of sample predictive performance. This problem is avoided when we only store the model that performs best out of sample. In all cases, the validation loss had already attained its minimum after 100 epochs.

After training the autoencoder, we train the three prediction models using the training

baskets, for all considered autoencoder settings. Finally, we pick for each of the three model variants the best performing setting in terms of validation loss. For this final setting, the model is evaluated on the test set. We use the *keras* package in *R* with *TensorFlow* backend for our computations. Autoencoders are trained on the *Lisa Compute Cluster*, supported by *SURFsara*. The time necessary to train each autoencoder ranged from 1,5 hours (two hidden layers, code size 5) to approximately 15 hours (five hidden layers, code size 40), depending on the number of parameters in the network.

5 Results

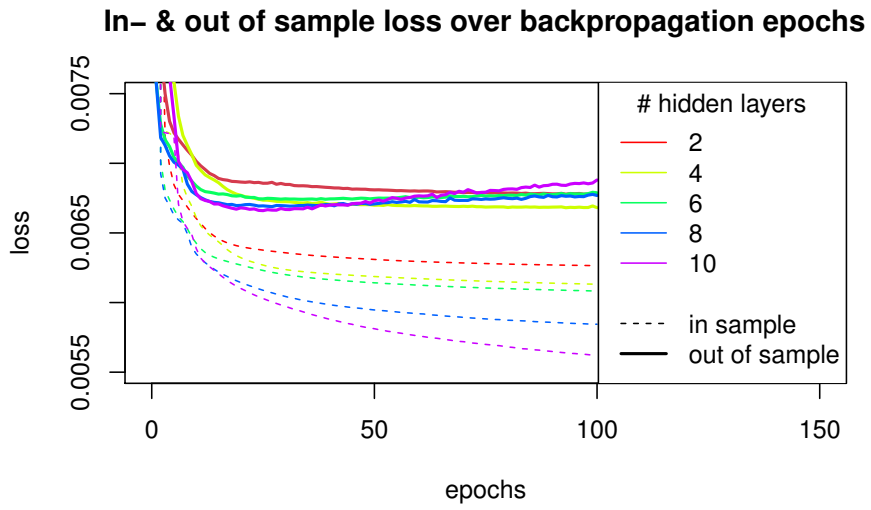
In this section, we will first discuss how the autoencoders trained over the 100 backpropagation epochs. Afterwards, we analyze the trained autoencoders in further detail. Finally, we discuss the results of the next basket predictions.

5.1 Training the autoencoder

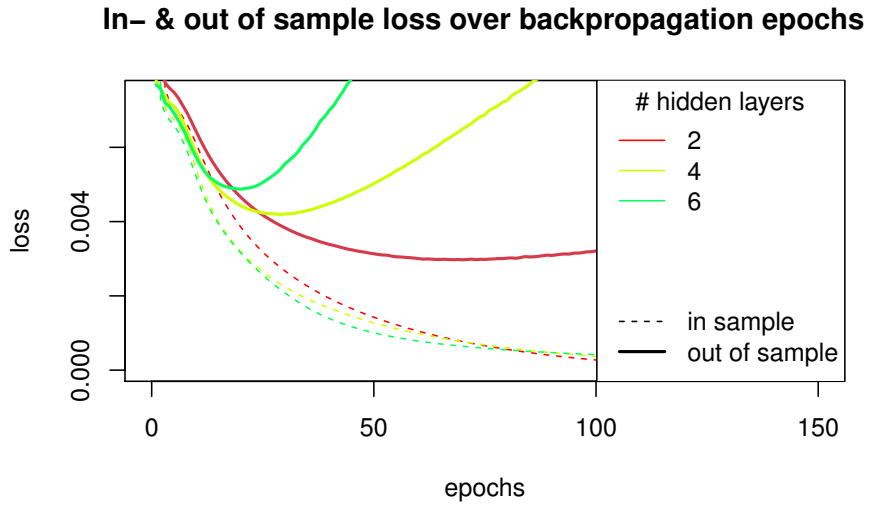
Figures 7a and 7b show the development of in- and out of sample loss of the autoencoders with code size 5 and 160, respectively. These two figures give some interesting insights into the learning process. Although the learning processes for other settings are not displayed, they yield the same insights and the effect of adding additional layers is similar. The epoch counter is displayed on the horizontal axis, whereas the vertical axis shows the average loss (binary cross-entropy) value of the network outputs. Dashed lines correspond to training loss and solid lines to validation loss. Different colours represent different numbers of hidden layers.

Note that the scale of the vertical axis is different in the two plots. Note also that binary cross-entropy is a logarithmic loss, so a relatively small absolute decrease in loss already indicates a big increase in model performance. The autoencoder with code size 160 attains lower loss values, both in the training and validation set. In general, wider autoencoders (i.e. autoencoders with a larger code size) achieve lower loss value. This is probably due to the fact that they have to pass basket information through a "wider bottleneck" than the narrower autoencoders and can therefore preserve more information of the original basket, making it easier to reproduce the original basket by the decoder.

