Chapter 14: Part 4

**Deep Learning**

**BM project - part 1: Building RBM class**

Data preprocessing & Building RBM class

**14.4.0.1 Objectives**

Boltzmann Machines can be seen from *two* different *points of view*:

1. An Energy-Based Model
2. A Probabilistic Graphical Model

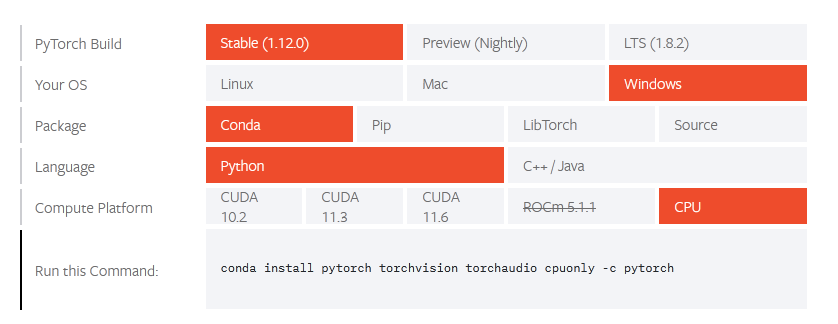
* In the *Intuition Lectures* we focused on the *Energy-Based Model* point of view, and then for the *Practical Lectures* we will focus more on the *Probabilistic Graphical Model* point of view.
* In these last two parts (*Boltzmann Machines* and *AutoEncoders*) of this book, we will create two types of Recommender Systems:

1. One that predicts binary ratings "***Like***" or "***Not Like***". We will build it in this section with a ***Boltzmann*** ***Machine***.
2. Another one that predicts ratings from ***1*** to ***5***. We will build it in next chapter with an ***AutoEncoder***.

* We will implement these two Deep Learning models with *PyTorch*, a highly advanced Deep Learning platform more powerful than *Keras*.
* Every single line of code will be explained in details but I would recommend to have a first look at the *PyTorch* *documentation* to start getting familiar with *PyTorch*:

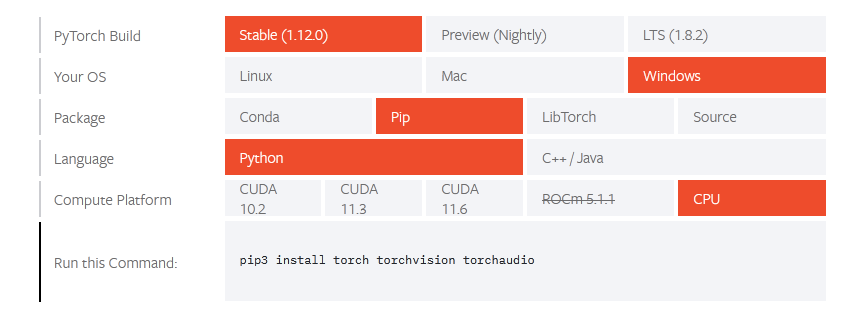
**14.4.0.2 Installing Pytorch**

Using Anaconda (conda)



**conda install pytorch torchvision torchaudio cpuonly -c pytorch**

Using pip:



**pip3 install torch torchvision torchaudio**

* However, following also install PyTorch with anaconda. If you do it, no need to run conda installer. If you want to install in *Local Python directory*, rename the folder "**anaconda3**" in: C:\Users\user\_name
* Rename "**anaconda3**" as "**anaconda3n**" or "**anaconda3bak**" or whatever you want. It is temporary, we'll undo it after Pytorch installed in Local Python.

Microsoft Windows [Version 6.1.7601]

Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\SolLaSi>pip3 install torch torchvision torchaudio

Collecting torch

Downloading torch-1.12.0-cp38-cp38-win\_amd64.whl (161.9 MB)

|████████████████████████████████| 161.9 MB 913 bytes/s

Collecting torchvision

Downloading torchvision-0.13.0-cp38-cp38-win\_amd64.whl (1.1 MB)

|████████████████████████████████| 1.1 MB 731 kB/s

Collecting torchaudio

Downloading torchaudio-0.12.0-cp38-cp38-win\_amd64.whl (969 kB)

|████████████████████████████████| 969 kB 369 kB/s

Requirement already satisfied: typing-extensions in c:\users\sollasi\anaconda3\lib\site-packages (from torch) (3.7.4.3)

Requirement already satisfied: requests in c:\users\sollasi\anaconda3\lib\site-packages (from torchvision) (2.24.0)

Requirement already satisfied: pillow!=8.3.\*,>=5.3.0 in c:\users\sollasi\anaconda3\lib\site-packages (from torchvision)

(8.0.1)

Requirement already satisfied: numpy in c:\users\sollasi\anaconda3\lib\site-packages (from torchvision) (1.22.3)

Requirement already satisfied: certifi>=2017.4.17 in c:\users\sollasi\anaconda3\lib\site-packages (from requests->torchv

ision) (2020.6.20)

Requirement already satisfied: idna<3,>=2.5 in c:\users\sollasi\anaconda3\lib\site-packages (from requests->torchvision)

(2.10)

Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in c:\users\sollasi\anaconda3\lib\site-packages (

from requests->torchvision) (1.25.11)

Requirement already satisfied: chardet<4,>=3.0.2 in c:\users\sollasi\anaconda3\lib\site-packages (from requests->torchvi

sion) (3.0.4)

Installing collected packages: torch, torchvision, torchaudio

Successfully installed torch-1.12.0 torchaudio-0.12.0 torchvision-0.13.0

C:\Users\SolLaSi>

PyTorch Documentation:

<https://pytorch.org/docs/master/>

* *Another method (Anaconda/Conda):* Without creating virtual environment in Anaconda/Conda: No need to use pip3 installer.

1. Open anaconda Powershell prompt, run as adminstrator
2. Run following codes to install TensorFlow in base(root):

**conda** activate base

**conda** **install** pytorch torchvision torchaudio cpuonly -c pytorch

(base) PS C:\Windows\system32> conda activate base

(base) PS C:\Windows\system32> conda install pytorch torchvision torchaudio cpuonly -c pytorch

Collecting package metadata (current\_repodata.json): done

Solving environment: |

The environment is inconsistent, please check the package plan carefully

The following packages are causing the inconsistency:

- defaults/win-64::anaconda==2020.11=py38\_0

- defaults/win-64::astropy==4.0.2=py38he774522\_0

- defaults/win-64::bkcharts==0.2=py38\_0

- defaults/win-64::bokeh==2.2.3=py38\_0

- defaults/win-64::bottleneck==1.3.2=py38h2a96729\_1

- defaults/noarch::dask==2.30.0=py\_0

- defaults/win-64::h5py==2.10.0=py38h5e291fa\_0

- defaults/noarch::imageio==2.9.0=py\_0

- defaults/win-64::matplotlib==3.3.2=0

- defaults/win-64::matplotlib-base==3.3.2=py38hba9282a\_0

- defaults/win-64::mkl\_fft==1.2.0=py38h45dec08\_0

- defaults/win-64::mkl\_random==1.1.1=py38h47e9c7a\_0

- defaults/win-64::numba==0.51.2=py38hf9181ef\_1

- defaults/win-64::numexpr==2.7.1=py38h25d0782\_0

- defaults/win-64::numpy==1.19.2=py38hadc3359\_0

- defaults/win-64::pandas==1.1.3=py38ha925a31\_0

- defaults/win-64::patsy==0.5.1=py38\_0

- defaults/win-64::pytables==3.6.1=py38ha5be198\_0

- defaults/win-64::pywavelets==1.1.1=py38he774522\_2

- defaults/win-64::scikit-image==0.17.2=py38h1e1f486\_0

- defaults/win-64::scikit-learn==0.23.2=py38h47e9c7a\_0

- defaults/win-64::scipy==1.5.2=py38h14eb087\_0

- defaults/noarch::seaborn==0.11.0=py\_0

- defaults/win-64::statsmodels==0.12.0=py38he774522\_0

- defaults/win-64::tifffile==2020.10.1=py38h8c2d366\_2

failed with initial frozen solve. Retrying with flexible solve.

Solving environment: failed with repodata from current\_repodata.json, will retry with next repodata

source.

