6. Deep Neural Networks and Genetic Algorithms

c

c

Summary: *In chapter six we will study recommendation algorithms based on artificial neural networks. In particular, we will describe the* ***Multi-layer Perceptron - (MLP)****, the basic, feed-forward, multi-layered neural network with many layers of analysis. We will also describe Convolutional Neural Networks, which are particularly effective in image processing and recognition and, since an image is usually represented by a two-dimensional matrix, they can - by analogy - also be applied to recommender systems, since they use a user-item rating matrix. Moreover, we will describe* ***Recurrent Neural Networks****, which allow the recommendation system to "forget" very old interactions of the user with items, by using the* ***LSTM*** *and* ***GRU*** *building blocks, and is therefore useful for item recommendations whose life span is relatively short (e.g., news articles). Finally, we will describe recommendation systems based on* ***genetic algorithms****, which simulate the natural phenomenon of* ***evolution*** *and* ***natural selection****: the search for the appropriate neighborhood of the target user, thus, starts with a number of random neighboring users based on a set of initial assumptions. On this initial population, and once its members have been evaluated by means of a* ***fitness function****, the new generation of neighbors is generated through reproduction operations (e.g. crossover, mutation, etc.).*

Required knowledge: *Prior study of Chapter 2 and Chapter 5 is recommended, because algorithms (item-based CF and UV-decomposition) will be implemented with neural networks in this chapter.*

6.1 The structure of a Perceptron

6.1.1 Single-layer Perceptron

In the human brain, a nerve synapse is a link that allows a neuron (or nerve cell) to transmit an electrical or chemical signal to another neuron, causing a change in the state of the latter. In biological systems, learning occurs through an increase in the strength (e.g. thickness) of synapses as a result of their being strengthened by the repetition of the same external stimuli.

Artificial neural networks now simulate the human brain by using neurons (or nodes) that are connected to each other through links called **synapses**. The most basic computational structure in *artificial neural networks* is the *perceptron*, which contains a set of input nodes and an output node. The architectural structure of a *perceptron* is shown in Figure 6.1.

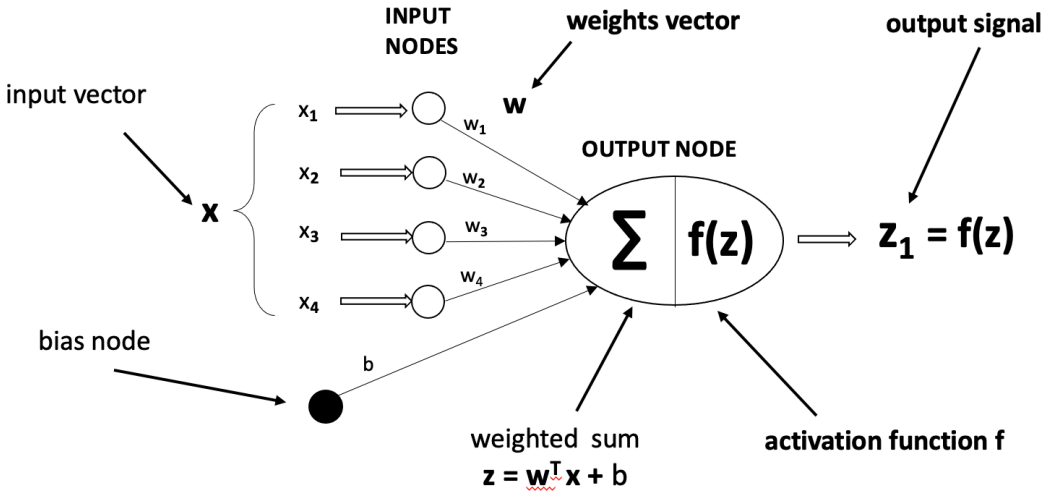


Figure 6.1: Example of a Single-Layer Perceptron using bias.

So *perceptron* takes a set of inputs, performs some mathematical calculations and gives an output. The incoming signals are represented as an input vector:

$$\mathrm{x} = [x_1, x_2, \dots, x_n], \quad x_i \in \mathbb{R}.$$

Neurons belonging to different **layers of a neural network** are connected to each other via *appropriate links* represented by using directed edges of a graph. The strength of the *links* is expressed using *weights* on the directed edges of the graph. The *weights* on the directed edges connecting a neuron of one level to a neuron of the next level are grouped into a *weights' vector*:

$$\mathrm{w} = [w_1, w_2, \dots, w_n]^T, \quad w_i \in \mathbb{R}.$$

The first computation performed by a *perceptron* is the **weighted sum**. More precisely, it multiplies each input by its corresponding weight. Then all the inputs are summed and a term called **bias** is added, as shown in the center of Figure 6.1 and is modeled as an inner product:

$$\mathbf{z} = \sum_{i=1}^{n} w_i x_i = \mathrm{w}^T \mathrm{x} + b$$

The second calculation performed by the neuron is the *activation function*. This is done by taking the output of the weighted sum and passing it through an *activation function* f, as shown in the right part of Figure 6.1.

$$z_1 = f(z) = f\left(\mathrm{w}^T \mathrm{x} + b\right) = f\left(\sum_{i=1}^{N} w_i x_i + b\right)$$

Choosing the appropriate activation function f is a critical part of neural network design. We emphasize that the most basic activation function $f(\cdot)$ is the *identity activation function*. However, it does not provide *non-linearity*, and only works for cases where the data can be separated into two groups using a simple line:

$$f(x) = x$$

The *linear/identity* activation function is often used at the output node of the neural network when our prediction is about a real number. For example, a recommendation system is trying to predict the rating a user would give to an item. In this case, the prediction of the rating is a variable expressed in terms of a real number, and therefore it makes sense to use the linear (identity) activation function. However, the resulting neural network algorithm is the same as *least squares regression*.

We note here that the main advantage of neural networks over other predicting methods is the cases in which the former use activation functions that can express non-linear correlations between data, especially in situations where the data are not separated into two groups using a simple line. Such classical, non-linear activation functions are the *sigmoid function* and the *tangent function (tanh function)*:

$$f(x) = \frac{1}{1+e^{-x}} \qquad \text{(sigmoid function)}$$

$$f(x) = \frac{e^{2x}-1}{e^{2x}+1} \qquad \text{(tanh function)}$$

We emphasize that most basic machine learning models can be implemented with simple neural network architectures. Below we will "model" two traditional machine learning techniques (*linear regression* and *UV-decomposition*) as neural network architectures, in order to demonstrate how *deep learning* can be a special case of traditional machine learning.

Example 6.1 Given the user-item rating matrix, which is shown in Figure 6.2, we are asked to predict the rating of user U_4 for item I_4 using a *single-layer perceptron*. The neural network architecture to be used to predict the rating using a *single-layer perceptron* is shown in Figure 6.3. The aforementioned neural network (see Figure 6.3) models the *item-based collaborative filtering* algorithm, which was described in details in Chapter 2.

	I_1	I_2	I_3	I_4
U_1	4	1	1	4
U_2	1	4	2	0
U_3	2	1	4	5
U_4	1	4	1	?

Figure 6.2: User-Item Rating matrix A (4×4).

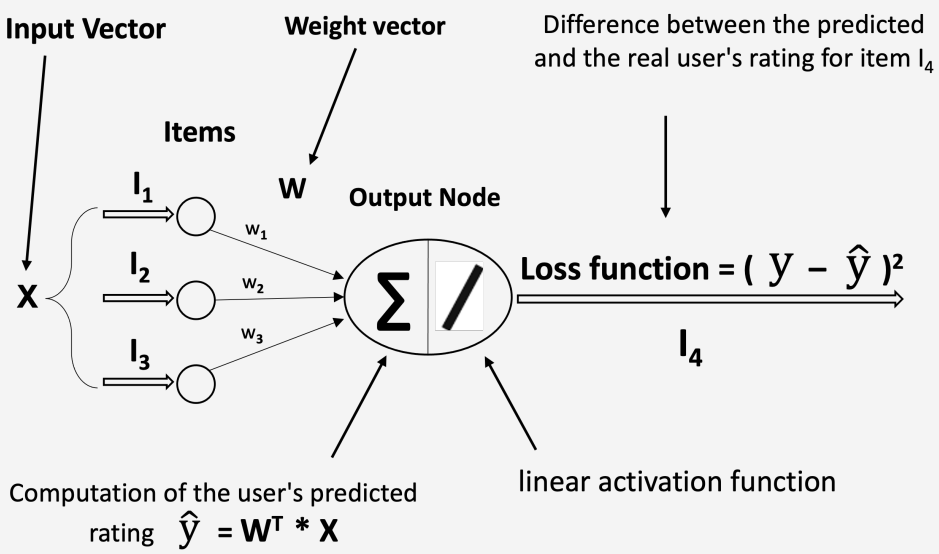


Figure 6.3: Example of a *Single-Layer Perceptron* that implements *Item-based Collaborative Filtering* algorithm (described in Chapter 2) on the data of Example 6.1

.

More specifically, the neural network in Figure 6.3 has $m-1$ input nodes $\{I_1, I_2, I_3\}$ where m is the number of items in Example 6.1. The ratings of the input items are used to predict the rating of the remaining fourth item I_4, as shown in Figure 6.3. The variable $\hat{y}$ represents the rating predicted by the neural network for the items, while the variable y represents the actual ratings of the items. The calculation of the predicted rating of an item $\hat{y}$ is computed as the inner product of the input vector $\mathbf{X}$ and the weight vector $\mathbf{W}$ of the neural network as follows:

$$\hat{y} = \mathbf{W}^T \cdot \mathbf{X}$$

As shown in the output node of Figure 6.3, the *activation function* of the neural network is the *linear* or *identity function* $f(x) = x$. Also, (see Figure 6.3) the objective of the *loss function* L is to minimize the difference or else the error e between the actual and predicted item ratings:

$$L = e^2 = \arg\min_{\mathbf{W}} \sum_{(\mathbf{X},y)\in\mathcal{D}} (y - \hat{y})^2$$

where $(\mathbf{X}, y)$ is the pair of training data $\mathcal{D}$ introduced into the neural network.