The validation loss is higher than the in sample loss for all settings and the difference between the two only increases as training progresses. Validation loss even starts increasing for larger networks after a certain number of epochs. In general, we observe that the larger the network, the earlier the out of sample loss starts increasing again. A clear example is given in Figure 7b. This clearly indicates that these networks overfit, as weight updates do



(a)



(b)

Figure 7: Development of in- and out of sample loss over 100 training epochs, for (a) an autoencoder with code size 5 and (b) an autoencoder with code size 160. Note that for the autoencoder with code size 160, we do not consider the networks with 8 or 10 hidden layers.

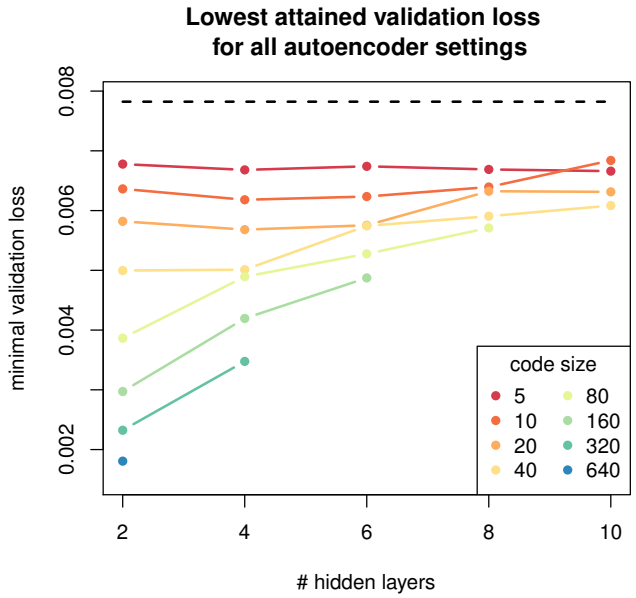


Figure 8: Lowest attained validation loss for all settings. Different colours represent different code sizes, while the number of hidden layers is displayed on the horizontal axis. For some code sizes, we limited the maximum number of hidden layers. The black dashed line represents the validation loss of always predicting the proportion of baskets in which a product appears for each product.

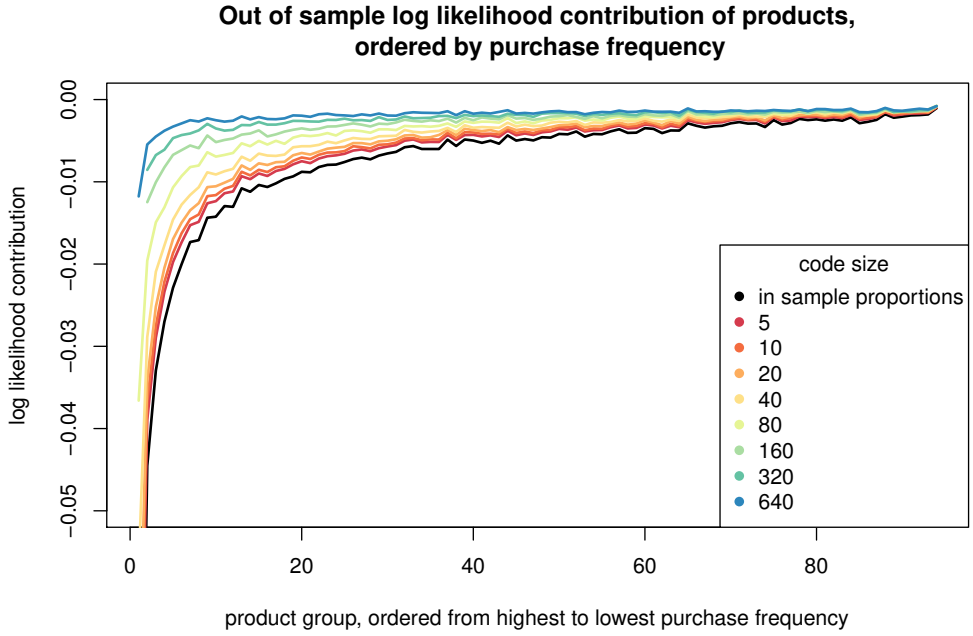
no longer help capturing a general pattern in the data. In fact, further training disturbs the earlier found general patterns, leading to higher validation loss.

For the autoencoder with code size 5, the attained minimum of the validation loss is lowest for the largest autoencoder (i.e. the autoencoder with most hidden layers), even though this autoencoder overfits after it has attained its lowest validation loss. However, for the autoencoder with code size 160, the largest autoencoder performs the worst. A summary of the lowest attained validation losses for all settings is given in Figure 8. In general, we observe that adding additional hidden layers is only useful for narrow autoencoders. Moreover, we observe that the code size is the dominant factor in the performance of the autoencoders. Regardless of its number of hidden layers, none of the autoencoders is able to outperform a wider autoencoder with two hidden layers. This does not have to mean that wider autoencoders will also perform better in terms of predicting future baskets. In fact, wider autoencoders are trained to approach the identity function between network in- and output and might not be able to capture general co-occurence patterns in baskets, which contain useful information for predictions.