Collecting package metadata (repodata.json): done

Solving environment: -

The environment is inconsistent, please check the package plan carefully

The following packages are causing the inconsistency:

- defaults/win-64::anaconda==2020.11=py38\_0

- defaults/win-64::astropy==4.0.2=py38he774522\_0

- defaults/win-64::bkcharts==0.2=py38\_0

- defaults/win-64::bokeh==2.2.3=py38\_0

- defaults/win-64::bottleneck==1.3.2=py38h2a96729\_1

- defaults/noarch::dask==2.30.0=py\_0

- defaults/win-64::h5py==2.10.0=py38h5e291fa\_0

- defaults/noarch::imageio==2.9.0=py\_0

- defaults/win-64::matplotlib==3.3.2=0

- defaults/win-64::matplotlib-base==3.3.2=py38hba9282a\_0

- defaults/win-64::mkl\_fft==1.2.0=py38h45dec08\_0

- defaults/win-64::mkl\_random==1.1.1=py38h47e9c7a\_0

- defaults/win-64::numba==0.51.2=py38hf9181ef\_1

- defaults/win-64::numexpr==2.7.1=py38h25d0782\_0

- defaults/win-64::numpy==1.19.2=py38hadc3359\_0

- defaults/win-64::pandas==1.1.3=py38ha925a31\_0

- defaults/win-64::patsy==0.5.1=py38\_0

- defaults/win-64::pytables==3.6.1=py38ha5be198\_0

- defaults/win-64::pywavelets==1.1.1=py38he774522\_2

- defaults/win-64::scikit-image==0.17.2=py38h1e1f486\_0

- defaults/win-64::scikit-learn==0.23.2=py38h47e9c7a\_0

- defaults/win-64::scipy==1.5.2=py38h14eb087\_0

- defaults/noarch::seaborn==0.11.0=py\_0

- defaults/win-64::statsmodels==0.12.0=py38he774522\_0

- defaults/win-64::tifffile==2020.10.1=py38h8c2d366\_2

done

## Package Plan ##

environment location: C:\Users\SolLaSi\anaconda3

added / updated specs:

- cpuonly

- pytorch

- torchaudio

- torchvision

The following packages will be downloaded:

package | build

---------------------------|-----------------

\_anaconda\_depends-2020.07 | py38\_0 6 KB

anaconda-custom | py38\_1 36 KB

certifi-2022.6.15 | py38haa95532\_0 153 KB

conda-4.12.0 | py38haa95532\_0 14.5 MB

cpuonly-2.0 | 0 2 KB pytorch

gmpy2-2.1.2 | py38h7f96b67\_0 160 KB

libllvm9-9.0.1 | h21ff451\_0 61 KB

libuv-1.40.0 | he774522\_0 255 KB

mpc-1.1.0 | h7edee0f\_1 260 KB

mpfr-4.0.2 | h62dcd97\_1 1.5 MB

mpir-3.0.0 | hec2e145\_1 1.3 MB

openssl-1.1.1q | h2bbff1b\_0 4.8 MB

pytorch-1.12.0 | py3.8\_cpu\_0 133.7 MB pytorch

pytorch-mutex-1.0 | cpu 3 KB pytorch

snappy-1.1.9 | h6c2663c\_0 2.2 MB

tbb-2021.5.0 | h59b6b97\_0 149 KB

torchaudio-0.12.0 | py38\_cpu 3.5 MB pytorch

torchvision-0.13.0 | py38\_cpu 6.2 MB pytorch

------------------------------------------------------------

Total: 168.7 MB

The following NEW packages will be INSTALLED:

\_anaconda\_depends pkgs/main/win-64::\_anaconda\_depends-2020.07-py38\_0

cpuonly pytorch/noarch::cpuonly-2.0-0

gmpy2 pkgs/main/win-64::gmpy2-2.1.2-py38h7f96b67\_0

libllvm9 pkgs/main/win-64::libllvm9-9.0.1-h21ff451\_0

libuv pkgs/main/win-64::libuv-1.40.0-he774522\_0

mpc pkgs/main/win-64::mpc-1.1.0-h7edee0f\_1

mpfr pkgs/main/win-64::mpfr-4.0.2-h62dcd97\_1

mpir pkgs/main/win-64::mpir-3.0.0-hec2e145\_1

numpy-base pkgs/main/win-64::numpy-base-1.19.2-py38ha3acd2a\_0

pytorch pytorch/win-64::pytorch-1.12.0-py3.8\_cpu\_0

pytorch-mutex pytorch/noarch::pytorch-mutex-1.0-cpu

snappy pkgs/main/win-64::snappy-1.1.9-h6c2663c\_0

tbb pkgs/main/win-64::tbb-2021.5.0-h59b6b97\_0

torchaudio pytorch/win-64::torchaudio-0.12.0-py38\_cpu

torchvision pytorch/win-64::torchvision-0.13.0-py38\_cpu

The following packages will be UPDATED:

ca-certificates 2020.10.14-0 --> 2022.4.26-haa95532\_0

certifi pkgs/main/noarch::certifi-2020.6.20-p~ --> pkgs/main/win-64::certifi-2022.6.15-

py38haa95532\_0

conda 4.9.2-py38haa95532\_0 --> 4.12.0-py38haa95532\_0

openssl 1.1.1h-he774522\_0 --> 1.1.1q-h2bbff1b\_0

The following packages will be DOWNGRADED:

anaconda 2020.11-py38\_0 --> custom-py38\_1

Proceed ([y]/n)? y

Downloading and Extracting Packages

openssl-1.1.1q | 4.8 MB | #################################### | 100%

pytorch-mutex-1.0 | 3 KB | #################################### | 100%

cpuonly-2.0 | 2 KB | #################################### | 100%

libuv-1.40.0 | 255 KB | #################################### | 100%

libllvm9-9.0.1 | 61 KB | #################################### | 100%

snappy-1.1.9 | 2.2 MB | #################################### | 100%

mpc-1.1.0 | 260 KB | #################################### | 100%

torchaudio-0.12.0 | 3.5 MB | #################################### | 100%

\_anaconda\_depends-20 | 6 KB | #################################### | 100%

mpir-3.0.0 | 1.3 MB | #################################### | 100%

pytorch-1.12.0 | 133.7 MB | #################################### | 100%

torchvision-0.13.0 | 6.2 MB | #################################### | 100%

mpfr-4.0.2 | 1.5 MB | #################################### | 100%

gmpy2-2.1.2 | 160 KB | #################################### | 100%

anaconda-custom | 36 KB | #################################### | 100%

tbb-2021.5.0 | 149 KB | #################################### | 100%

certifi-2022.6.15 | 153 KB | #################################### | 100%

conda-4.12.0 | 14.5 MB | #################################### | 100%

Preparing transaction: done

Verifying transaction: done

Executing transaction: done

(base) PS C:\Windows\system32>

* Verification: To ensure that PyTorch was installed correctly, we can verify the installation by running sample PyTorch code. Here we will construct a randomly initialized tensor.

From the command line, type:

python

then enter the following code:

import torch

x = torch.rand(5, 3)

print(x)

The output should be something similar to:

tensor([[0.3380, 0.3845, 0.3217],

[0.8337, 0.9050, 0.2650],

[0.2979, 0.7141, 0.9069],

[0.1449, 0.1132, 0.1375],

[0.4675, 0.3947, 0.1426]])

Additionally, to check if your GPU driver and CUDA is enabled and accessible by PyTorch, run the following commands to return whether or not the CUDA driver is enabled:

import torch

torch.cuda.is\_available()

* Use Conda powershell prompt:

(base) PS C:\Windows\system32> python

Python 3.8.5 (default, Sep 3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32

Type "help", "copyright", "credits" or "license" for more information.