The difference between actual and predicted ratings is used to update the weight vector $\mathbf{W}$ in a manner similar to the optimization procedure followed in *least squares regression*, which was described in detail in the previous section. In particular, the optimization of the parameters of the weight vector $\mathbf{W}$ is equivalent to the updates based on the *gradient descent* method as applied in *least-squares regression.* This method (*gradient descent*) finds the direction in which the weight vector $\mathbf{W}$ should be changed in order to minimize the *loss function* L. Note that the updates of the

gradient descent method, also known as the **delta rule**, are determined by computing the partial derivative of the *loss function* e^2 with respect to the weight vector $\mathbf{W}$. Note that the set of partial derivatives relating to all variables of the *loss function* e^2 is called the *gradient*. Therefore, whenever a pair of training data $(\mathbf{X}, y)$ is introduced into the neural network, the *Gradient* of the *loss function* can be calculated as follows:

$$\frac{\partial e^2}{\partial \mathbf{W}} = -2 \cdot e \cdot \mathbf{X}$$

Moving each time in the opposite direction of the above computed partial derivative, we can minimize the *loss function* e^2 with the following update rule:

$$\mathbf{W}^{t+1} = \mathbf{W}^t + 2 \cdot \eta \cdot e \cdot \mathbf{X}$$

where $\eta \in (0, 1]$ is the learning rate and W^t is the value of the weight vector at the t-th iteration of the algorithm. As shown in Figure 6.3 at the input of the neural network we gradually introduce into the input vector $\mathbf{X}$ all the observed ratings of the items $\{I_1, I_2, I_3\}$ in Figure 6.2, i.e., the following triplets:

$$\{4, 1, 1\}, \{1, 4, 2\}, \{2, 1, 4\} \text{ και } \{1, 4, 1\}.$$

The neural network in Figure 6.3 is used to predict the rating of U_4 on item I_4. In particular, after inserting a triplet of training data into the input of the neural network the items $\{I_1, I_2, I_3\}$ the neural network predicts a rating for the item I_4. Next, we compare the rating predicted by the neural network for item I_4 with the actual rating of the item, which is found in the 4th column of the user-item rating matrix A (see Figure 6.2). Thus, the predictions of the ratings obtained by the neural network are used to generate a new approximation user-item rating matrix $\hat{A}$, which is compared with the original matrix A. If there is a difference in their ratings, then we correct the weights' vector $\mathbf{W}$ accordingly using the back-propagation technique, which propagates backwards (from the output of the neural to its input) those values that minimize the sum of squared errors between the actual and predicted ratings. This approach is repeated iteratively until the values of the matrix $\hat{\text{A}} = \mathbf{W}^T \cdot \mathbf{X}$ converge, so that they do not change any further. Note that each iteration requires the insertion of n of training data (n triplets) into the neural network where n is the number of users. In our example, in each iteration 4 triplets are inserted into the neural network, given that we have 4 users who have rated the items $\{I_1, I_2, I_3\}$ whose ratings are used to predict the rating of item I_4.

The main disadvantage of *perceptron* is that it cannot split non-linearly separable data since it consists of only the input node layer and a single output node. To understand better the above disadvantage we will present the example below:

Example 6.2 Suppose that we are given the two-input *perceptron* of Figure 6.4 that implements the *logical operation OR* using as *activation function* the *step function* with *threshold* $t = 0$. The input values of *perceptron* may be 0 (false) or 1 (true). For example, if the pair of

input values of *perceptron* is $\{x_1 = 0, x_2 = 1\}$, then its weighted sum is:

$$z = w_1 * x_1 + w_2 * x_2 + b = 2 * 0 + 2 * 1 - 1 = 1$$

and the output of its step function is $f(z) = 1$, because $z > 0$. Therefore, the output of the perceptron is 1 (True), which is consistent with the truth table of the logical operation OR as shown in the right-hand side of Figure 6.4. Alternatively, if the pair of input values of *perceptron* is $\{x_1 = 0, x_2 = 0\}$, then its weighted sum is:

$$z = w_1 * x_1 + w_2 * x_2 + b = 2 * 0 + 2 * 0 - 1 = -1$$

and the output of its *step activation function* is: $f(z) = -1$, because $z < 0$. Therefore, the output of *perceptron* is -1 (False), which is consistent with the *truth table* of the *logical operation OR* as shown in the right part of Figure 6.4.

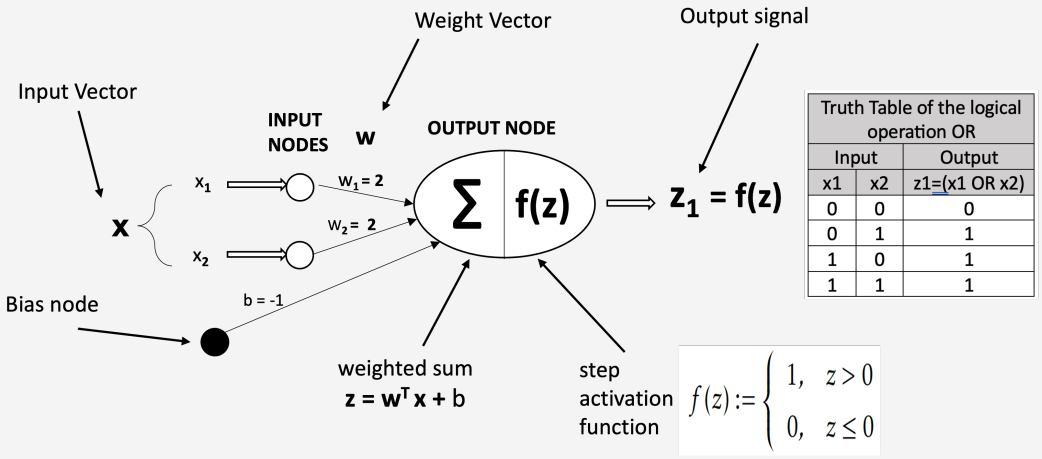


Figure 6.4: Example of a two-input *perceptron* implementing the *logical operation OR*.

It should be emphasized here that the *perceptron* of Figure 6.4 can distinguish the points of a Cartesian x-y axis system between two hemispaces separated by a hyperplane and modulated by the x values of the inputs of *perceptron* for which the following equality holds:

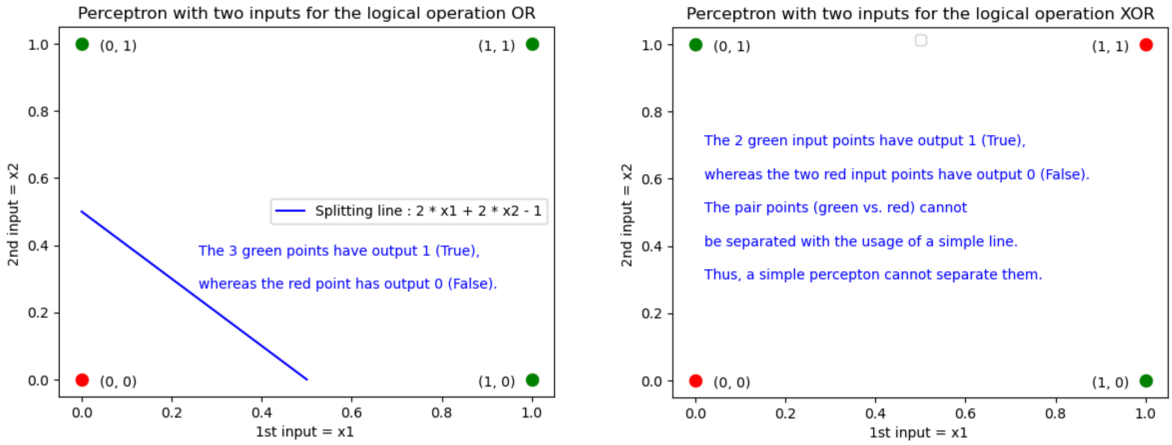
$$w_1 * x_1 + w_2 * x_2 + b = 0$$

Thus, as shown by the blue split line in Fig. 6.5(a), the *perceptron* of example 6.2 splits the space of the Cartesian x-y axis system into two hemispaces with the following linear equation:

$$2 * x_1 + 2 * x_2 - 1 = 0$$

We emphasize, here, that the coefficients $w_1 = 2$, and w_2= 2 are the ones responsible for the slope of the line, and b = -1 (bias) is its intercept with the Cartesian x-y axis system. Please note that there are many different combinations of the above coefficients that make the separation of the

inputs and output of the *logical OR operation* into false and true values. This is, after all, the main goal that **machine learning** is trying to achieve: that is, to find the most appropriate weight coefficients w and the appropriate *bias coefficient* b, to separate the data in an optimal way. However, there are some problems that cannot be solved by a *perceptron*, and we give a relevant example below.



(a) (b)

Figure 6.5: Geometric representation of the inputs and outputs of the (a) *OR* and (b) *XOR* logical operations.

Example 6.3 Suppose we are asked to implement the *logical gate XOR* using a *perceptron*, which outputs 1 (True) if one of its inputs is 1 and the other input is 0, according to the following *truth table* of the *logical XOR operation*:

Inputs		Output
x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

As shown in Figure 6.5(b), the input points of the *logical operation XOR* are impossible to be separated by means of a straight line. For this reason, neural networks were developed which have more intermediate layers of neurons to be able to separate the data, which will be included either in *convex solution regions* or in *non-convex solution regions*, and which we will discuss in detail in the next Section. [a].

[a] Convex solution regions are those that do not have a straight line segment that has both ends inside the region and some points outside. Otherwise the region is called non-convex

6.1.2 Multi-layer Perceptron

In multi-layer neural networks, neurons are arranged in layers, of which those of input and output are separated by another group of additional intermediate layers. These intermediate layers are called *hidden*, because the computations performed in them are not visible to the user. Multi-layer neural networks following this architecture are called *Feed Forward Neural Networks*, because the successive layers feed each other in a forward direction, from input to output. The architecture of these networks usually assumes that each node in one layer is connected to all nodes in the next layer **(Fully Dense Network)**.

The main advantage of *multi-layer neural networks* is that they provide the ability to compute complex nonlinear functions which is not easy with other prediction methods (e.g. via *linear regression*). In particular, in the neurons of each different layer of such a neural network we can apply alternative *activation functions* (e.g. *Tanh, ReLU, Sigmoid*, etc.) which allow to separate the data in a non-linear way. Such a three-layer neural network with an additional hidden layer is illustrated in Figure 6.6 based on the data in Example 6.1.