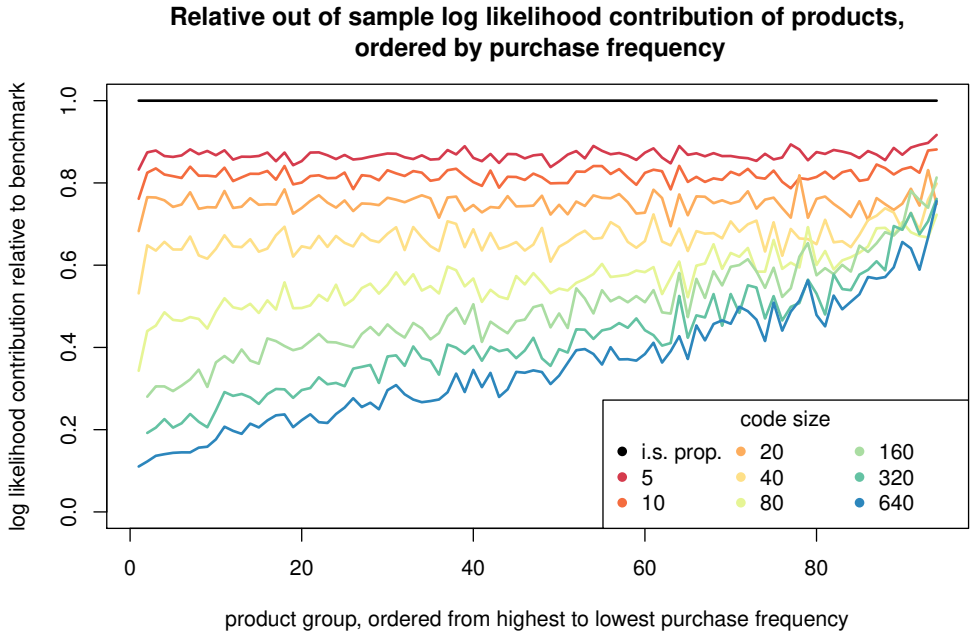
5.2 Analysis of the trained autoencoder

In this section, we discuss the properties of the trained autoencoders in more detail. Trained autoencoders are the autoencoders that attained the lowest validation loss during training for a certain setting. To start, we are interested in whether the autoencoder is able to reproduce all products well and not only the products that appear often in baskets. In Figure 9a, the out of sample log likelihood contributions (negatives of the binary cross-entropy contributions) of all products are displayed. Products are ordered from highest to lowest purchase frequency in the entire dataset. The plot only shows the two-hidden layer autoencoders for all code sizes, compared to a benchmark where in sample purchase proportions (the proportion of training baskets in which a product appears) are used as predictions for the validation baskets. To judge relative differences, the likelihood contributions relative to the benchmark are displayed in Figure 9b. The figures show that likelihood contributions of the autoencoders are higher than the benchmark over the entire spectrum of products. This indicates that the autoencoder has predictive power for all products and is not only fitted to the most frequently appearing products. However, Figure 9b does show that the predictive gain for non-frequently bought products is relatively smaller than the predictive gain for frequently bought products, especially for the wider autoencoders. In general, wider autoencoders outperform narrower autoencoders over the entire spectrum of products.

Next, Figure 10 provides insights into how the values of code elements influence the network outputs. For a network with code size 5 and two hidden layers, the relation between each code element and several outputs is plotted. The code value is displayed on the horizontal axis, whose limits are given by the minimum and maximum code value obtained in the validation data. The mean element value is given as well, together with the 5% and 95% quantiles. While changing the value of one code element, all other elements are kept constant at their mean. Figure 10a, 10b and 10c show the influence of the code on the output node corresponding to respectively the highest, second highest and median ranked product in terms of purchase frequency in the training data. The plots clearly show non-linear relationships between the code elements and network outputs. In some cases (e.g. the most bought product and code element 2), the influence of the code element is not strictly positive or negative. In those cases, increasing the code might first increase the likelihood of a product appearing in the basket, but later decrease this likelihood, or vice versa. Note that the scale of the vertical axis is different across the three products: for the first and second most bought product, the network outputs are a lot higher than for the median bought product. This is not surprising, because a well fitted autoencoder will, on average, assign higher outputs to products that are bought more frequently. Note also that axis scales within products, but across code elements can differ a lot. For example,