>>> import torch

>>> x = torch.rand(5, 3)

>>> print(x)

tensor([[0.0298, 0.0806, 0.8720],

[0.0490, 0.3365, 0.4080],

[0.7628, 0.1640, 0.0817],

[0.0627, 0.7596, 0.3849],

[0.1151, 0.4519, 0.9773]])

>>>

|  |  |
| --- | --- |
| * No Anaconda (Local Python): Intalling on **C:\Users\SolLaSi\AppData\Local\Programs\Python\Python38**. Alongside ***anaconda***. * Without anaconda or *No Anaconda environment*. (if any "***anaconda***" version installed previously)   Note: If ***Anaconda3*** or any version installed in your pc, goto: ***C:\Users\user\_name*** in my case ***C:\Users\SolLaSi*** | |
| * Find the folder named " ***anaconda3***" rename it, so that ***pip*** *doesn't* ***locate*** *it* and by default it uses ***local python directory***. * Rename "***anaconda3***" as "***anaconda3n***" or "***anaconda3bak***" or whatever you want. It is temporary, we'll undo it after *Pytorch* installed in *Local Python*. |  |
| **pip install torch**  **pip install torchvision**  **pip install torchaudio** | |

**Microsoft Windows [Version 6.1.7601]**

**Copyright (c) 2009 Microsoft Corporation. All rights reserved.**

**C:\Users\SolLaSi>pip install torch**

**Collecting torch**

**Using cached torch-1.12.0-cp38-cp38-win\_amd64.whl (161.9 MB)**

**Requirement already satisfied: typing-extensions in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-pac**

**kages (from torch) (4.2.0)**

**Installing collected packages: torch**

**Successfully installed torch-1.12.0**

**WARNING: There was an error checking the latest version of pip.**

**C:\Users\SolLaSi>pip install torchvision**

**Collecting torchvision**

**Using cached torchvision-0.13.0-cp38-cp38-win\_amd64.whl (1.1 MB)**

**Requirement already satisfied: torch==1.12.0 in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-package**

**s (from torchvision) (1.12.0)**

**Requirement already satisfied: requests in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-packages (fr**

**om torchvision) (2.27.1)**

**Requirement already satisfied: numpy in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-packages (from**

**torchvision) (1.22.3)**

**Requirement already satisfied: pillow!=8.3.\*,>=5.3.0 in c:\users\sollasi\appdata\local\programs\python\python38\lib\site**

**-packages (from torchvision) (9.1.0)**

**Requirement already satisfied: typing-extensions in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-pac**

**kages (from torchvision) (4.2.0)**

**Requirement already satisfied: idna<4,>=2.5 in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-packages**

**(from requests->torchvision) (3.3)**

**Requirement already satisfied: certifi>=2017.4.17 in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-pa**

**ckages (from requests->torchvision) (2021.10.8)**

**Requirement already satisfied: urllib3<1.27,>=1.21.1 in c:\users\sollasi\appdata\local\programs\python\python38\lib\site**

**-packages (from requests->torchvision) (1.26.9)**

**Requirement already satisfied: charset-normalizer~=2.0.0 in c:\users\sollasi\appdata\local\programs\python\python38\lib\**

**site-packages (from requests->torchvision) (2.0.12)**

**Installing collected packages: torchvision**

**Successfully installed torchvision-0.13.0**

**WARNING: There was an error checking the latest version of pip.**

**C:\Users\SolLaSi>pip install torchaudio**

**Collecting torchaudio**

**Using cached torchaudio-0.12.0-cp38-cp38-win\_amd64.whl (969 kB)**

**Requirement already satisfied: torch==1.12.0 in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-package**

**s (from torchaudio) (1.12.0)**

**Requirement already satisfied: typing-extensions in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-pac**

**kages (from torch==1.12.0->torchaudio) (4.2.0)**

**Installing collected packages: torchaudio**

**Successfully installed torchaudio-0.12.0**

**WARNING: There was an error checking the latest version of pip.**

**C:\Users\SolLaSi>**

Verification:

**C:\Users\SolLaSi>python**

**Python 3.8.10 (tags/v3.8.10:3d8993a, May 3 2021, 11:48:03) [MSC v.1928 64 bit (AMD64)] on win32**

**Type "help", "copyright", "credits" or "license" for more information.**

**>>> import torch**

**>>> x = torch.rand(5, 3)**

**>>> print(x)**

**tensor([[0.0342, 0.2619, 0.0458],**

**[0.4530, 0.1518, 0.9431],**

**[0.3442, 0.6217, 0.1544],**

**[0.8963, 0.9547, 0.4306],**

**[0.7427, 0.4610, 0.1228]])**

**>>>**

* Now rename "***anaconda3n***" to "***anaconda3*** " or again.

**14.4.1 Problem description**

* We are gonna make a *recommender system* that will *predict* if a user is going to like a movie: *yes* or *no*. This one predicts a binary outcome, ***1*** or ***0***, that is ***yes*** or ***no***.
* We'll build another *recommender* *system* that is going to *predict* the *rating* of a movie by a user using *Auto Encoders* in next chapter (predicts a rating from ***1*** to ***5***).
* And this way you have the *two recommender systems* that are mostly used in the *industry*.
* So we're gonna make the recommender system that predicts a *binary outcome*: yes or no with our Restricted Boltzmann machines (RBM).
* Dataset: For both these recommended systems (with RBM and AE) we're gonna start with the same data set. This is the most real world data set that you can find online. It is called MovieLens.
* We downloaded the dataset for ***100k*** and ***1m***. We extracted the files in following folders: " ***movie\_lens\_100k***" and " ***movie\_lens\_1m***".
* There are different size dataset in MovieLens. In our Project we use 100k and 1million ratings dataset.

The dataset source is MovieLens: <https://grouplens.org/datasets/movielens/>

MovieLens 25 million: <https://files.grouplens.org/datasets/movielens/ml-25m.zip>

MovieLens 10 million: <https://files.grouplens.org/datasets/movielens/ml-10m.zip>

MovieLens 1 million: <https://files.grouplens.org/datasets/movielens/ml-1m.zip>

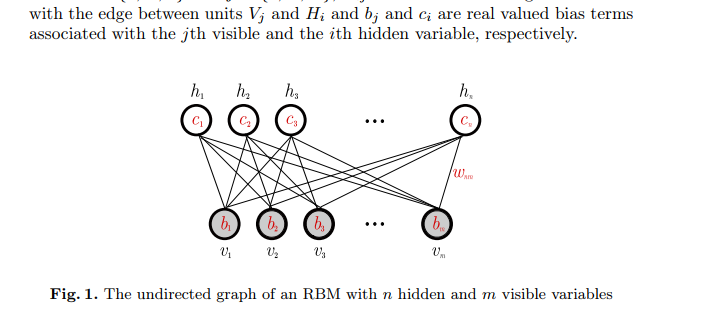
MovieLens 100k: <https://files.grouplens.org/datasets/movielens/ml-100k.zip>

**14.4.2 Overview of the project**

For two different recommender systems with RBM and AE (next chapter), we have a common data preprocessing phase.

1. We're going to *import* the data set,
2. prepare the *training* *set* and the *test* *set*.
3. We're going to get the *number of users* and the *number of movies*.
4. And then we're going to *convert* our *data* into an *array* where we have our *users* in *lines*, and *movies* in *columns*.
5. And finally, in the last step of this data preprocessing phase, we will convert the data into ***torch.Tensor***.

* After these ***data preprocessing*** steps we will start the steps that are specific to ***Boltzmann machines***. We're going to start dealing with ***binary ratings***, then create an *architecture of a neural network*, that will be a ***Restricted Boltzmann Machine (RBM)*** and note that, it is a Probabilistic Graphical Model.
* We have our two data sets, the *1 million ratings data set*, and the *100K ratings data set*.
* Remember, we're going to train our RBM on this one, the one 100k ratings, but of course you can practice on this data set as well, if you want to evaluate more the performance.
* ***AItRBM:*** We have this PDF file named ***AItRBM-proof.pdf***, also available online, that contains all the *theory behind the* *neural* *network* that we are about to make. So I strongly encourage to have a look at this *PDF file*, if you can read it, that's really, *really good*. It contains all the *theory*, *explains* on the *intuitive level*, but also it goes into the *math* pretty much in *detailed*.
* Here all you need to know about graphical models, because a Restricted Boltzmann Machine (RBM) is a Probabilistic Graphical Model.
* Here you have the theory of *graphical models*, with *Undirected Graphs* and *Markov Random* *Fields*.
* You have a section on *Unsupervised Learning*.
* You have the mathematical details on the *computations of the likelihood*.
* The *KL-divergence*, and some optimization theory, which is of course very useful for ***Boltzmann machines***.
* And then you have some theory about MCMC techniques (Markov chain Monte Carlo techniques), very important, because we're going to use Gibbs sampling to estimate the gradient of the likelihood, and Gibbs sampling is based on MCMC.
* You have the definition of a Markov chain, if you want to go deeper in mathematics,
* And then you have of course Gibbs sampling that's very important, at the heart of Boltzmann machines,
* We will remind the intuition behind *Gibbs sampling*, when we *implement* our *Boltzmann machines*.