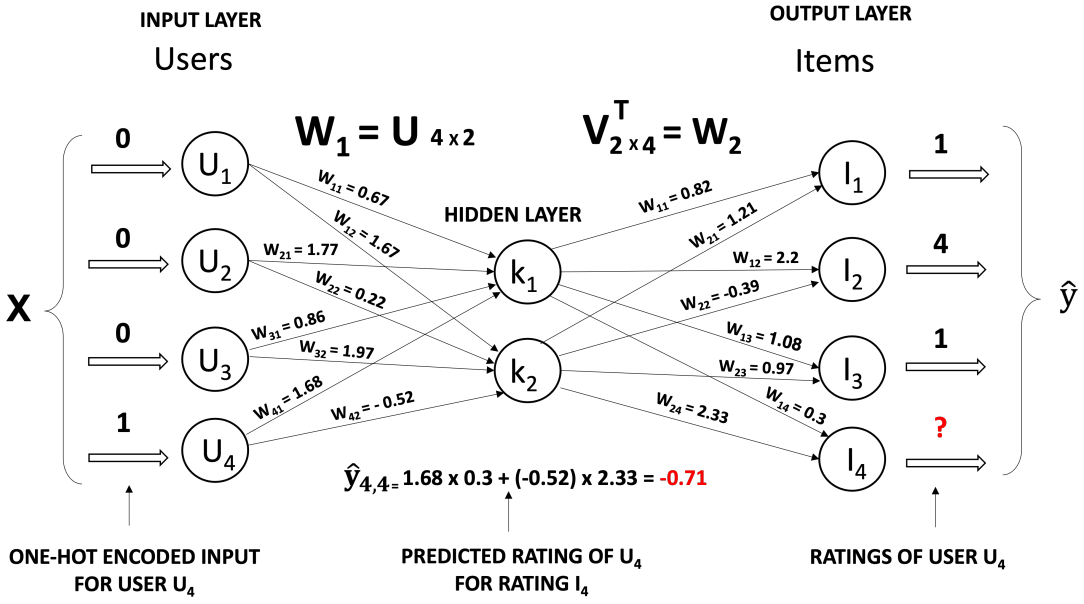


Figure 6.6: Example of a *Multi-Layer Perceptron* to implement *UV-decomposition* based on the data in Example 6.1.

The neural network in Figure 6.6 implements the *UV-decomposition* algorithm, which we described in detail in the previous chapter. Briefly, *UV-decomposition* expresses the original user-item rating matrix A of our current example as the product of two other U and V matrices as shown below:

$$A \approx U \cdot V^T \tag{6.1}$$

More generally, if we have a table A with n rows and m columns, we can compute two tables: the table U with n rows and k columns, and the table V with m rows and k columns, so that the product of the two tables $U \cdot V^{\mathrm{T}}$ approximates the original table A. We emphasize that the variable k controls the **latent vector space** and the size of the U and V matrices.

In the three-layer neural network of Figure 6.6 the U_{4x2} matrix expresses the latent vectors of the users, and is represented by the weighted edges connecting the first layer (input) neurons to the hidden layer. And the matrix V^T_{2x4} represents the **latent** vectors of the items, and is represented by the directed edges with weights connecting the neurons of the hidden layer to the output layer, as shown in Figure 6.6.

Henceforth, the U and V^T matrices will be called *weight* W_1 and W_2 matrices respectively, as shown in Figure 6.6, in order to preserve the terminology of neural networks. Finally, notice that the hidden layer neurons in Figure 6.6 are two, which means that we create a two-dimensional vector space (k=2) by keeping only the two prinipal components from U and V^T respectively.

In the neural network in Figure 6.6 we introduce the input vector X, which is *one-hot encoded*, meaning that the input signal is equal to one (1) only for the target user, and zero (0) for the others. Therefore, for user U_4 the input vector X is {0,0,0,1}, as shown in the left part of Figure 6.6. Then, the *Multi-layer Perceptron (MLP)* of Figure 6.6 projects the input vector X of user U_4 in the two-dimensional vector space of the hidden layer. More precisely, given the initial input vector X, the hidden layer of *MLP* maps it to a hidden representation $Z_1 \in \mathbb{R}^K$ via the following *activation function*:

$$\mathrm{Z}_1 = f\left(\mathrm{W}_1^T * X + \mathrm{b}_1\right) \tag{6.2}$$

where $\mathrm{W}_1 \in \mathbb{R}^{n \times k}$ a weight matrix and $\mathrm{b}_1 \in \mathbb{K}$ the *bias* (bias).

At this point we note that given a *MLP* with l layers, for each layer of *MLP* the input vector is represented in another vector of the latent space of the next layer. The resulting *latent vector representation* of the next layer is depicted as follows:

$$\begin{aligned} \mathrm{Z}_1 &= f\left(\mathrm{W}_1^T * X + \mathrm{b}_1\right) \\ \mathrm{Z}_2 &= f\left(\mathrm{W}_2^T * Z_1 + \mathrm{b}_2\right) \\ &\cdots \\ \mathrm{Z}_l &= f\left(\mathrm{W}_l^T * \mathrm{Z}_{l-1} + \mathrm{b}_l\right) \end{aligned} \tag{6.3}$$

In the *MLP* of Figure 6.6 we have only one hidden layer and therefore compute only two latent vectors (Z_1 expressing the user and Z_2 expressing the item):

$$\begin{aligned} \mathrm{Z}_1 &= f\left(\mathrm{W}_1^T * X + \mathrm{b}_1\right) \\ \mathrm{Z}_2 &= f\left(\mathrm{W}_2^T * Z_1 + \mathrm{b}_2\right) \end{aligned} \tag{6.4}$$

To address the problem of predicting a user's rating over an item we use an *objective function* to complete the missing values of the corresponding user-item rating matrix as follows:

$$\min_{\mathrm{u},\mathrm{v}} \sum_{y_{ij} \in \mathcal{R}} \left(y_{ij} - \mathrm{u}_i \cdot \mathrm{v}_j^T \right)^2 + \lambda_u \cdot \|\mathrm{u}_i\|^2 + \lambda_v \cdot \|\mathrm{v}_i\|^2 \tag{6.5}$$

where $\mathcal{R}$ is the set of observed ratings, y_{ij} is the rating of user i for item j, $\mathbf{u}_i$ is the latent vector of user i, and $\mathbf{v}_j$ is the latent vector of item j, while the parameters λ_u and λ_v "normalize" the L^2 *norm* of $\mathbf{u}$ and $\mathbf{v}$.

Finally, the output of the neural network is the prediction of the ratings $\hat{y}$ of a user for the items. Next, we compare the ratings predicted by the neural network with the actual ratings of user U_4, which are located in the 4th row of the user-item rating matrix A as shown in Figure 6.2. In conclusion, the predicted ratings obtained from the neural network are used to create a user-item rating matrix $\hat{A} = U \cdot V^\top$, which is compared with the original user-item rating matrix A, and if there is a difference, then we correct the weight vectors W_1 and W_2 accordingly. This approach is repeated until the values of the predicted matrix $\hat{A} = U \cdot V^\top$ converge to the point where they no longer vary. Therefore, the neural network shapes the weight vectors W_1 and W_2 accordingly so that, by taking their inner product, it can predict the users' ratings over the items. For example (see in the center of Figure 6.6), we compute the rating of user U_4 over the item I_4 in the following way:

$$\hat{y}_{4,4} = w_{4,1} \cdot w_{1,4} + w_{4,2} \cdot w_{2,4} = 1.68 \cdot 0.3 + (-0.52) \cdot 2.33 = -0.71 \tag{6.6}$$

Therefore, we predict that user U_4 does not like item I_4, which makes sense because user U_4 has similar ratings to user U_2 as shown in Figure 6.2.

Since, now, the neural network in Figure 6.6 consists of a hidden layer of two neurons, it is therefore able to separate the data of our problem by means of an open, *convex solution region.* The hidden two-neuron layer can therefore separate the Cartesian x-y axis plane with two intersecting lines delimiting the aforementioned region. However, if there were a third neuron in the hidden layer, then there would be a third line which would delimit the x-y Cartesian axial plane more clearly, defining a closed, *convex (triangular) solution region.* Consequently, more complex networks of neurons, organized in many layers, are capable of solving complex nonlinearly separable problems.

6.2 Convolutional Neural Networks

A **Convolutional Neural Network - CNN** is a multi-layer perceptron designed specifically to recognize objects from two-dimensional images. The pattern of connectivity between neurons in a *CNN* is inspired by the organization of the visual cortex in organisms, which has a small region of cells that are sensitive only to specific regions of the visual field. By analogy, in an artificial *co-evolutionary*

layer of a CNN each neuron receives input only from a restricted region of the previous layer, which is called **the neuron's receptive field**. The existence of this field therefore allows local features to be extracted from the image. However, once a feature has been extracted, its exact position is less important since its approximate relative position to other features is preserved. The basic computational layers of a *CNN* are three as can be seen in Figure 6.7:

- The **convolution layer** uses a set of filters that detect the presence of specific features or patterns present in the original image given input by generating multiple two dimensional **feature maps**, within which individual neurons are constrained to share similar weight vectors.
- Each *convolution layer* is followed by a **pooling layer** by which the resolution of each feature map is reduced based on an selected aggregation function (e.g., max pooling or average pooling). The reason for the existence of this layer is the gradual reduction in the size of the dimensions, since initially the input is a "big picture", while the output consists of only a few classes of icons of much smaller dimensionality.
- In the end of a CNN, there are several **fully connected layers** having usually Relu as the activation function, where each node in one layer is connected to all nodes in the next layer. The output layer of a CNN, as can be seen in the right part of Figure 6.7 computes a probability for classifying an image to a class (e.g, bird, dog, sunset, etc.), where all probabilities sum up to one.

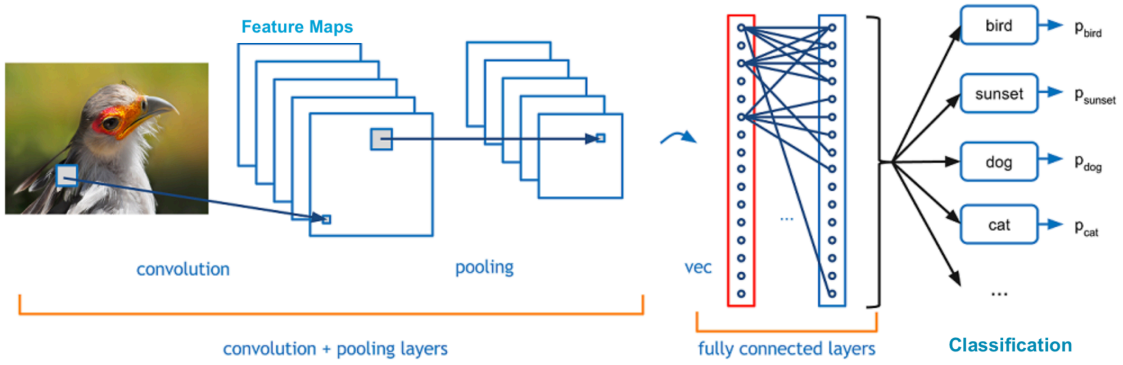


Figure 6.7: Example of a *Convolutional Neural Network*.