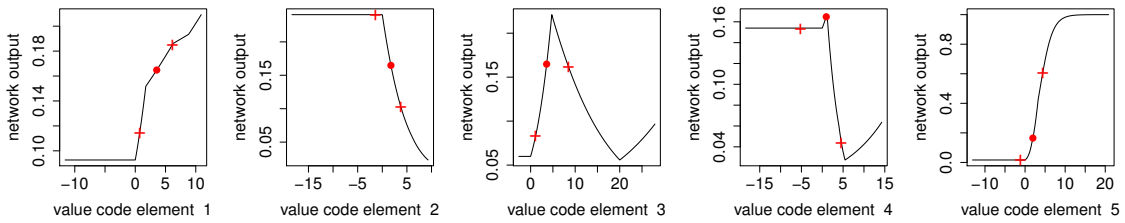


(a)

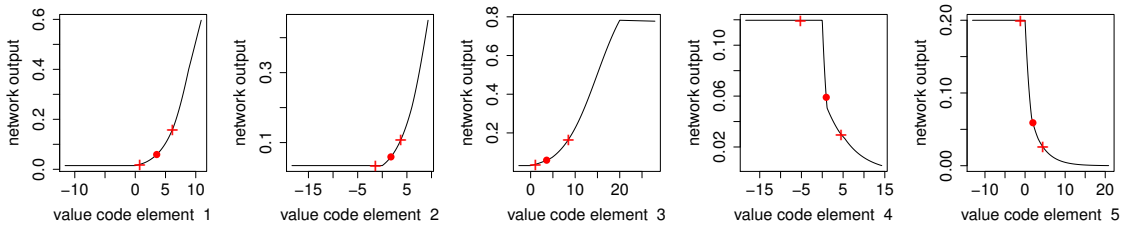


(b)

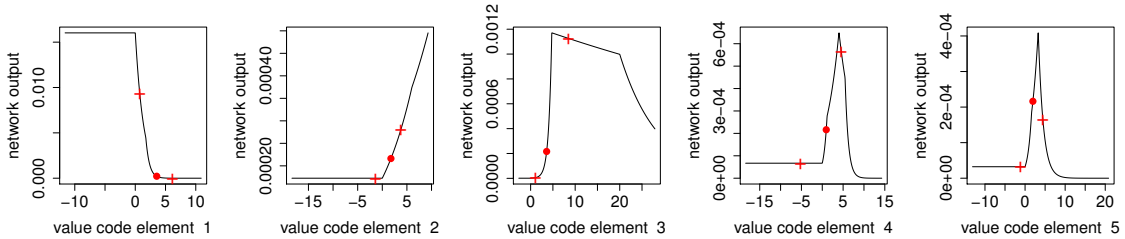
Figure 9: (a) Per-product log likelihood contributions for two-hidden layer autoencoders with different code sizes, compared to a benchmark where in sample frequency proportions are used as predictions. Products are ordered from highest to lowest purchase frequency in the complete dataset. The plotted contributions are averaged over groups of 100 products. (b) The same log-likelihood contributions for the same predictions, this time divided by the log-likelihood contributions of the benchmark.



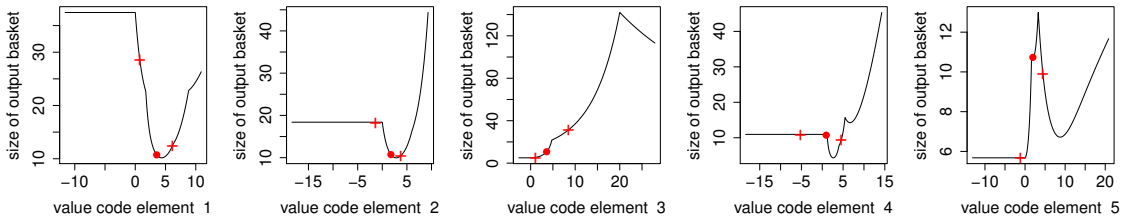
(a) Most bought product



(b) Second most bought product



(c) Median product in terms of purchase frequency



(d) Basket size

Figure 10: Relation between code elements and network outputs in a 2-hidden layer autoencoder with code size 5 for four different outputs: (a) the most bought product, (b) the second most product, (c) the median product in terms of purchase frequency and (d) the size (sum) of the entire output basket. Horizontal axis limits correspond to the minimum and maximum code value observed in the validation data. Circles indicate the average value of the code element, plus signs indicate the 5% and 95% quantiles.