* Finally, you have, of course, our Restricted Boltzmann Machines (RBM).
* There is the formula for the energy that we're going to try to *minimize*, because we're trying to *minimize* the *free energy*, and that is by *maximizing* the *log likelihood*.
* Here you have the *architecture* of the *Restricted Boltzmann Machines*, with the *visible nodes*, that's our *input*, which are going to be the *ratings* of the *movies* by the *users*, and
* The users are going to be the *different* *observations* going into the *network*, one by one.
* You have the *hidden* *nodes*, and so all this makes the architecture of the RBM, and that's exactly what we're going to make, in Python, by building a class. And mostly this class will contain the *Contrastive Divergence technique*, that will be used to *maximize* the *likelihood*.

|  |  |
| --- | --- |
| * In this class, that we're going to call *RBM*, we will implement the *Contrastive* *Divergence* technique. We will implement the following algorithm, ***k-step contrastive divergence*** this is this algorithm, the heart of our RBM. |  |

**14.4.3 Data Preprocessing 1 : Train & Test set**

Now we're going to import the libraries that will be using to implement our RBM,

# *Importing the libraries*

**import** pandas **as** pd

**import** numpy **as** np

**import** torch

**import** torch.nn **as** nn

**import** torch.nn.parallel

**import** torch.optim **as** optim

**import** torch.utils.data

**from** torch.autograd **import** Variable

1. ***NumPy***, because we will be working with ***NumPy*** arrays.
2. ***Pandas*** to import the dataset and create the *training set* and the *test set*,

* And then we have all the Torch libraries,

1. So, for example this ***nn*** is the module of *Torch* to implement *Neural Networks*,
2. ***parallel*** for the *Parallel Computations*,
3. ***optim*** for the *Optimizer*,
4. ***Variable*** is for *Stochastic Gradient Descent*.

* Importing Movie dataset: Now we're going to import the dataset. This will be different, because we'll import the dataset part-by-part, because the dataset is not that simple and we'll need to use some of the arguments:
* The first dataset we're going to import is all our movies. We have to import ***movies.dat***.

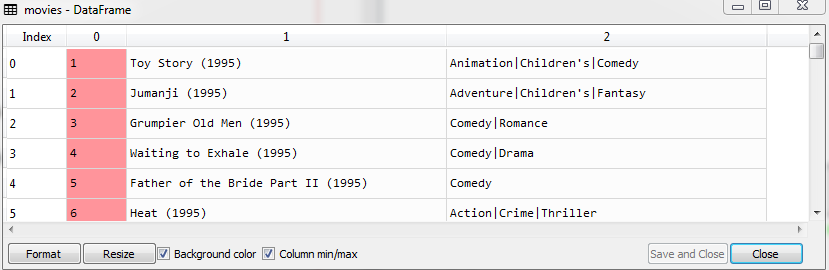
# *importing the dataset*

movies = **pd.read\_csv**("./movie\_lens\_1m/movies.dat", sep= "::", header=None, engine="python", encoding="latin-1")

* The path that contains the dataset, **"./movie\_lens\_1m/movies.dat"**
* The separator the *default separator* is a ***comma***, that works for ***csv*** file, where the features are *separated* by *commas*.
* But here that's not the case because some of the titles of the movies contain *commas* *inside the title*, hence we cannot use the *comma* as a *separator*. Therefore the separator is a ***double colon***, like this **'::'**. So **sep= "::"**
* If you open the *movies.dat* file, you will see that the movies are *separated* by their *ratings* and their other *features*, by a *double colon* like this.
* The third parameter is the header. Because actually the file movies.dat, doesn't contain ***headers***, that is, ***names of columns***. And therefore we need to specify this because, the default value of header is not ***None*** (no column names), and therefore we need to specify that there is no column names, and to do this we put **header=None**.
* Then the next parameter is going to be ***engine***, and this is to make sure that the dataset gets imported correctly. And we will use the Python engine, ***engine = "python"*** .
* Last argument, is the encoding. And we need to input a different encoding than usual because some of the movie titles contain special characters that cannot be treated properly with the classic encoding, ***UTF-8***. So, we're just adding this encoding argument because of some of the special characters in the movie titles. ***encoding = "latin-1"***.

***movies = pd.read\_csv("./movie\_lens\_1m/movies.dat", sep= "::", header=None, engine="python", encoding="latin-1")***

* Notice we didn't use ***header="None"***, because ***"None"*** is string, and ***None*** is a data-type.



* In this ***Movieland*** database, we have thousands of movies, and for each of these movies you have this ***first column*** which is the ***movie ID***, we will use the ***movie ID*** to make a *recommender system*, we will *not* be using the *titles*. It will be much *more simple* with the *movies' IDs*.
* Actually we will not be using this dataset to make the ***training set*** or the ***test set***. It is just to show you what's going on with all the movies.

|  |  |
| --- | --- |
| * Users data-set: Same As above we just *copy-paste* and give *new variable*. All augments are same, we just need to change the file path. * These are all the information is about the different users. * *First column* is the ***user ID***, * the *second column* is the ***gender***, * the *third column* is the ***age***, * the *fourth column* is some codes that correspond to the ***user's job***, and * the *last column* is the ***zip code***. |  |

|  |  |
| --- | --- |
| * User ratings dataset: We just *copy-paste* and give *new variable*. All augments are same, we just need to change the file path. * Now its really *important* to understand the *structure*, because we are getting closer to the *training* *set* and the *test* *set* we'll make, to train our model. * The *first column*corresponds to the ***users***. So this ***1*** here that we see, corresponds to the *first user* of the *database*. So all these ***1's*** here correspond to the *same user*. * Then the *second column* corresponds to the ***movies***. And the *numbers* (1193, 661 etc) that we see here are the ***movies' IDs***. * That are contained in the ***movies*** DataFrame, and so that's why we imported this DataFrame, it's for you to see which movie IDs corresponds to which movie, just if you want to play or test the recommender system in the end. |  |

* The third column corresponds, of course, to the ***ratings***. So the ratings go from ***one*** to ***five***, ***1*** means that the user ***didn't like*** the movie, and ***5*** means that user ***absolutely loved*** the movie.
* For example, this second line here of index 2, means that, the user number 1, rated the movie number 914, and gave it 3 stars.
* The *fourth* *column* we absolutely *don't care*, these are just the ***timestamps***, that basically specifies *when each user* *rated* the *movie*. We will remove this data afterwards when creating the ***training set*** and the ***test set***.

# *importing the dataset*

movies = **pd.read\_csv**("./movie\_lens\_1m/movies.dat", sep= "::", header=**None**, engine="python", encoding="latin-1")

useRs = **pd.read\_csv**("./movie\_lens\_1m/users.dat", sep= "::", header=**None**, engine="python", encoding="latin-1")

RaTings = **pd.read\_csv**("./movie\_lens\_1m/ratings.dat", sep= "::", header=**None**, engine="python", encoding="latin-1")

* Training set and the test set: Now we're going to prepare the ***training set*** and the ***test set***.
* We are going to take the *100k ratings* data set, which is in the ***movie\_lens\_100k*** folder. It is the five train-test-split of the whole dataset composed of *100,000 ratings*.
* So as you can see, we have we have ***u1.base*** and ***u1.test***; ***u2.base*** and ***u2.test***; . . . . ; ***u5.base*** and ***u5.test***. Each one of those pairs of sets are actually some separate ***training set*** (base means training set) and ***test set***. These splits will help us to use K-fold CV. In this case we use *5-fold* instead of *10-fold*.
* For now we only use ***u1.base*** and ***u1.test*** as *train-set* and *test-set*.
* Fist we import data as ***data-frame*** and later we convert it to an ***array***.
* The separator for this U1 based file, in this case *not a double colon*, but a *tab*. We need to specify it, because the *default* separator is a *comma*.