CNNs have been successfully applied to various areas of data processing and analysis, such as **Image Processing** and **Natural Language Processing**, achieving significant improvements in terms of producing efficient and correct predictions (e.g. image and face recognition, voice and speech recognition, etc.). The great effectiveness of *CNN* in particular in image processing and recognition has led to its application also in **Graph Data Analytics**. Thus, **Graph Convolutional Networks - GCNs** were developed, which concern the architecture of a neural network using the **adjacency matrix** of a graph and its **node features**, and applies techniques to monitor the parallel evolution of

links and node features of the graph. Since most recommendation systems are "modeled" as bipartite graphs, the *GCN* neural network has attracted the interest of the recommender systems scientific community. The following section discusses the recommendation methods based on *GCNs*.

6.3 Recurrent Neural Networks

A *recurrent neural network* (*RNN*) is distinguished from a *feedforward neural network* (e.g., *MLP*), by the fact that it has at least one *feedback loop*. This is the reason, we call RNNs as feedback neural networks. The feedback loop is a neural synapse (or otherwise, a directed edge), which is connected not to a neuron in the next layer of the neural network, but to a neuron in the same or previous layer. Based on the above characteristic, the main difference between a *CNN or MLP* and a *RNN* is the ability of the latter to process time-dependent data (e.g. time series, word sequences, user sessions, etc.).

The key element of a *RNN* is therefore that there is one input x_t at each time t and one **hidden state** h_t in RNN, which changes as new data instances arrive. Also, at each time point there is a predicted output value $\hat{y}_t$. The term *recurrent* expresses the ability offered by the architecture of *RNNs* to perform the same task for each item of a sequence with the result each time depending on previous computations. Therefore, in this type of neural networks the information has a "memory" influenced by the past.

An important characteristic of *session-based recommender systems*, is that we are mainly dealing with *anonymous sessions* (*user sessions*) and therefore we are not able to create user profiles that express their interactions with items over long periods of time, because these users have not subscribed to the recommendation service. For this reason, the recommendations of these systems are best modeled with *feedback neural networks* so that the system can "forget" past user interactions with items.

Hidasi et al. [Hidasi et al., 2015] presented a recommendation system, known as *GRU4Rec*, which is based on *Gated Recurrent Units* (*GRUs*). More specifically, a *GRU* is a simple neural network that contains *feedback connections* in order to be able to "forget". That is, the *algorithm GRU4Rec* learns when and how to update the *hidden state* of the *GRU* module to "forget" past user interactions with the items. In the same direction, several architectural improvements of the *GRU4REC* (see [Quadrana et al., 2017]) were proposed that led to an increase in the effectiveness of the original *GRU4Rec* algorithm. Also, in the research papers of De Souza Pereira Moreira et al., [De Souza Pereira Moreira et al., 2018] and [Moreira et al., 2019], a recommendation system called *CHAMELEON* was proposed, which uses a *CNN* to exploit the content of articles and a *RNN* with **Long and Short Term Memory** (LSTM) units for the sequential processing of *usage data* related to user interactions with items.

6.4 Genetic Algorithms

The following subsection describes *genetic algorithms* and how they are applied to recommender systems. Holland [Holland, 1992] was the first to present in his book "Adaptation in Natural and Artificial Systems" the *genetic algorithms*, which are based on the mechanisms of (genetic) *evolution* of organisms and in particular on *natural selection*. In particular, Holland and his colleagues were able to design artificial software systems that exploited important mechanisms of the natural functions of organisms, such as **natural selection**, **crossover**, and **mutation** in genetic algorithms. So genetic algorithms follow a search process for the optimal solution, which is guided by a **fitness function**, that evaluates a large number of different possible solutions. We emphasize that the optimal solution search process uses *random selection* as a tool to lead to a high-quality solution. In general, we can describe a *genetic algorithm* as a process of collecting qualitatively good structural components using genetic-type operations. Therefore, the basic idea behind *genetic algorithms* is based on the fact that good solutions are built from good structural components.

6.4.1 The Structural Components of Genetic Algorithms

Genetic algorithms consist of the following structural components:

- **Chromosome Population.**

  In *natural systems* two or more chromosomes are combined to form the overall recipe in order to satisfy the construction instructions and basic functions of organisms. Chromosomes in turn are composed of genes that control specific building blocks of the organism, such as, for example, the gene that controls the eye color of a mammal and is given the value "blue eyes". On the other hand, in *artificial software systems*, structural elements, such as chromosomes, can be defined by a set of binary digits (i.e. 0 and 1). Also, *genetic algorithms* require the set of parameters of an optimization problem to be encoded by a finite-length binary representation, which can be considered a snapshot of a generic schema. The schema $1**1$, for example, represents all possible binary encodings starting and ending with 1 and having length equal to 4. The asterisks in it represent a value that can be 0 or 1. The following Figure 6.8 illustrates a general schema or *mask* of a chromosome and some of its valid *instances*:

  Therefore, solving problems using *genetic algorithms* involves creating an initial population of chromosomes. As the *genetic algorithm* explores the space of possible solutions through the process of *evolution*, the initial chromosomes, which are usually in different locations in the search space, are combined to create the new generation of chromosomes. Thus, the *genetic algorithm* can explore different regions of possible solutions by combining chromosomes through the genetic operations of *selection* and *crossover*.

- **Fitness function**

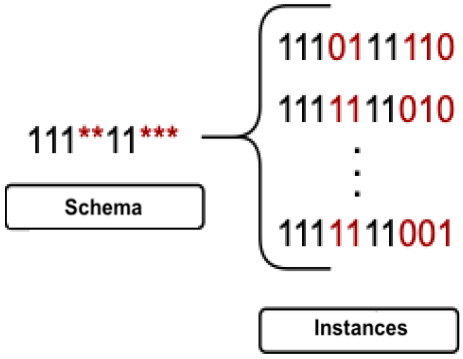


Figure 6.8: A simple example of chromosome instances which are built from a general schema.

In *natural organisms*, health is crucial and exists as an inherent property in order for the organism to be able to survive epidemics and everything that threatens its existence, but also to achieve reproduction. By analogy, in artificial software systems, it is the *fitness function*, which is responsible to decide about the further "life" of *string* creations or their "death" (*extinction*). In other words, *genetic algorithms* require the existence of a function that assigns a degree of fitness to each chromosome of the current population, so that it can be selected as a parent for the creation of the chromosomes of the next population (the next generation). Thus, the fitness of a chromosome depends on how well that chromosome solves the problem under consideration.

6.4.2 Genetic Operations

Genetic algorithms usually support the following basic genetic operations: (a) *reproduction operator* or otherwise, *parent selection* (b) *crossover operator*, and (c) *mutation factor*. We describe them in detail below:

- **(a) Reproduction operator** or **Parental selection operator** The *reproduction* process allows the selection of the best chromosomes (based on a *fitness function*) from the current population, which will produce the children/offspring of the next population. For this reason, the *genetic algorithm* should strike a balance between **exploitation** of the current population by maximizing the cumulative fitness of the chromosomes on the one hand, and **exploration** of the search space by preserving the diversity of the chromosomes on the other hand. In fact, there is the following *trade-off* between *exploitation* and *exploration* of the current population:

- A strict selection of the best chromosomes will reduce the diversity of the next population, which will indeed lead to a fast convergence of the algorithm solutions (from the very first generations), but will usually result in a good but not perfect solution given inefficient exploration.

- A more elastic selection of the best chromosomes leads to a very slow time improvement of the population due to inefficient exploitation.

As a result of the above *trade-off*, different parent selection strategies have been adopted, the most important of which are listed below:

- **Roulette selection** : *Roulette wheel Selection* is a genetic operator used in *genetic algorithms* to select potentially useful individuals (chromosomes) from the population to become the parents of the next population. If f_i is the fitness of individual i in the population based on a *fitness function* f, then its probability of selection is:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

where N is the number of individuals in the population. The main problem of *Roulette wheel Selection*, known as *early convergence*, is the limited exploration of individuals in the population. More specifically, in the early steps of the *genetic algorithm* only a small number of individuals are estimated to be much more competent and suitable than the rest of the individuals. Consequently, this portion of the population is reproduced much more frequently than the rest of the individuals, so that other possible solutions to the problem that might lead to a better solution are not explored.

- σ **scaling selection** [Goldberg, 1989]: The fitness of a chromosome based on σ *scaling selection* is calculated from the *fitness function*, which takes into account the population mean fitness and the population standard deviation. Thus, the σ *scale selection* assumes that the fitness of an individual increases in proportion to its standard deviation from the population mean. This method allows to maintain a stable balance between exploration and exploitation of the population.

- **Boltzmann tournament selection** [Goldberg, 1990]: In contrast to σ *scaling selection*, chromosome selection based on *Boltzmann distribution* evolves over generations, so *the genetic algorithm* starts the optimization with high levels of exploration of possible solutions and ends with similarly high levels of population exploitation.

- **(b) Crossover operator**

The *crossover* allows the exchange of one or more "genes" (i.e. blocks of bits) between parental chromosomes. The simplest form of this operator is **single-point crossover**, which randomly selects a position on the chromosomes by which the operator determines the genes to be exchanged. Next we present a schematic representation of the *single-point crossover operator* that results in two children.