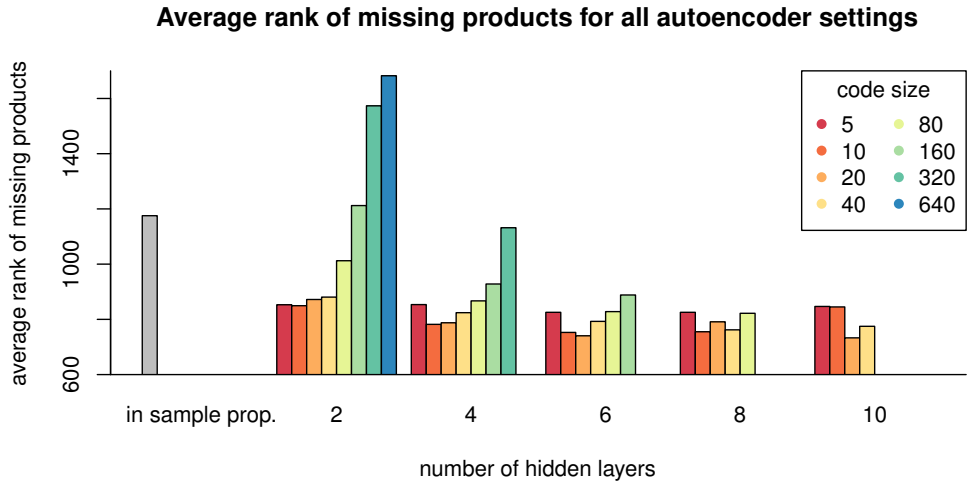


Figure 11: For each autoencoder setting, the average rank of the network output corrresponding to the product that is manually left out of a basket before passing the basket through the autoencoder.

the first code element has by far the biggest effect on the output of the product with median purchase frequency. The upper vertical axis limit in this plot is approximately 0.0015, but this does not have to mean that the output for this product can not exceed this value. The joint effect of changing all elements may result in higher outputs. Figure 10d shows the influence of code elements on the entire basket size (i.e. the sum of all outputs). Again, this effect is highly non-linear for all elements. For some code elements, we can see a clear positive or negative effect, e.g. mainly negative for the first one. This indicates that aggregate basket properties, such as the basket size, are captured in the code as well.

Finally, we are interested in how well the autoencoders capture co-occurrences of products in baskets. To measure this, we randomly remove one product from each validation basket. The incomplete basket is then propagated through the trained network and products are ranked from high to low network output. As we are not interested in the products that remained in the basket, these products are removed from the ranking. The idea is that if the autoencoder captures co-occurences, the missing product should be ranked relatively high. Figure 11 shows the average rank of the network output corresponding to the removed product. The average rank (out of the 9,407 products) of the missing products is displayed in the figure for all autoencoder settings and for a benchmark where in sample purchase proportions are used as predictions. The benchmark always assigns the highest rank to the most frequently bought product that does not appear in the basket. Remarkably, the wide autoencoders, which performed best in terms of validation loss, perform a lot worse here compared to the narrower networks. Apparently, the information stored

in wider codes is more focused on individual products, whereas narrower codes are forced to capture co-occurence patterns over all products. We also see that the networks with more hidden layers do not perform worse here. Moreover, they often assign a better average rank to the missing products. Apparently, adding additional layers does not help to reproduce the input basket, but does help capturing co-occurence patterns in the baskets.

5.3 Next basket prediction

In this section, we discuss the results of the next basket predictions. First, we determine for each model the optimal settings, based on the validation loss of the one-step ahead predictions. From this point onwards, with validation loss we mean the binary cross-entropy evaluated on the next basket predictions. With VAR model, we mean the first prediction variation, while we denote the second and third model with fixed and flexible neural network, respectively.

Figure 12 shows for all three models the lowest attained validation loss for all settings. The black dashed line represents the benchmark model, which always predicts the in sample proportion of baskets in which a product is contained. The first two models outperform the benchmark for most settings, but not for all. Both models take a weighted sum of the input code and a constant and pass the output through the fixed decoder. Table 3 and 4 respectively show the coefficients of the VAR model and the weights of the first layer with identity activation in the fixed neural network, both with code size 5 and two hidden layers. Although there are some differences, the constants and diagonal elements are dominant in both cases. This indicates that code element i at time t contains most predictive information about code element i at time $t+1$ compared to the other code elements. The differences between parameters across the two models is due to the fact that the VAR model analytically minimizes the mean squared error of the predicted codes, whereas the fixed neural network minimizes the loss on the predicted baskets through backpropagation. By the design of the model, the second model should outperform the first model. Although this is the case for most settings, the VAR model predicts better for models with narrow codes and a lot of hidden layers. Apparently, the validation loss of the fixed neural network for these settings has fallen into a local minimum or suffers from overfitting. For the VAR model, narrow codes generally predict better than wide codes. The network with code size 5 and two hidden layers performs best. This strengthens our idea that these autoencoders fit the identity link well, but are not necessarily good at predicting future baskets. For wider codes, additional layers help decreasing the validation loss. Combining this result with Figure 11, which shows a similar pattern, we suspect that adding the additional layers helps finding product co-occurences, which could explain the increased prediction performance. For the fixed neural network, there seems to be

Table 3: Estimated VAR(1) coefficients for codes from an autoencoder with two hidden layers en code size 5. * = significant on 5% significance level.