# *preparing the training set and test set*

training\_set = **pd.read\_csv**("./movie\_lens\_100k/u1.base", delimiter="\t")

* The *delimiter tab* should rather be taken with this ***delimiter*** ***argument*** rather than previously used ***sep*** ***argument***.
* Split-size: What is the split? I mean, what is the *proportion* of the *training set* compared to the *whole set*?
* Remember, the ***original*** dataset contains ***100,000*** ratings. And since each observation corresponds to one rating, Here in ***training-set*** we have ***80,000*** observations.

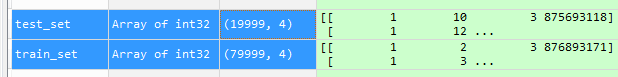
|  |  |
| --- | --- |
| * That means that we have ***80,000*** ratings. And therefore, the training set is ***80%*** of the original dataset composed of the ***100,000*** ratings. So that will be an ***80%, 20% train-test split***. That's the optimal train-test split to train a model. * The first column (not including Index) corresponds to the users, the ***second column*** corresponds to the ***movies*** and the ***third column*** corresponds to the ***ratings*** and then the ***fourth column*** corresponds to the ***timestamps***. * Take the ***index-4 row***, it is the ***5th*** observation, the ***user*** number ***1*** rated the ***movie*** number ***7***, and gave it ***4 stars***. |  |

* NymPy Array: Now we have to convert it into an array because ***Pytouch.tensors*** expects the data as array.
* Also we *convert* all number into *Integers*, to do that we used ***dtype="int"***

train\_set = **np.array**(training\_set, dtype="int")

|  |  |
| --- | --- |
| * test\_set: Now we're gonna do the same for the test set, * Notice that the ***training set*** and the ***test set*** have *different ratings*, you know, there is no common rating of a same movie by the same user between the ***training set*** and the ***test set***. |  |

* However we have the same users, we start with ***user 1*** as in the ***training-set*** and ***test-set***. But for this same ***user 1*** we won't have the *same movies* because the *ratings* are *different*.



# *preparing the training set and test set*

training\_set = **pd.read\_csv**("./movie\_lens\_100k/u1.base", delimiter="\t")

train\_set = **np.array**(training\_set, dtype="int")

ts\_set = **pd.read\_csv**("./movie\_lens\_100k/u1.test", delimiter="\t")

test\_set = **np.array**(ts\_set, dtype="int")

|  |
| --- |
| * Next, we will get the *maximum number* of *users* and the *maximum number* of *movies* in two separate variables because then we will need these two variables to prepare our RBM. |

**14.4.4 Data Preprocessing 2 : max User & max Movies**

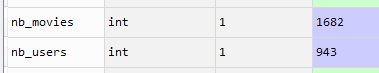
Now we're gonna get the total number of users and movies. Then we are going to ***convert*** our ***training set*** and ***test set***, into a ***matrix*** where the lines/rows are going to be ***users***, the ***columns*** are going to be ***movies***, and the ***cells*** are going to be the ***ratings***.

* We are going to create such a *matrix* for the *training set*, and another one for the *test set*. And, in *each* of these two *matrices*, we want to include all the *users* and all the *movies* from the *original data set*.
* In the training-set, if a user *didn't rate* a movie, well we'll put a 0, into the corresponding cell of the matrix.
* These *matrices* will have the *same* *number* of *users* and the *same number* of *movies*, so they will have the *same number* of *rows* and the *same number* of *columns*.
* In these two matrices, each cell of indexed **U**, **I**, where **U** is the user and **I** is the movie. Each cell **U**, **I** will get the rating of the movie **I**, by the user **U**. And if, this user **U** didn't rate the movie **I**, we'll put a **0**.
* We're gonna make a function to do this.
* Since these *two matrices* will contain the *total number* of *users* and the total *number of movies*, we need to *find* those numbers.
* Finding max-user and max-movies: We could *scroll* our *dataset* and find out *those numbers*, and manually implement those numbers. But to make our code more flexible, we use ***max()*** function, it make us enable to use *any* *train-test set* and finds the corresponding ***max-user*** and ***max-movies*** automatically.
* Manually using *maximum number* can cause *problem* because *different train-test split* may have different amount of *users* and *movies*.
* So we use the code that finds the maximum from both ***train set*** and ***test set***.

# *Getting the number of Users and Movies*

nb\_users = **int**(max(max(train\_set[:, 0]), max(test\_set[:, 0])))

nb\_movies = **int**(max(max(train\_set[:, 1]), max(test\_set[:, 1])))



|  |
| --- |
| * Notice: * **max(train\_set[:, 0]** finds the *maximum* from the first column (user-Id) of *training-set*. **max(train\_set[:, 0]** similarly finds the *maximum* from the *test-set*. Then we pick the *maximum* of these *2 maximum numbers*. Then we convert those to integers. * For movies, **int**(**max**(**max**(train\_set[:, 1]), **max**(test\_set[:, 1]))) finds the *maximum of maximums* from 2nd columns (Movie No.) from ***train-set*** and ***test-set***. * Also notice how we changed the *index no.* **0** & **1** to select the corresponding columns for ***Users ID*** and ***Movie No***. |

* *Next* we *convert* our *training set* and *test set*, into two *matrices* where the ***lines*** are the ***users***, the ***columns*** are the ***movies***, and the *timestamps* will be *removed*.

**14.4.5 Data Preprocessing 3 : Creating Matrices**

Now we are going to convert our training set and test set into an *array* with ***users*** in *lines* and ***movies*** in *columns*.

* Because we need to make a *specific structure* of data that *RBM expects* as *inputs*. We will have the observations in lines and the features in columns.
* We're just making the *usual structure* of data for *neural networks* or even for *machine learning* in general that is with the *observations* in *lines* and the *features* in *columns*.
* Since we're gonna do this for both the *training set* and the *test set*, we're gonna create a *function* that we will apply on both *training set* and the *test set*.
* We're gonna call this function ***convert***, we need to give an arguments to the function, which will be our dataset.
* We could use 2-dimensional ***NumPy array***. But we're gonna use ***Torch*** afterwards, we ***won't create a two-dimensional NumPy array***, we will create a list of lists.
* Since we have ***943*** users, we'll have ***943 lists***, these will be ***horizontal*** ***lists***, each corresponds one ***user***. Each user is a ***list*** of ***observations*** in ***lines***. i.e the first list will correspond to the first user, the second list will correspond to the second user.
* Each ***list*** contains the ratings of the ***1,682 movies*** by the ***user*** corresponding to the ***list***.
* If the *user didn't rate* the movie, then we'll get a **0** for that.
* That's why the new *converted training set* and *test set* will have the same size because basically for both the training set and the test set, we are considering all the users and all the movies, and we just put a ***0*** when the user ***didn't rate*** the movie.
* So, this whole ***list of lists*** will be a list of ***943*** ***lists*** because we have ***943*** ***users***, and each of these ***943*** ***lists*** will be a ***list*** of ***1,682 elements*** because we have ***1,682 movies***.