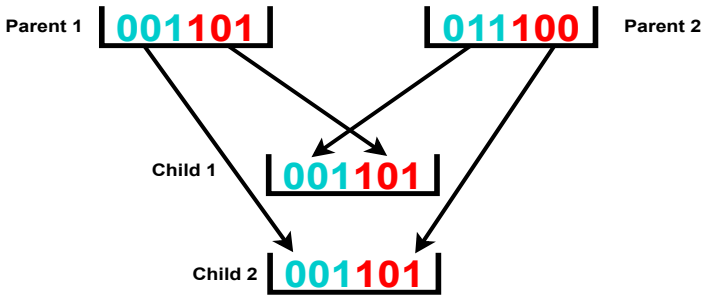


Figure 6.9: Example *single-point crossover factor* for creating 2 ”offspring-children”

However, this operator is limited by two well-known problems: a) the problem of *positional bias* caused by the high correlation of bit positions and their crossover probability, and b) the impossibility of applying crossover to chromosomes of unequal length. Therefore, to mitigate *the problem of positional bias*, the **two-points crossover** operator was proposed, according to which two bits are randomly selected, and based on them the genes to be exchanged can be determined. Furthermore, this operator allows more genetic schemes to be crossed than the single-point crossover. Again, however, two-point crossing cannot be applied to all genetic schemes. Therefore, more statistical methods (such as *Poisson distribution*) were used to select the number of crossover points. Please note that the research work published so far has not established clear guidelines on which particular crossover may be preferred over another.

- **(c) Mutation operator**

The mutation operator allows the value of a bit of the chromosome to be reversed according to some probability, as shown in the figure below:

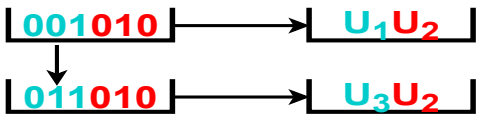


Figure 6.10: Example of a one bit textitmutation operation.

It can also be applied to the bits of any gene or to the *mask* of the *crossover* so that the *crossover* points can evolve during the optimization process. The *interchange* function often protects against irreversible losses of good solutions. In other words, when used judiciously together with *replication* and *crossover* it is a safe *policy* against a premature loss of a good solution. And the frequency of *change* to obtain good results, empirically speaking, is about one change per few thousand bit positions.

6.4.3 Related Work

Genetic algorithms can exhaustively search a large number of different sets of neighborhoods (i.e., nearest users) of the target user in a recommender system. For this reason, they have been used many times in the past to solve the *problem of user clustering*. For example, Rahman and Islam [Rahman and Islam, 2014] modified the *k-means algorithm* so that it can be used together with a *genetic algorithm* to address the known problems such as how to select the initial centers and the number of clusters to be created. The authors of the paper therefore proposed the use of a *fitness function*, which on the one hand minimizes the distances between members within clusters, and on the other hand maximizes the distances between users belonging to different clusters.

In addition, a *genetic algorithm* was used by Bobadilla [Bobadilla et al., 2011] to address the *problem of sparsity of the user-item rating matrix*. Thus, for each pair of users, they computed a user-user similarity matrix with respect to their common preferences for items. Additionally, they computed an item-item similarity matrix, which captures the similarity among items in the same way. Finally, a *fitness function* combined the two aforementioned similarity matrices to select the "best" users as parents to be used for reproducing the next, improved, generation.

Katarya and Verna [Katarya and Verma, 2016], in addition, proposed a *hybrid* user clustering technique consisting of three algorithms: the *k-means algorithm*, the *Particle Swarm Optimization - PSO* and the *fuzzy c-means clustering algorithm*. Initially, users are clustered according to their preferences for the categories of items they interact with. Then, a combination of the *PSO* and *k-means* algorithms is performed to select the initial centers of the user groups being created. Finally, these centers are used as input to the *c-means algorithm* to finalize the created user groups.

Moreover, other *genetic algorithms* have been used to improve the quality of the recommendations both in terms of accuracy and diversity of the proposed items. For example, Zhang and Hurley [Zhang and Hurley, 2009] proposed a *clustering algorithm* to introduce more diversity among the items of the recommendation list. For this reason, they adopted a reordering method of the list of recommended items to increase their diversity.

Finally, Bag et al. [Bag et al., 2019], as well as Berbague et al. [Berbague et al., 2021], proposed a *fitness function* for selecting the *k-nearest neighbors* of the target user, in order to maximize the diversity of the recommended items inside the recommendation list, whereas at the same time they try to maximize the relevance of the nearest users (neighbors) with the target user.

6.4.4 The Architecture of a Genetic Algorithm for Recommendation Systems

Recommender systems provide recommendations that should satisfy different metrics for computing the quality of the recommended items. That is they try to provide accurate and at the same time novel recommendations. A *genetic algorithm*, therefore, in order to meet the above requirements, should use a *fitness function*, which combines two different criteria *relevance* and *diversity*), as shown in the

right-hand side of Figure 7.10:

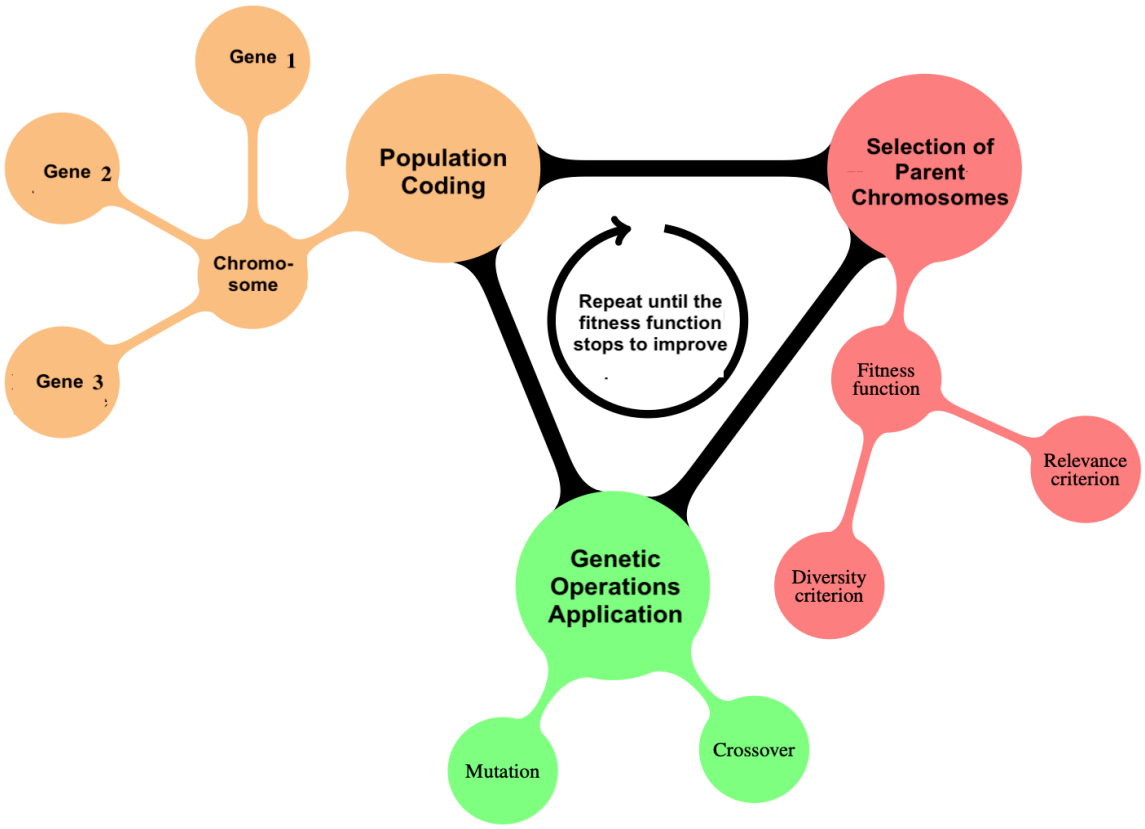


Figure 6.11: The basic architecture of a *genetic algorithm* for recommender systems.

Turning now to the population of chromosomes, the coding of a chromosome often takes the form of a sequence of genes, as shown in the left-hand side of the Figure 7.10. The initial chromosomes evolve over generations with genetic operations such as *selection*, *mutation* and *crossover* to optimize their quality (see in the center of Figure 7.10). This process is repeated until the *fitness function* stops improving or a maximum number of specified steps of the algorithm is reached. As for the other two genetic operations, *crossover* always produces two new chromosomes based on the two parental chromosomes by swapping some of their genes, while *mutation* randomly changes a binary value of a chromosome.

Based on the aforementioned Schema 7.10, each time we select a target user, we randomly generate an initial set of chromosomes. These chromosomes evolve n times, and at each generation the *crossover* and *mutation* genetic operators are applied to the population to generate new chromosomes (i.e., generating each time the *nearest neighbors of the target user*). Then, the new generation is selected independently of the *fitness function*. Finally, the *fitness function* guides the optimization of the users' neighborhood in terms of its closeness to the target user, as well as in terms of the diversity of the items included in this neighborhood.

6.4.5 The Genetic Algorithm in pseudocode form

In this subsection we will present in pseudocode form a general *genetic algorithm* whose basic architecture was described by Fig 7.10. As already mentioned, with the algorithm for optimizing the *neighborhood of the target user*, a new generation (i.e., a new group of different users that constitute its neighborhood) is generated at each iteration until we arrive at the optimal neighborhood. In each generation we apply genetic operations such as *selection*, *mutation* and *crossover* to generate the next generation. The algorithm takes as input the target user u_0 and a set U of users surrounding it, and tries to identify an optimal neighborhood N_0 with the nearest users to u_0.

Algorithm 3 Optimizing the neighborhood of the nearest users.

Input:

$U = \{u_1, u_2, ..., u_n\}$: the set of users.

u_0: the target user.

m: a predefined maximum number of generations to be generated.

f: the size of the initial population.

Output:

N_0: the optimal neighborhood of user u_0 or else the optimal chromosome.

1: population ← Creating a random population (size = f)

2: **While** the number of generations g < m **do**

descendants ← applySelectionOperator(population) Parent Selection

descendants ← applyCrossoverOperator(descendants) Crossover

descendants ← applyMutationOperator(descendants) Mutation

Population ← descendants ∪ Population

g ← g+1

3: **end_while**

4: N_0 ← Select_best_neighborhood (population) {based on their Fitness score}

So, as shown in line 2 of Algorithm 3, the process of optimizing *the neighborhood of the nearest users of the target user* u_0 involves m generations. In each generation g, the following three genetic functions are applied: (1) the *selection* operator of the parents (chromosomes) from the total population, (2) the *crossover* operator for the selected chromosomes, and (3) the *mutation* operator, which is applied to the chromosomes created after the *crossover*. Finally, after optimizing the algorithm for m generations, the best chromosome (*neighborhood of users*) in the population is selected based on its *fitness score*.