dep. var.	**regressors**					
	constant	$c_{1,t-1}$	$c_{2,t-1}$	$c_{3,t-1}$	$c_{4,t-1}$	$c_{5,t-1}$
$c_{1,t}$	1.556*	0.632*	0.008*	-0.036*	-0.047*	-0.022*
$c_{2,t}$	0.964*	-0.010*	0.602*	-0.002	-0.015*	-0.059*
$c_{3,t}$	1.123*	-0.030*	-0.010*	0.640*	0.039*	0.099*
$c_{4,t}$	0.648*	-0.124*	-0.050*	0.105*	0.711*	-0.112*
$c_{5,t}$	0.836*	-0.063*	-0.048*	0.049*	-0.021*	0.665*

Table 4: Network weights w_{ij} in the first layer of the prediction network with two hidden layers and code size 5

j	i					
	0 (bias)	1	2	3	4	5
1	0.310	0.777	0.029	0.009	-0.082	-0.041
2	0.125	0.061	0.673	0.085	-0.044	-0.084
3	0.552	0.045	0.061	0.778	0.050	0.046
4	-0.100	-0.037	-0.027	0.040	0.995	-0.018
5	0.407	0.011	-0.053	0.062	-0.064	0.817

no relation between code size, number of hidden layers and prediction performance. The network with code size 320 and two hidden layers performs best.

The flexible neural network outperforms the benchmark and both other models for all settings. It turns out that wide codes predict better in this case than narrow codes, in contrast to the VAR model. The differences in performance between models with different code sizes is a lot smaller than in Figure 8, though. Adding additional layers helps slightly for narrow networks, but not for wide networks. The network with code size 640 and two hidden layers performs best overall.

Next, we evaluate the three models with their optimal settings on the test set. Table 5 reports several performance measures for the three models and the benchmark. Firstly, the binary cross-entropy (BCE) is evaluated on the test baskets. The obtained values are similar to the validation loss given in Figure 12. Secondly, the predictions of basket size, i.e. the sum of predicted outputs, is measured with the mean squared error (MSE). Lastly, the average accuracy of the top ranked products is reported, in percentages. Although these percentages do not seem very high at first sight, it must be noted that there are 9407 products to choose from and the average basket only contains about 10 products. Across all measures, the flexible neural network clearly performs best. The difference compared to the fixed neural network is substantial for the binary cross-entropy, but small for the other two measures. The VAR model performs worst in all measures, but still better than the benchmark.