# *converting the data into an array with users in lines and movies in column.*

**def** **conVert**(data):

    new\_data = []

**for** id\_user **in** **range**(1, nb\_users + 1):

        # *use "data[:, 0] == id\_user" as condition over movie column "data[:, 1]"*

        id\_movies = data[:, 1][data[:, 0]**==** id\_user]    # *returns a list*

        # *use "data[:, 0] == id\_user" as condition over ratins column "data[:, 2]"*

        id\_ratings = data[:, 2][data[:, 0]**==** id\_user]

        # *vector of zeros*

        ratings = **np.zeros**(nb\_movies)

        ratings[id\_movies - 1] = id\_ratings

**new\_data.append**(**list**(ratings))

**return** new\_data

trn\_set\_cnvt = **conVert**(train\_set)

tst\_set\_cnvt = **conVert**(test\_set)

* Notice how we get the list of movies that a user rated:

id\_movies = data[:, 1][data[:, 0]**==** id\_user]    # *returns a list*

* Here, ***data[:, 1]*** selects the *movie column* (2nd column), then we apply the condition: **"data[:, 0]== id\_user"**, it gets the list of movies that are rated by the **id\_user**.
* Similarly we get the corresponding ratings, that are given by **id\_user**

id\_ratings = data[:, 2][data[:, 0]**==** id\_user]

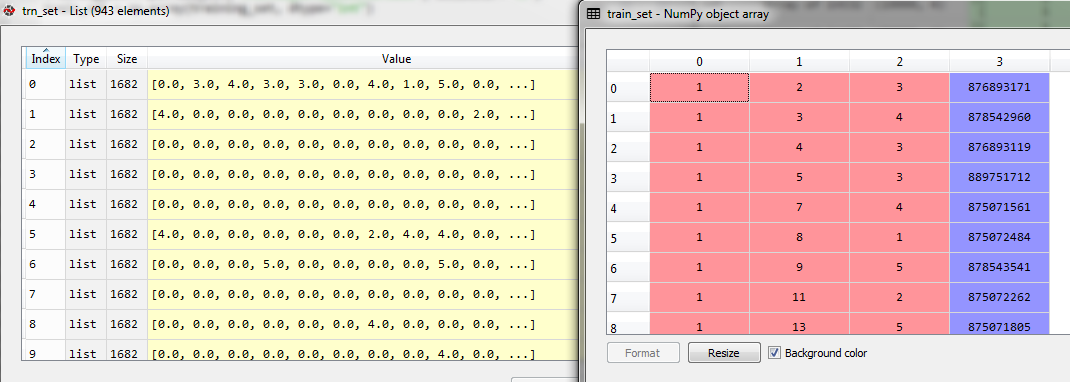
* Here, ***data[:, 2]*** selects the *ratings column* (3rd column), then we apply the condition: **"data[:, 0]== id\_user"**, it gets the list of movies that are rated by the **id\_user**.
* **data[:, 0]== id\_user** means from the user\_id column (first column) select the item matches the **id\_user**.
* Following creates a list of zeros named ***ratings*** of size ***nb\_movies***.

ratings = **np.zeros**(nb\_movies)

In the ***ratings*** list we put a movie of id "**n**" at the index "**n-1**", since the ids of movies starts from **1**.

* List operation: Also notice the following list operation, it puts the ratings ***id\_ratings*** given by ***id\_user*** to the corresponding movie ***id\_movies***. Indexes in Python start at zero, and our movies' IDs start at one.
* Note that both ***id\_movies*** and ***id\_ratings*** are lists, so following uses each element of ***id\_movies*** as an *index* and from ***id\_ratings*** uses each element as corresponding ratings.

ratings[id\_movies - 1] = id\_ratings



* Notice each line has different ratings now. Since user ***1 didn't rated*** movies **1, 6, 10, 12** and many-others then there are ***0.0*** for these.
* Also we used ***list()*** method, to make sure that ***ratings*** will be a *list*:

**new\_data.append**(**list**(ratings))

* Note in following, range is because there is ***no user*** **0** in our dataset, and for loop excludes *"****nb\_users + 1****"*, so our users will be ***1*** to ***nb\_users***.

**for** id\_user **in** **range**(1, nb\_users + 1):

**14.4.6 Data Preprocessing 4 : Convert Matrices to torch.Tensor**

Now we've converted our ***training set*** and our ***test set*** into the ***arrays*** composed of the *users* in *lines* and the *movies* in *columns*.

* The *columns* are the *features* that are going to be the *input nodes* in the *network*.
* For each ***user*** we will have its ***ratings of all the movies***, ***zeros*** included, and these ratings are going to be the *input nodes* for this observation going into the *network*.
* Then ***PyTorch*** comes into play, because we will build the architecture with ***PyTorch tensors***.
* A tensor is a multi-dimensional matrix but instead of being a ***NumPy array***, this is a ***PyTorch array***.
* In fact, we could build a ***NN*** with ***NumPy arrays***, but that would be much less efficient and that's why we're using ***tensors*** using the ***torch.Tensors***.
* Note: With ***TensorFlow*** we have exactly the same. With ***TensorFlow*** we work with tensors.
* Those are another ***kind*** of ***tensor***, another kind of ***multi-dimensional matrix***, and so we could also implement our ***AE/RBM*** from scratch with ***TensorFlow***.
* But for ***AE/RBM***, ***PyTorch*** gives *better results*, and also this is much more simple.

# *Converting the data into Torch Tensosrs*

train\_set\_tensor = **torch.FloatTensor**(trn\_set\_cnvt)

test\_set\_tensor = **torch.FloatTensor**(tst\_set\_cnvt)

* Now ***training set*** and the ***test set*** are ***torch.Tensor***, two, separate *multi-dimensional matrices* based on ***PyTorch***.
* We used the class ***FloatTensor***, that creates two object of this class: ***train\_set\_tensor*** and ***test\_set\_tensor***. These objects will be the ***torch.Tensor*** itself.
* A ***torch.Tensor*** is a multi-dimensional matrix with a *single data-type*. Since we're taking the ***FloatTensor*** class, the data-type will be ***float***.
* Inside each of those classes, we used *one argument* which has to be a ***list of lists***, our ***train-set*** and ***test-set*** (list-of-lists).
* The ***FloatTensor*** class expects a list of lists.
* This will give the exact, same matrix with the ***users*** in ***lines*** and the ***movies*** in ***columns***, but instead of being a ***NumPy array***, this will be a ***torch.Tensor***.
* Now, I have to warn you, the training set and the test set in variable explorer may *disappear*, because the variable explorer pane in *Spyder* may *not recognize* *torch.Tensors* yet.

|  |  |
| --- | --- |
|  |  |

* Now the ***common*** ***data pre-processing*** for recommended system is done and now it's time to take care of what is specific to ***RBM***.
* Remember, with ***RBM***, we're gonna ***predict*** if a user likes ***yes*** or ***no*** a movie (binary prediction). So we have to convert all the ratings into ***binary ratings***, ***0*** or ***1***. Because these are gonna be the inputs of our RBM.

**All data-preprocessing at once**

# *----------- RBM : Recommender ---------------*

# *Importing the libraries*

**import** pandas **as** pd

**import** numpy **as** np

**import** torch

**import** torch.nn **as** nn

**import** torch.nn.parallel

**import** torch.optim **as** optim

**import** torch.utils.data

**from** torch.autograd **import** Variable

# *importing the dataset*

movies = **pd.read\_csv**("./movie\_lens\_1m/movies.dat", sep= "::", header=**None**, engine="python", encoding="latin-1")

useRs = **pd.read\_csv**("./movie\_lens\_1m/users.dat", sep= "::", header=**None**, engine="python", encoding="latin-1")

RaTings = **pd.read\_csv**("./movie\_lens\_1m/ratings.dat", sep= "::", header=**None**, engine="python", encoding="latin-1")

# *preparing the training set and test set*

training\_set = **pd.read\_csv**("./movie\_lens\_100k/u1.base", delimiter="\t")

train\_set = **np.array**(training\_set, dtype="int")

ts\_set = **pd.read\_csv**("./movie\_lens\_100k/u1.test", delimiter="\t")

test\_set = **np.array**(ts\_set, dtype="int")

# *Getting the number of Users and Movies*

nb\_users = **int**(max(max(train\_set[:, 0]), max(test\_set[:, 0])))

nb\_movies = **int**(max(max(train\_set[:, 1]), max(test\_set[:, 1])))

# *converting the data into an array with users in lines and movies in column.*

**def** **conVert**(data):

    new\_data = []

**for** id\_user **in** **range**(1, nb\_users + 1):

        # *use "data[:, 0] == id\_user" as condition over movie column "data[:, 1]"*

        id\_movies = data[:, 1][data[:, 0]**==** id\_user]    # *returns a list*

        # *use "data[:, 0] == id\_user" as condition over ratins column "data[:, 2]"*

        id\_ratings = data[:, 2][data[:, 0]**==** id\_user]

        # *vector of zeros*

        ratings = **np.zeros**(nb\_movies)

        ratings[id\_movies - 1] = id\_ratings

**new\_data.append**(**list**(ratings))

**return** new\_data

trn\_set\_cnvt = **conVert**(train\_set)

tst\_set\_cnvt = **conVert**(test\_set)

# *Converting the data into Torch Tensosrs*

train\_set\_tensor = **torch.FloatTensor**(trn\_set\_cnvt)

test\_set\_tensor = **torch.FloatTensor**(tst\_set\_cnvt)

**14.4.7 Prepare Input Data : Convert rating-tensors to *binary-rating-tensor***

We have to convert these ratings for our RBM. Right now, we have ratings from ***1*** to ***5***,in our ***training set*** and ***test set***.