6.4.6 Step-by-Step Execution of the Genetic Algorithm

In this section, we gradually apply the *genetic algorithm* described in the previous section to the example data 6.4, which is displayed in Table 6.1.

Example 6.4 You are given the data of Table 6.1, which holds the ratings of 7 users on 9 items.

	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9
u_0	5	4	2	1	5	4	5	?	?
u_1	4	5	2	1	5	2	2	3	5
u_2	4	3	4	5	4	2	1	4	2
u_3	4	1	2	5	2	2	1	2	1
u_4	2	1	5	3	4	5	4	3	2
u_5	5	4	1	1	5	1	4	5	1
u_6	1	2	4	5	2	2	5	1	2

Table 6.1: User-item ratings matrix, where we attempt to predict the rating of user u_0 on items i_8 and i_9.

Suppose we are looking for the optimal neighborhood with the nearest users of the target user u_0. We assume that the neighborhood of the target user u_0 consists of a set of 2 neighboring users (neighborhood size k=2). Thus, we have the set of candidate neighbors $U = \{u_1, u_2, u_3, u_4, u_5, u_6\}$ and the set of items $I = \{i_1, i_2, i_3, i_4, i_5, i_6, i_7\}$ that will be the training data of our prediction model. Each user is identified by her ratings on the items, and each rating is in the range of $[1-5]$. In this example, we want to predict the ratings of u_0 on items i_8 and i_9 (*test data*), which are shown with red English question marks (?) in Table 6.1.

Chromosome encoding

To represent the candidate neighborhood of size $k = 2$ of the target user u_0, we will adopt a simple binary encoding of a chromosome consisting of two genes, so that we have one gene for each candidate member in the neighborhood. Since we have 6 potential neighbors that can be included in the neighborhood of $k = 2$ nearest neighbors of the target user, it is sufficient to represent each gene with only 3 bits. Thus, 001 can represent user u_1, 010 can represent user u_2, and so on. In Figure 6.12 we show 2 examples of neighborhoods based on the chromosome encoding we follow:

Using the above encoding we have 2^6= 64 different binary representations. We emphasize that during the evaluation of a chromosome we should eliminate all unacceptable encodings. For example, we should discard encodings that include redundant neighbors (e.g. $001001 \rightarrow u_1u_1$). On the other hand, we should also discard chromosomes that do not include users belonging to the user set

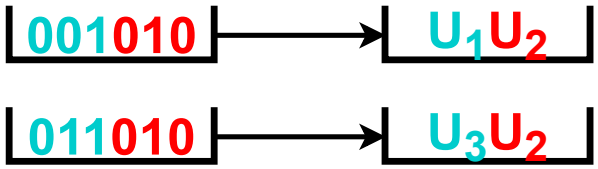


Figure 6.12: Examples of the coding of a chromosome composed of 2 genes representing neighborhoods of size $k = 2$ nearest users.

$U = \{u_1, u_2, .., u_6\}$ ($\pi.\chi.$ $111001 \rightarrow u_7u_1$, $u_7 \notin U$).

Fitness function

To compute the similarity of the target user u_0 with other users, we can empirically assume that, as long as two users rate some items similarly, they may continue to have similar rating behavior for other items in the future. We can adopt this assumption to define a *measure of relevance* or *similarity* between two users u_0 and u_c, as in Equation 6.7:

$$Rel(u_0, u_c) = \frac{\sum_{\substack{|R_{0,i} - R_{c,i}| <= 1 \\ i \in I_{u_c} \cap I_{u_0}}} 1}{|I_{u_c} \cap I_{u_0}|} \tag{6.7}$$

where $R_{0,i}$ and $R_{c,i}$ are the ratings of users u_0 and u_c on item i. Also, I_{u_c} and I_{u_0} are the sets of items rated respectively by users u_c and u_0. Furthermore, in the numerator of Equation 6.7 we count the number of items that both users had rated, where we find that their ratings differed by at most one unit ($|R_{0,i} - R_{c,i}| <= 1$). In the denominator we count the number of items that have been rated by both users. Thus, the *relevance* between the two users is normalized in the interval [0,1].

Now, applying Equation 6.7 to the data of Example 6.4, we compute the similarity vector of the nearest users with the target user u_0 as presented in Table 6.2.

Notice that user u_5 has the highest similarity, equal to 0.85, to the target user u_0. We emphasize that for computing the similarity vector we only consider the ratings for items $i_1, i_2, ...i_7$, while for items i_8 and i_9, we only indent to predict their ratings.

	u_1	u_2	u_3	u_4	u_5	u_6
u_0	0.71	0.57	0.28	0.42	0.85	0.14

Table 6.2: Calculation of the similarity of nearest users to target user u_0. The similarity is defined in the range of values [0,1].

Therefore, based on Equation 6.7 we can evaluate the quality of a given chromosome N by calculating the average similarity between the two users of the chromosome and the target user u_0, as shown in the *fitness function* below:

$$Fitness\_function(u_0, N) = \begin{cases} 0, & N \text{ includes rejected user codings} \\ \frac{1}{|N|} \sum_{v \in N} Rel(v, u_0), & \text{otherwise} \end{cases} \tag{6.8}$$

where N is the size of the neighborhood of target user u_0 or else the size of the chromosome under consideration, which consists of two genes (i.e., two users).

We emphasize that for a number n of users the total number of possible pairs of users per two that can be generated is $\frac{n!}{2!(n-2)!}$. That is, we have $\begin{pmatrix} n \\ 2 \end{pmatrix} = \frac{n!}{2!(n-2)!}$. In our case (Example 6.4), the user pairs that can be generated are 15 since we have six candidate users $U = \{u_1, u_2, u_3, u_4, u_5, u_6\}$. In other words, we have $\begin{pmatrix} 6 \\ 2 \end{pmatrix} = \frac{6!}{2!(6-2)!} = 15$.

After generating all possible pair combinations of users (chromosomes), we apply Equation 6.8 to evaluate the quality of each possible neighborhood with respect to the target user u_0, which is shown in Table 6.3.

Then, by applying the pseudocode of Algorithm 3 to the data of Example 6.4, we obtain Table 6.4 that holds the values of the main genetic operations. So according to the Algorithm 3, we first randomly select the user pairs (chromosomes) to be included in the initial population, which appears in the first row of Table 6.4. As we have 4 user neighborhoods (4 chromosomes) in the original population, we select the 2 most suitable chromosomes (those with the highest fitness score) to replicate, as shown in the 2nd row of Table 6.4. Thus, the 1st generation is shown in row 2 of Table 6.4.

The offspring are created by applying the *crossover* and *mutation* operators, which are described in detail below:

- **Crossover operator:** The neighborhood N_{10} is the combination of u_3 and u_4 (i.e., $N_{10} \leftarrow \{u_3, u_4\}$.), while N_5 is the combination of u_1 and u_6 (i.e., $N_5 \leftarrow \{u_1, u_6\}$.). Using the *crossover operator* we combine u_1 and u_4 to create the neighborhood N_3, and similarly combine u_3 and u_6 to create the neighborhood N_{12}.

- **Mutation Operator:** With *mutation* we can create from the combination $N_3 \leftarrow \{u_1, u_4\}$ the new combination $N_2 \leftarrow \{u_1, u_3\}$ where u_4 is mutated to u_3. Similarly, we create the neighborhood $N_9 \leftarrow \{u_2, u_6\}$ from the combination $N_{12} \leftarrow \{u_3, u_6\}$ where u_3 is mutated to u_2.

In a similar way, then, genetic operations are applied to generate the 2nd and 3rd generations respectively, until we arrive at the optimal generation (see the last row of Table 6.4). Note that in the last column of Table 6.4 the average fitness score of the population slowly improves as new generations are produced. Finally, according to command line 4 of the Algorithm 3, chromosome N_8 will be selected as the optimal neighborhood of the target user u_0, since it has the highest fitness

n	Chromosome	Genes	Fitness score
1	N_1	$\{u_1, u_2\}$	0.64
2	N_2	$\{u_1, u_3\}$	0.49
3	N_3	$\{u_1, u_4\}$	0.56
4	N_4	$\{u_1, u_5\}$	0.78
5	N_5	$\{u_1, u_6\}$	0.42
6	N_6	$\{u_2, u_3\}$	0.42
7	N_7	$\{u_2, u_4\}$	0.49
8	N_8	$\{u_2, u_5\}$	0.71
9	N_9	$\{u_2, u_6\}$	0.35
10	N_{10}	$\{u_3, u_4\}$	0.35
11	N_{11}	$\{u_3, u_5\}$	0.56
12	N_{12}	$\{u_3, u_6\}$	0.21
13	N_{13}	$\{u_4, u_5\}$	0.63
14	N_{14}	$\{u_4, u_6\}$	0.28
15	N_{15}	$\{u_5, u_6\}$	0.49

Table 6.3: For the data in Example 6.4 we have formed user combinations of size $k = 2$ (the so-called neighborhoods) for the 6 candidate users and their corresponding fitness score with respect to the target user u_0.

Generation	Population	Selection	Crossbreeding	Mutation	Fitness score
random	$\{N_{10}, N_5, N_{12}, N_{14}\}$	-	-	-	0.31
1	$\{N_{10}, N_5, N_{12}, N_{14}\}$	$\{N_{10}, N_5\}$	$\{N_3, N_{12}\}$	$\{N_2, N_9\}$	0.31
2	$\{N_{10}, N_5, N_2, N_9\}$	$\{N_2, N_5\}$	$\{N_2, N_5\}$	$\{N_6, N_3\}$	0.40
3	$\{N_3, N_6, N_2, N_5\}$	$\{N_3, N_6\}$	$\{N_7, N_2\}$	$\{N_8, N_2\}$	0.47
optimal	$\{N_3, N_6, N_2, N_8\}$	-	-	-	0.54

Table 6.4: Optimization of the *neighborhood of the nearest users of the target user* of Example 6.4 (m=3)

.

score compared to the other chromosomes in the optimized population, which is shown in the last row of Table 6.4.