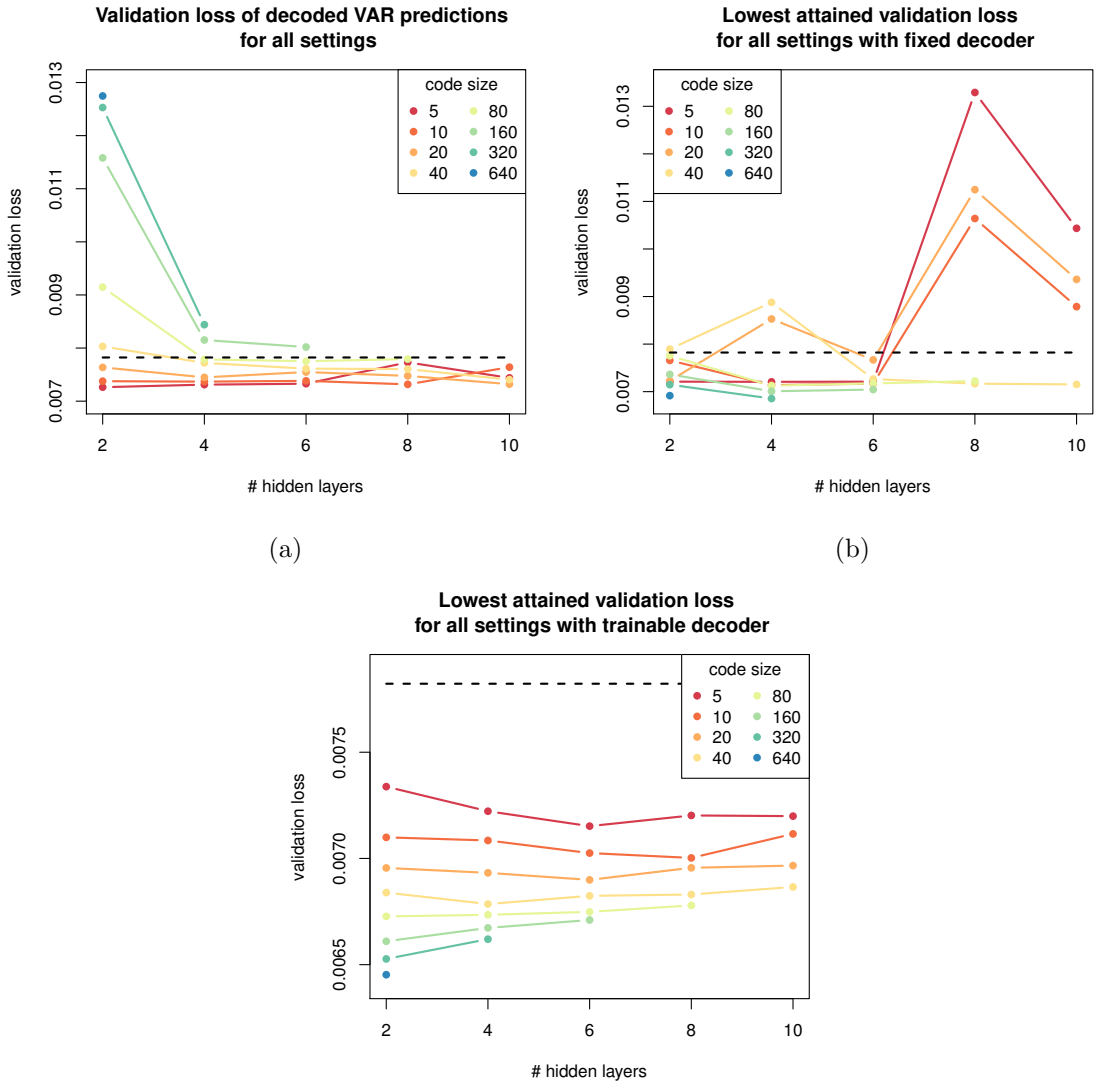


Figure 12: Lowest attained validation loss for one-step ahead predictions with (a) a VAR model followed by a decoder, (b) a neural network with fixed, non-trainable decoder and (c) a neural network with trainable decoder. Different colours represent different code sizes, while the number of hidden layers is displayed on the horizontal axis. The black dashed line represents the validation loss of always predicting the in sample proportion of baskets in which a product appears.

Finally, the ranking of all products is evaluated by means of the ROC curve in Figure 13. In this curve, all (customer, product) pairs in the test set are ordered from highest to lowest predicted output, from left to right. True positive rate (i.e. the proportion of pairs that actually result in a purchase in the test set) is given on the vertical axis and false positive rate (i.e. the proportion of pairs not resulting in an actual purchase) on the horizontal axis, meaning that the curve with the biggest area under it corresponds to the best ranking. That being said, we conclude that the predictions of the flexible neural network result in the best ranking. The other two models also clearly outperform the benchmark.

Table 5: Measures of predictive performance of the three models, compared to the benchmark: binary cross-entropy over all basket predictions, mean squared error of the predicted basket size (sum of the predicted outputs) and average accuracy of the top N^c_{t+1} predicted products, where N^c_{t+1} denotes the number of products in test basket $\mathbf{b}^c_{t+1}$.

model	BCE ($\times 10^{-3}$)	basket size MSE	top prod. accuracy (%)
proportions	7.60	62.21	3.80
VAR + fixed decoder	7.02	46.17	4.79
network layer + fixed decoder	6.65	44.43	9.33
network layer + trainable decoder	6.27	44.50	9.99

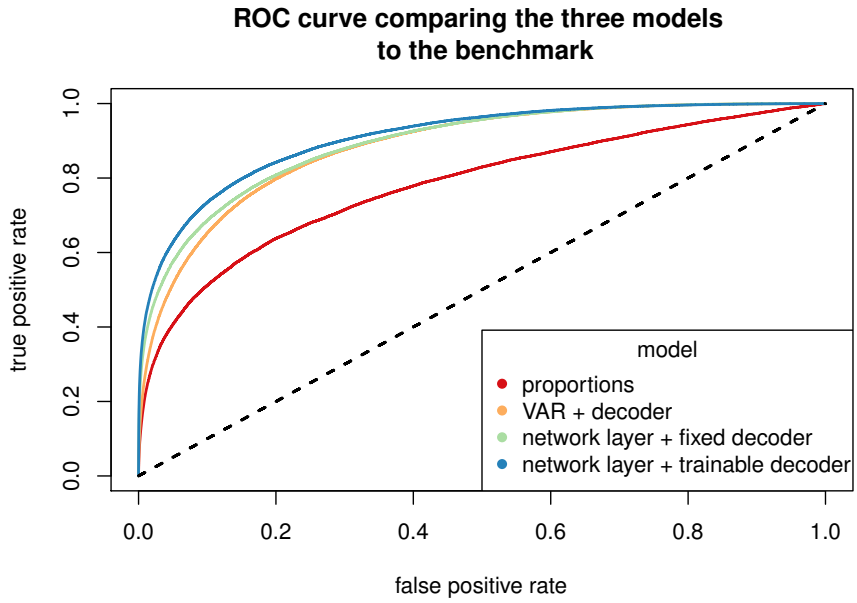


Figure 13: ROC curve of the three models and benchmark. The black dashed line corresponds to the expected true positive rate of a random ranking. For each model, all (product, customer) combinations are ranked from highest to lowest predicted output.

6 Conclusion and discussion

In this study, we have proposed a three-step autoencoder-based model for next basket prediction. We considered three models, differing in flexibility. All of these models outperform the benchmark model on the *Instacart* dataset, but the third model clearly performs the best in almost all performance measures. This model first summarizes baskets using a trained autoencoder with code size 640 and two hidden layers. Codes are mapped on the next basket with a new network, containing the original decoder and an additional linear layer just before it. In contrast to the second model, all weights of this network are trained again before making predictions.