* Now we have to convert these *ratings* into *binary ratings*. Because our *RBM-recommender* *system* will generate *binary-ratings*.
* ***1 = liked***, or ***0 = not liked***.

|  |
| --- |
| * Why do we have to convert these ratings? * Because we want to predict some binary ratings. * We also *need* the *inputs* to have the *binary* *format* **0** or **1**, because, the RBM will take the *input* *vector*, from this vector, RBM will predict the ratings for the *movies* that were not originally rated by the *user*. * Since these *predicted ratings* are computed, originally, from the *existing ratings* of the *input vector*, well, then the *predicted ratings* in the output must have the *same format* as the existing ratings in the *input*. * Otherwise, things would be inconsistent for the RBM. |

* Let's convert all these ratings into ***binary ratings***, **1** or **0** for both the ***training set*** and the ***test set***.
* We're gonna replace all the **0** in this original training set, by **-1**. Because all the ***zeroes*** in the original training set, corresponded to the movies that were not rated by the users.
* Now, minus one (**-1**) will mean that there was *not a rating* for a *specific movie*, given by a *specific user*.

train\_set\_tensor[train\_set\_tensor **==** 0] = -1

* And in **[]** brackets, we simply added the *condition* that we want to get these *ratings*. This is a *torch-tensor-operation*, it finds all **0**'s in the elements of the tensor and replace these with "**-1**".
* We're gonna do the same for the other ratings, that is, the ratings from **1** to **5**.
* The ratings that we want to ***convert*** into ***zero***, that is, not liked. Are the movies that were given ***one star*** or ***two stars***.
* Unfortunately, the "**or**" doesn't work with ***torch*** objects. There is no option in ***torch.tensor*** object that we can apply compound condition as:

(test\_set\_tensor **==** 1) or (test\_set\_tensor **==** 2)

* The **or** operator doesn't work like that for PyTorch. That’s why we have to do these rating saperately:

train\_set\_tensor[train\_set\_tensor **==** 1] = 0

train\_set\_tensor[train\_set\_tensor **==** 2] = 0

* this *torch-tensor-operations*, finds all **1,2**'s in the elements of the tensor and replace these with "**0**".
* For the movies that the users liked, we make them 1 for those movies that are rated ***3***,***4*** or ***5***.

train\_set\_tensor[train\_set\_tensor **>=** 3] = 1

* In **[]** brackets, we simply added the *condition* that "**>=** 3", this *torch-tensor-operation*, finds all **3,4, 5**'s in the elements of the tensor and replace these with "**1**".
* The movies that were rated at least three stars were rather liked by the users. So, three stars, four stars, five stars are gonna become one.
* Now we do the same for test-set

test\_set\_tensor[test\_set\_tensor **==** 0] = -1

test\_set\_tensor[test\_set\_tensor **==** 1] = 0

test\_set\_tensor[test\_set\_tensor **==** 2] = 0

test\_set\_tensor[test\_set\_tensor **>=** 3] = 1

* Now all the ratings from one to five will be converted into binary ratings in both the training set and the test set. So, we're getting our inputs ready to go into the RBM, and then RBM will returns the ratings (predicts) of the movies that were not originally rated in the input vector.

# *Converting the ratings into binary ratings: 1 (liked),  0 (not-liked)*

train\_set\_tensor[train\_set\_tensor **==** 0] = -1

        # *torch doesn't support combined condition*

train\_set\_tensor[train\_set\_tensor **==** 1] = 0

train\_set\_tensor[train\_set\_tensor **==** 2] = 0

train\_set\_tensor[train\_set\_tensor **>=** 3] = 1

test\_set\_tensor[test\_set\_tensor **==** 0] = -1

test\_set\_tensor[test\_set\_tensor **==** 1] = 0

test\_set\_tensor[test\_set\_tensor **==** 2] = 0

test\_set\_tensor[test\_set\_tensor **>=** 3] = 1

* That's Unsupervised Deep Learning, and that's exactly how it works.

**To view our tensors as a NumPy array**

* Converting a ***torch.tensor*** object into a **NumPy** array object: You need to call **.detach()** before saving your data e.g. **x.detach().numpy()** if your *tensors* have *grads*...also you might need to call **cpu()**. I think this should work: **x.detach().cpu().numpy()**
* Here ***x*** is the ***torch.tensor*** object

train\_tensor\_to\_view = **train\_set\_tensor.detach**().**cpu**().**numpy**()

test\_tensor\_to\_view = **test\_set\_tensor.detach**().**cpu**().**numpy**()

* However we can simply use: **train\_set\_tensor.detach**().**numpy**() or **train\_set\_tensor.numpy**(). For our simple dataset, we can use following:

train\_tensor\_to\_view = **train\_set\_tensor.numpy**()

test\_tensor\_to\_view = **test\_set\_tensor.numpy**()



|  |
| --- |
| Some real life examples converting tensors to numpy array:  **Example: Shared storage**  PyTorch tensor residing on CPU shares the same storage as numpy array na  import torch  a = torch.ones((1,2))  print(a)  na = a.numpy()  na[0][0]=10  print(na)  print(a)  Output:  tensor([[1., 1.]])  [[10. 1.]]  tensor([[10., 1.]])  **Example: Eliminate effect of shared storage, copy numpy array first**  To avoid the effect of shared storage we need to copy() the numpy array na to a new numpy array nac. Numpy copy() method creates the new separate storage.  import torch  a = torch.ones((1,2))  print(a)  na = a.numpy()  nac = na.copy()  nac[0][0]=10  ​print(nac)  print(na)  print(a)  Output:  tensor([[1., 1.]])  [[10. 1.]]  [[1. 1.]]  tensor([[1., 1.]])  Now, just the nac numpy array will be altered with the line nac[0][0]=10, na and a will remain as is.  **Example: CPU tensor with requires\_grad=True**  import torch  a = torch.ones((1,2), requires\_grad=True)  print(a)  na = a.detach().numpy()  na[0][0]=10  print(na)  print(a)  Output:  tensor([[1., 1.]], requires\_grad=True)  [[10. 1.]]  tensor([[10., 1.]], requires\_grad=True)  In here we call:  na = a.numpy()  This would cause: *RuntimeError: Can't call numpy() on Tensor that requires grad. Use tensor.detach().numpy() instead.,* because tensors that **require\_grad=True** are recorded by PyTorch AD. Note that **tensor.detach()** is the new way for **tensor.data**.  This explains why we need to detach() them first before converting using numpy().  **Example: CUDA tensor with requires\_grad=False**  a = torch.ones((1,2), device='cuda')  print(a)  na = a.to('cpu').numpy()  na[0][0]=10  print(na)  print(a)  Output:  tensor([[1., 1.]], device='cuda:0')  [[10. 1.]]  tensor([[1., 1.]], device='cuda:0')  ​  **Example: CUDA tensor with requires\_grad=True**  a = torch.ones((1,2), device='cuda', requires\_grad=True)  print(a)  na = a.detach().to('cpu').numpy()  na[0][0]=10  ​print(na)  print(a)  Output:  tensor([[1., 1.]], device='cuda:0', requires\_grad=True)  [[10. 1.]]  tensor([[1., 1.]], device='cuda:0', requires\_grad=True)  Without detach() method the error RuntimeError: Can't call numpy() on Tensor that requires grad. Use tensor.detach().numpy() instead. will be set.  Without .to('cpu') method TypeError: can't convert cuda:0 device type tensor to numpy. Use Tensor.cpu() to copy the tensor to host memory first. will be set.  You could use cpu() but instead of to('cpu') but I prefer the newer to('cpu'). |

**14.4.8 RBM-architecture : RBM class - \_\_init\_\_()**

* Now we build the architecture of our NN i.e, the architecture of the RBM.
* We're gonna make a class, which will define the *architecture of the RBM*.
* And then, we simply, create an object of this class, that object will be the RBM model.
* Here we will choose the *number* of *hidden nodes*, and mostly, we will build the newel network just like how it works, that is, we're gonna make this Probabilistic Graphical Model. Because, let's remember, a *RBM* is a *probabilistic graphical model*.
* The class that we build for RBM, will contain following. These are first parameters that we need to initialize the RBM, using "**\_\_init\_\_**".
* The *number* of hidden *nodes*
* The *weights*.
* The *weights* for the *probability* of the *visible node*, given the *hidden node*.
* The *bias* for the same *probability*
* The *bias* for the *probability* of the *visible node,* given the *hidden node*.
* We also add some functions, for example a self-driving car will need a lot of functions. Some *functions* to recognize *objects* on the *street*. Then some *functions* to *turn right*, to *turn left*, to move *forward*, to move *backward*, or to *stop* when there is an *obstacle* on the street.
* We're gonna make 3 functions.