6.5 Chapter Questions

1. Draw the basic building blocks and describe the basic function of a *Single-layer Perceptron.*

2. State some basic *activation functions* that can be used in a neural network. What is the difference between *linear* and *non-linear activation functions*?
3. What is the *back propagation* method used for in neural networks? What is its relation to the *gradient descent* method?
4. Draw the basic building blocks and describe the basic function of a *Multi-layer Perceptron*.
5. Which loss function is used for computing the *output* of a *multi-layer Perceptron*? Describe each parameter of the loss function separately.
6. What are the basic layers of a *convolutional neural network* (*CNN*)? Describe the function of each one separately.
7. Describe the basic operation of a *Recurrent Neural Network*. How does it differ from a feed forward network?
8. What are the basic building blocks of *genetic algorithms*? Briefly describe each component separately.
9. What are the basic operations of *genetic algorithms*? Briefly describe each operator separately.
10. Describe the basic steps of the *genetic algorithm* for recommender systems using either a schema or pseudocode.

6.6 Python Programming Exercises

6.6.1 Factorize the user-item rating matrix by using a neural network with Python.

You are given the user-movie rating matrix **A** of Figure 6.13, where columns M_{1-4} represent movies, and rows U_{1-4} refer to users. Implement the *UV-decomposition algorithm* with the help of a neural network and compute the *approximation matrix* $\hat{A}$ of the original matrix A.

	M_1	M_2	M_3	M_4
U_1	4	1	1	4
U_2	1	4	2	0
U_3	2	1	4	5
U_4	1	4	1	?

Figure 6.13: User-Movie Rating Matrix **A** (4×4).

Solution

For this example we will use the *library keras* (*pip install keras*), which implements several types of neural networks. The *library keras* will run as a *frontend environment*, while as a *backend* we will use the *library tensorflow* (*pip install tensorflow*), which contains machine learning algorithms. Therefore, we need to install the above libraries in the *environment* of *Anaconda* where we will run the *Python* code. Furthermore, to visualize the neural network model, we need to install the *library graphviz* (*brew install graphviz*). With the following code we *import* the *libraries keras, tensorflow,* and *graphviz.*

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import warnings
import pydot
import keras
import graphviz
from sklearn.model_selection import train_test_split
from IPython.display import SVG
from tensorflow.keras.optimizers import SGD
from keras.utils.vis_utils import model_to_dot
from sklearn.metrics import mean_absolute_error
from keras.utils.vis_utils import model_to_dot
```

Next, we divide the *data set* of 16 ratings of Figure 6.13 into *train* (containing 12 ratings) and *test* (containing 4 ratings) *subsets.* Then, based on the model we learn from the *train subset,* we will proceed to check our rating predictions with the *test subset.*

```
dataset = pd.read_csv("u2.data",sep='\t',names="user_id,item_id
    ,rating".split(","))
dataset.user_id = dataset.user_id.astype('category').cat.codes.
    values
dataset.item_id = dataset.item_id.astype('category').cat.codes.
    values

print("Dataset \n")
print("User_id        Movie_id        Real_Rating")
for i in range(0,16):
    print(f"{str(dataset.user_id.iloc[i]+1) + chr(9) + chr(9) +
```

```
    str(dataset.item_id.iloc[i]+1) + chr(9) + chr(9) + str(
    dataset.rating.iloc[i])}")

Dataset

User_id         Movie_id         Real_Rating
    1               1               4
    1               2               1
    1               3               1
    1               4               4
    2               1               1
    2               2               4
    2               3               2
    2               4               0
    3               1               2
    3               2               1
    3               3               4
    3               4               5
    4               1               1
    4               2               4
    4               3               1
    4               4               0

train, test = train_test_split(dataset, test_size=0.2)
n_users, n_movies = len(dataset.user_id.unique()), len(dataset.
    item_id.unique())
print("\n train subset \n")
print("User_id         Movie_id         Real_Rating")
for i in range(0,12):
    print(f"{str(train.user_id.iloc[i]+1) + chr(9) + chr(9) +
    str(train.item_id.iloc[i]+1) + chr(9) + chr(9) + str(train
    .rating.iloc[i])}")

print("\n test subset \n")
print("User_id         Movie_id         Real_Rating")
for i in range(0,4):
```

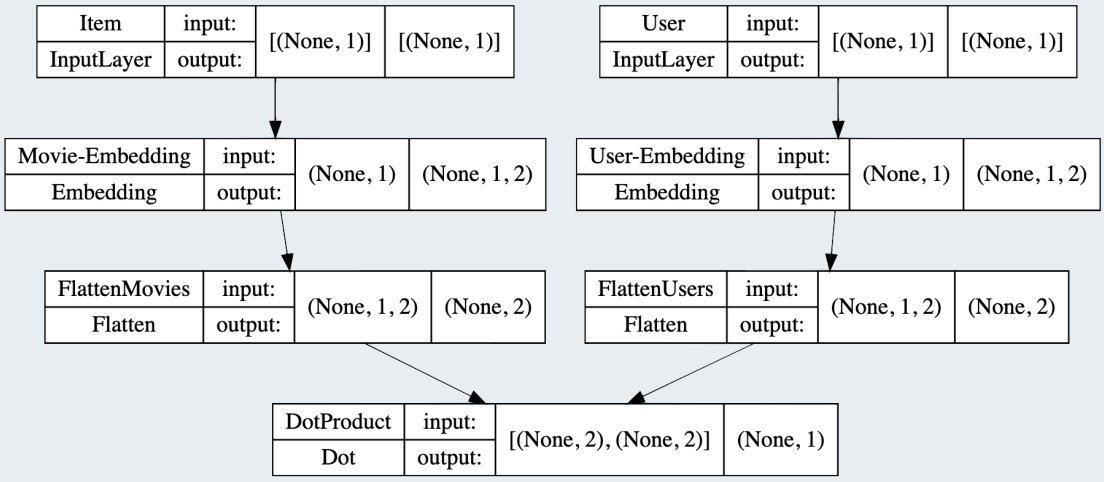
```
    print(f"{str(test.user_id.iloc[i]+1) + chr(9) + chr(9) +
    str(test.item_id.iloc[i]+1) + chr(9) +  chr(9) +  str(test.
    rating.iloc[i])}")

train subset
User_id         Movie_id         Real_Rating
    2               1               1
    3               4               5
    2               4               0
    3               3               4
    1               2               1
    1               4               4
    4               3               1
    4               2               4
    1               1               4
    2               3               2
    2               2               4
    3               2               1

 test subset
User_id         Movie_id         Real_Rating
    1               3               1
    4               1               1
    3               1               2
    4               4               0
```

Next, we try to learn new (low-dimensional) representations of users and movies. These representations are also called *embeddings*. With the following code and the *command keras.layers.Embedding* we create the *user embeddings* (2 latent factors * 4 users) and *movie embeddings* (2 latent factors * 4 movies), respectively. Then, we will combine a user's *embedding* with each movie *embedding* using the inner product operation to predict a user's rating for a movie. In other words, to predict the rating of each user-movie pair we will take the inner product of the corresponding *latent representation* of the user vector and the movie vector. For example, suppose we use $k = 2$ dimensions for the latent vector of users and movies, respectively. These latent vectors may correspond, e.g., to how much a user likes action and romantic movies, while the latent vector of the movie corresponds to how much action and

how much romance there is in the movie.

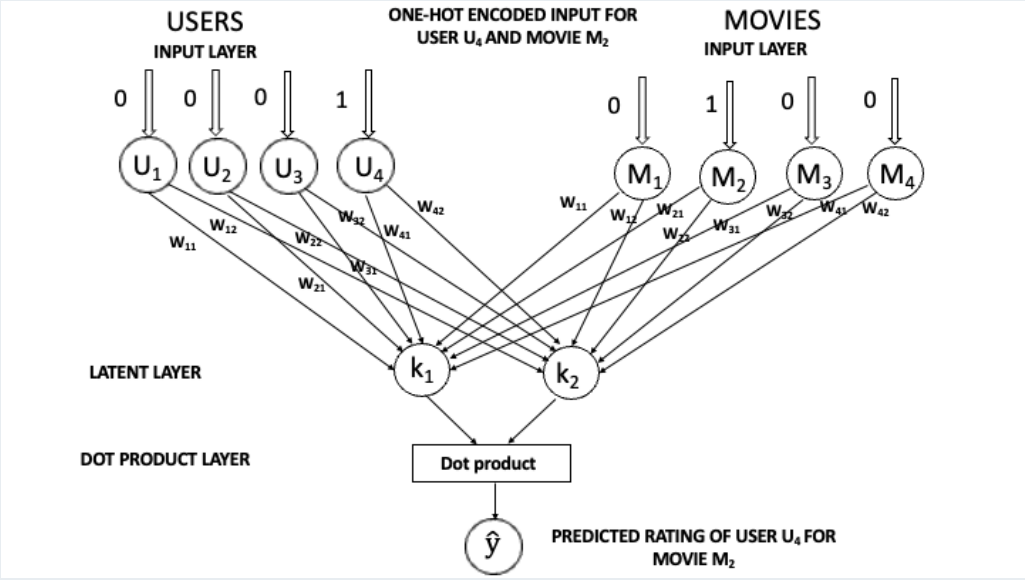
```
n_latent_factors_user = 2
n_latent_factors_movie = 2
movie_input = keras.layers.Input(shape=[1],name='Item')
movie_vec = keras.layers.Flatten(name='FlattenMovies')(keras.
    layers.Embedding(n_movies, n_latent_factors_movie,name='
    Movie-Embedding')(movie_input))
user_input = keras.layers.Input(shape=[1],name='User')
user_vec = keras.layers.Flatten(name='FlattenUsers')(keras.
    layers.Embedding(n_users, n_latent_factors_user,name='User-
    Embedding')(user_input))

prod = keras.layers.dot([movie_vec, user_vec], axes=1, name='
    DotProduct')
model = keras.Model([user_input, movie_input], prod)