Besides that, we have shown how the trained autoencoder is able to summarize high-dimensional baskets in such a way that an input basket can accurately be reproduced. Even an autoencoder with only 5 nodes in the code layer is able to gain a lot of predictive performance compared to the benchmark model over the entire spectrum of products. The code size turns out to be predominant factor in out of sample prediction performance: wide autoencoders (i.e. autoencoders with large code size) with two hidden layers strictly outperform narrower autoencoders. Additional hidden layers can be useful for narrow autoencoders (code size < 40), but only lead to strong overfitting in wide autencoders. Another interesting feature of the autoencoders is their ability to capture product co-occurrence patterns, without imposing any co-occurrence structure. Especially narrow baskets are able to capture these patterns. Remarkably, adding additional layers to wide networks does improve the network's ability to capture co-occurrence patterns.

In future research, it would be interesting to compare our model with other existing next basket prediction models, such as the ones mentioned in the Literature section. There are still several ways in which we could further develop the model. Our approach is limited in the sense that it only takes one lagged basket into account and does not capture general user taste in the model. Potentially, prediction performance can be further improved when all previous baskets of a customer are taken into account. One possible way to do this is by training a *recurrent* neural network such as in the *DREAM* model (Yu et al., 2016). We could integrate our model with the *DREAM* model by using our encoded baskets as input, rather than the latent basket vector that is used in the *DREAM* model now. This would allow the model to make predictions based on the entire purchase history of a customer.

Another interesting direction for future research would be using *denoising autoencoders*. In denoising autoencoders, inputs are randomly corrupted (e.g. by changing an input from 1 to 0 or vice versa), but the required output is the original, clean input. The

underlying idea is that during training, the autoencoder learns to "denoise" the input and retrieve the clean input before corruption. In our application, this could force the autoencoder to learn general basket patterns rather than just the identity function. Besides the fact that this could result in codes with more predictive power, it is also very interesting in the field of market basket analysis. We have already seen that our autoencoder is able to assign relatively high outputs to missing products in incomplete baskets. As a denoising autoencoder is specifically trained on this property, it will probably be even better in capturing product co-occurrences.

References

Ackley, D. H., Hinton, G. E., & Sejnowski, T. J. (1985). A learning algorithm for boltzmann machines. *Cognitive Science*, *9*(1), 147 - 169. Retrieved from `http://www.sciencedirect.com/science/article/pii/S0364021385800124` doi: https://doi.org/10.1016/S0364-0213(85)80012-4

Baldi, P., & Hornik, K. (1989). Neural networks and principal component analysis: Learning from examples without local minima. *Neural networks*, *2*(1), 53–58.

DeMers, D., & Cottrell, G. W. (1993). Non-linear dimensionality reduction. In *Advances in neural information processing systems* (pp. 580–587).

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249–256).

Guidotti, R., Rossetti, G., Pappalardo, L., Giannotti, F., & Pedreschi, D. (2017). Market basket prediction using user-centric temporal annotated recurring sequences. In *Data mining (icdm), 2017 ieee international conference on* (pp. 895–900).

Hecht-Nielsen, R. (1992). Theory of the backpropagation neural network. In *Neural networks for perception* (pp. 65–93). Elsevier.

Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, *6*(02), 107–116.

The Instacart Online Grocery Shopping Dataset. (2017). (Accessed from `https://www.instacart.com/datasets/grocery-shopping-2017` on March 26, 2018)

Japkowicz, N., Hanson, S. J., & Gluck, M. A. (2000). Nonlinear autoassociation is not equivalent to pca. *Neural computation*, *12*(3), 531–545.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., & Tang, P. T. P. (2016). On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Lütkepohl, H. (2005). *New introduction to multiple time series analysis*. Springer Science & Business Media.

Mitchell, T. M. (1997). *Machine learning*. McGraw-Hill, Inc. (New York, NY, USA), edn. 1, ch. 4, Artificial Neural Networks, 81-127.

Rendle, S., Freudenthaler, C., & Schmidt-Thieme, L. (2010). Factorizing personalized markov chains for next-basket recommendation. In *Proceedings of the 19th international conference on world wide web* (pp. 811–820).

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985). *Learning internal representations by error propagation* (Tech. Rep.). California Univ San Diego La Jolla Inst for Cognitive Science.

Wang, P., Guo, J., Lan, Y., Xu, J., Wan, S., & Cheng, X. (2015). Learning hierarchical representation model for nextbasket recommendation. In *Proceedings of the 38th international acm sigir conference on research and development in information retrieval* (pp. 403–412).

Yu, F., Liu, Q., Wu, S., Wang, L., & Tan, T. (2016). A dynamic recurrent model for next basket recommendation. In *Proceedings of the 39th international acm sigir conference on research and development in information retrieval* (pp. 729–732).