SVG(model_to_dot(model, show_shapes=True, show_layer_names=
    True, rankdir='HB').create(prog='dot', format='svg'))
```

Running the above Python code, it displays a schematic representation of our example neural network that implements the UV-decomposition algorithm or else known as *matrix factorization.*



The above model takes two inputs: a user and a movie. To express a user, we use 2 dimensions. Since we have 4 users in our example, we will create a 2 * 4 dimension matrix (users weight

matrix). Similarly, for movies we will create a matrix of dimensions 2 * 4 (movies weight matrix). To further understand the structure of the neural network, we also provide the following Figure:



To see how many total parameters we want to learn in our model, we run the following code:

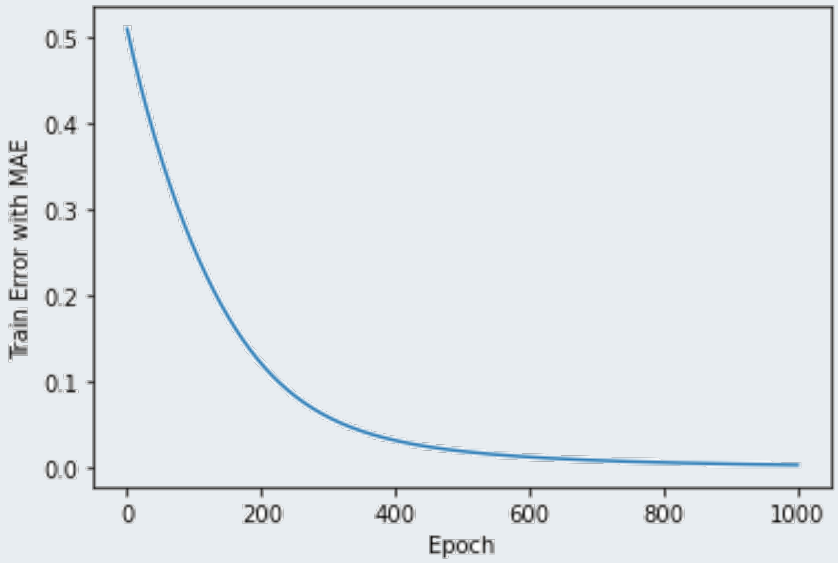
```
model.summary()
```

```
Model: "model_1"
__________________________________________________________________________________________________
 Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
 Item (InputLayer)               [(None, 1)]          0           []

 User (InputLayer)               [(None, 1)]          0           []

 Movie-Embedding (Embedding)     (None, 1, 2)         8           ['Item[0][0]']

 User-Embedding (Embedding)      (None, 1, 2)         8           ['User[0][0]']

 FlattenMovies (Flatten)         (None, 2)            0           ['Movie-Embedding[0][0]']

 FlattenUsers (Flatten)          (None, 2)            0           ['User-Embedding[0][0]']

 DotProduct (Dot)                (None, 1)            0           ['FlattenMovies[0][0]',
                                                                  'FlattenUsers[0][0]']

==================================================================================================
Total params: 16
Trainable params: 16
Non-trainable params: 0
```

As shown in the above Figure, we have 16 parameters to learn in our prediction model, which are analyzed as follows: 2*4 = 8 are for users, while 2*4 = 8 are for movies.
For the purposes of our example and in order to train our model, we will run it for 1000 iterations, or *epochs*, using the following Python code:

```
model.compile('SGD', 'mean_squared_error')
history = model.fit([train.user_id, train.item_id], train.
    rating, epochs=1000, verbose=1)

y_hat = np.round(model.predict([test.user_id, test.item_id]),0)
y_true = test.rating

pd.Series(history.history['loss']).plot()
plt.xlabel("Epoch")
plt.ylabel("Train Error with MAE")

Epoch 1/1000
1/1 [==============================] - 0s 428ms/step - loss:
    0.5089
Epoch 2/1000
1/1 [==============================] - 0s 4ms/step - loss:
    0.5054
Epoch 3/1000
1/1 [==============================] - 0s 3ms/step - loss:
    0.5018
Epoch 4/1000
1/1 [==============================] - 0s 3ms/step - loss:
    0.4983
Epoch 5/1000
1/1 [==============================] - 0s 4ms/step - loss:
....
.....

1/1 [==============================] - 0s 2ms/step - loss:
    0.0028
Epoch 999/1000
1/1 [==============================] - 0s 3ms/step - loss:
    0.0028
Epoch 1000/1000
1/1 [==============================] - 0s 2ms/step - loss:
    0.0028
```

As can be seen from the first line of the above Python code, our model will be optimized using the *Stochastic Gradient Descent* (*SGD*) method, and the metric that will be used to calculate the difference between the user's predicted rating ($\hat{y}$) for a movie versus his/her actual rating (y) is the *Mean Absolute Error*. As can be seen in the following Figure, the Train Error in terms of *MAE* decreases drastically (0.0028 in the training data) after 1000 iterations (*epochs* - *epoch*).



Next, we use the following Python code to print for each user-movie pair the actual and predicted rating for the *train* and *test subsets*:

```
predictions = model.predict([train.user_id.head(12), train.
    item_id.head(12)])
print("\n train subset \n")
print("User_id        Movie_id        Real_Rating      [
    Predicted_Rating]")
for i in range(0,12):
     print(f"{str(train.user_id.iloc[i]+1) + chr(9) + chr(9) +
    str(train.item_id.iloc[i]+1) + chr(9) +  chr(9) +  str(train
    .rating.iloc[i]) + chr(9) + chr(9) + str(predictions[i])  }"
    )

predictions = model.predict([test.user_id.head(4), test.item_id
    .head(4)])
print("\n train subset \n")
print("User_id        Movie_id        Real_Rating      [
    Predicted_Rating]")
```

```
for i in range(0,4):
    print(f"{str(test.user_id.iloc[i]+1) + chr(9) + chr(9) +
   str(test.item_id.iloc[i]+1) + chr(9) + chr(9) + str(test.
   rating.iloc[i]) + chr(9) + chr(9) + str(predictions[i]) }")
```

```
 train subset

User_id       Movie_id        Real_Rating     [Predicted_Rating]
2               4               0               [0.21990767]
4               4               0               [0.05071604]
1               4               4               [4.2736845]
2               1               1               [0.6660578]
4               3               1               [1.047406]
3               4               5               [4.68342]
3               1               2               [2.6894011]
4               2               4               [3.381989]
3               3               4               [3.9802878]
1               1               4               [3.3085825]
1               2               1               [1.6772952]
3               2               1               [0.5219314]

 test subset

User_id       Movie_id        Real_Rating     [Predicted_Rating]
2               2               4               [0.7832108]
2               3               2               [0.4125717]
1               3               1               [3.9893165]
4               1               1               [2.4312117]
```

Next, we provide the Python code that displays the *latent representations* of users, movies and the final predicted user-item rating matrix.

```
movie_embedding_learnt = model.get_layer(name='Movie-Embedding').get_weights()[0]
print("\n The  4*2  weight matrix with the movie embeddings \n")
print(pd.DataFrame(movie_embedding_learnt))
print("\n The  4*2  weight matrix with the user embeddings \n")
user_embedding_learnt = model.get_layer(name='User-Embedding').get_weights()[0]
print(pd.DataFrame(user_embedding_learnt))
print("\n The predicted user-movie rating matrix \n")
print(np.around(np.asmatrix(user_embedding_learnt)*np.asmatrix(movie_embedding_learnt).T,2))
```

The 4*2 weight matrix with the movie embeddings

```
          0         1
0  1.518744  0.657337
1  1.671875 -0.485499
2  1.040667  1.510207
3  0.665299  2.025937
```

The 4*2 weight matrix with the user embeddings

```
          0         1
0  1.475147  1.625061
1  0.456455 -0.041349
2  0.897869  2.016879
3  1.853398 -0.583605
```

The predicted user-movie rating matrix

```
[[3.31 1.68 3.99 4.27]
 [0.67 0.78 0.41 0.22]
 [2.69 0.52 3.98 4.68]
 [2.43 3.38 1.05 0.05]]
```

In summary, as can be seen in the above Figure, the rating prediction for user 4 for movie 4 is 0.05. This rating prediction is reasonable given that User 4's rating behavior is similar to that of User 2 and the latter has rated Movie 4 with zero (0) as shown in Figure 6.13.

Bibliography

S. Bag, A. Ghadge, and M. K. Tiwari. An integrated recommender system for improved accuracy and aggregate diversity. *Computers & Industrial Engineering*, 130:187–197, 2019.

C. E. Berbague, N. E. islem Karabadji, H. Seridi, P. Symeonidis, Y. Manolopoulos, and W. Dhifli. An overlapping clustering approach for precision, diversity and novelty-aware recommendations. *Expert Systems with Applications*, 177:114917, 2021. ISSN 0957-4174. https://doi.org/10.1016/j.eswa.2021.114917. URL https://www.sciencedirect.com/science/article/pii/S0957417421003584.

J. Bobadilla, F. Ortega, A. Hernando, and J. Alcalá. Improving collaborative filtering recommender system results and performance using genetic algorithms. *Knowledge-Based Systems*, 24(8):1310–1316, 2011. 10.1016/j.knosys.2011.06.005. URL https://doi.org/10.1016/j.knosys.2011.06.005.

G. De Souza Pereira Moreira, F. Ferreira, and A. M. da Cunha. News session-based recommendations using deep neural networks. In *Proceedings of the 3rd Workshop on Deep Learning for Recommender Systems*, pages 15–23. ACM, 2018.

D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1989. ISBN 0201157675.

D. E. Goldberg. A note on boltzmann tournament selection for genetic algorithms and population-oriented simulated annealing. *Complex Syst.*, 4, 1990.

B. Hidasi, A. Karatzoglou, L. Baltrunas, and D. Tikk. Session-based recommendations with recurrent neural networks. *CoRR*, abs/1511.06939, 2015. URL http://arxiv.org/abs/1511.06939.

J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

R. Katarya and O. P. Verma. A collaborative recommender system enhanced with particle swarm optimization technique. *Multimedia Tools and Applications*, 75(15):9225–9239, 2016. 10.1007/s11042-016-3481-4. URL https://doi.org/10.1007/s11042-016-3481-4.

P. Moreira, D. Jannach, and A. M. da Cunha. Contextual hybrid session-based news recommendation with recurrent neural networks. *arXiv preprint arXiv:1904.10367*, 2019.

M. Quadrana, A. Karatzoglou, B. Hidasi, and P. Cremonesi. Personalizing session-based recommendations with hierarchical recurrent neural networks. In *Proceedings of the Eleventh ACM Conference on Recommender Systems*, RecSys '17, pages 130–137, New York, NY, USA, 2017. ACM. ISBN

978-1-4503-4652-8. 10.1145/3109859.3109896. URL http://doi.acm.org/10.1145/3109859.3109896.

M. Rahman and M. Islam. A hybrid clustering technique combining a novel genetic algorithm with k-means. *Knowledge-Based Systems*, 71:345–365, 2014. 10.1016/j.knosys.2014.08.011. URL https://doi.org/10.1016/j.knosys.2014.08.011.

M. Zhang and N. Hurley. Novel item recommendation by user profile partitioning. In *2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*. IEEE, 2009. 10.1109/wi-iat.2009.85. URL https://doi.org/10.1109/wi-iat.2009.85.

10