1. One function to *initialize* the *RBM object* that will be created afterwords.
2. Second function will be ***sample\_H***, that will *sample* the *probabilities* of the *hidden nodes* given the *visible nodes*. .
3. Third function will be ***sample\_V***, that will sample the *probabilities* of the *visible nodes* given the *hidden nodes*. .
4. 4th function will be ***train***, that will apply the Contrastive divergence.

* **\_\_init\_\_:**  **\_\_init\_\_**  function is to define the parameters of the object that will be created once the class is made. There are 3 arguments:
* "self": ***self*** corresponds to the ***object*** that will be created afterwords. All the variables that are attached to the object will be created by putting a ***self*** before the variable.
* **nv**: The second argument is **nv**, and that's the *number* of *visible nodes*.
* **nh**: The third argument is **nh**, the *number* of *hidden nodes*.
* Then we need to initialize the ***weight*** and the ***bias***.
* Inside this **\_\_init\_\_**  function there going to be all the ***parameters*** that we will ***optimize*** during the training of the ***RBM***.

1. the ***weights*** and
2. the ***bias***.

* Let's start with the weights, we're gonna call them W.
* Why capital W? Because ***all*** the ***weights*** are going to be ***initialized*** in a ***torch.tensor***.
* These weights are all the ***parameters*** of the ***probabilities*** of the ***visible*** *nodes* given the ***hidden*** *nodes*. .
* According to the theory, they are ***initialized*** in a ***matrix*** of size ***nh***, number of hidden *nodes*, and ***nv***, the number of visible *nodes*. Since we're working with PyTorch, well, this matrix is going to be a ***torch.tensor***. A tensor of one single type.

self.W = **torch.randn**(nh, nv)

* Since these weights have to be *initialized* *randomly* according to a *normal distribution*, we need to use this ***rendn()*** of ***PyTorch***. It will initialize all the weights in our tensor.
* This **torch.randn**(nh, nv) initializes a tensor of size . According to a *normal distribution*. This *normal distribution* has a ***mean*** of ***0*** and a ***variance*** of ***1***.

That initializes all the weights for the probabilities of the ***visible*** *nodes* given the ***hidden*** *nodes*.

* Let's initialize the ***bias***.
* There is some ***bias*** for the *probability* of the *hidden node* given the *visible node* , and
* some ***bias*** for the *probability* of the *visible node* given the *hidden node* .
* ***bias*** for the *probability* of the *hidden node* given the *visible node* : We're gonna give the name "**a**".

self.a = **torch.randn**(1, nh)

* We're gonna use our ***rendn*** function again to *initialize* the *weights* according to ***normal distribution*** of ***mean*** ***0*** and ***variance*** ***1***. This will be a Tensor of size (i.e. a *vector* of size of the *hidden nodes*).
* Since there is *one bias* for *each* *hidden* *node* and we have ***nh*** hidden node, that’s why we created a vector of ***nh*** element. And initialized to some numbers randomly that follow a normal distribution.
* But we need to create an *additional dimension* *corresponding* to the *batch*, and therefore, this *vector* shouldn't have *1 dimension*, like a single input vector, it should have *2 dimensions*.
* The ***first dimension*** corresponding to the ***batch*** and the ***second dimension*** corresponding to the ***bias***.
* It's because the functions that we're gonna use of ***PyTorch*** ***cannot accept a single input vector*** of one dimension as argument, but a ***two dimensional*** ***tensor*** with the ***first dimension*** corresponding to the ***batch*** and the ***second dimension*** corresponding to the ***bias***.
* So that's why here, we ***cannot*** put directly ***nh***.
* **torch.randn**(1, nh) creates a 2-D tensor with this ***1*** here corresponding to the first dimension that is the ***batch***. And this ***nh*** element here corresponding to the ***bias***.
* The bias for *Probability* of the *visible node* given the *hidden node,* ***p\_v\_given\_h***,***:*** It is same as above, we give it a name "**b**"
* This time we use **nv** (instead of **nh**), number of the visible node, while we initialize the tensor we make it a 2D-tensor as we did above.

self.b = **torch.randn**(1, nv)

Those will initialize our *future* *objects* of the *RBM class*.

# *Creating the architecture of the Neural Netwark*

**class** RBM():

**def** **\_\_init\_\_**(self, nv, nh) :

        self.W = **torch.randn**(nh, nv)

        self.a = **torch.randn**(1, nh)

        self.b = **torch.randn**(1, nv)

**14.4.9 RBM-architecture : RBM class – sample\_h()**

Now we make the function that will *sample* the *hidden node*. The second function is about *sampling* the *hidden nodes* according to the probabilities, where ***h*** is a ***hidden*** node and ***v*** is a ***visible*** node

* From the intuition section we saw, this probability is nothing else than the ***sigmoid*** *activation function*.
* Why do we need this ***sample\_h*** function? Because during the training we will *approximate* the *log likelihood gradient* and we will do that through *Gibbs sampling*.
* To apply *Gibbs sampling*, we need to compute **,** the *probabilities* of the *hidden nodes* given the *visible nodes*. Once we have this probability, we can sample the *activations* of the *hidden nodes*.
* We're gonna call this function ***sample\_h*** because it will return some *samples* of the different *hidden nodes* of our *RBM*.
* Suppose we have ***100*** *hidden**nodes* in our RBM. Well, this function will ***sample*** the *activations* of these *hidden**nodes*, that is for ***each*** of these ***100*** *hidden**nodes*, it will ***activate*** them according to a *certain**probability* that we will compute in this same function. And for each of this hidden node, this probability is .
* That, is the ***probability*** that this ***hidden node = 1*** ***given v*** that is given the value of **v**,. And that is this *probability* that is *equal* to the *activation* *function*.
* This ***sample\_h*** function takes two arguments, ***self*** and ***x***. ***x*** will correspond to the visible neurons, ***v***, in the probabilities, p h given v, i.e..
* First we have to compute . That is the probability that the *hidden* neuron *equals one* given the values of the *visible neurons*. That is actually our input vector of observations with all the ratings.
* It is nothing else than the Sigmoid Activation Function applied to .
* **wx** is the product of the *vector of weights* "" with the *vector of visible neurons*"".
* The bias, **a** responds to the *bias* of the *hidden nodes*.

[bias **b** corresponds to the *bias* of the *visible nodes*. We'll apply it to the sample\_V function, for the visible nodes]

* Product wx: To compute the product of the ***weights*** times the ***neurons***, that is. To make the product of two ***tensors*** in ***Pytorch*** we need to use a function called .

wx = **torch.mm**(x, self**.W.t**())

* To make things *mathematically correct*, we actually need to take the *transpose* of weights .
* Activation-value: Now let's compute what is going to be inside the *sigmoid activation function*.
* The value for the activation function is: **(wx + bias)**.
* It is same as for every Deep-Learning model, inside the *activation function* is a *linear function* of the ***neurons*** where the ***coefficients*** are the ***weights***.
* Here we have the ***bias = a***. So the value for activation is **(wx + a)**. We'll name this variable ***activation***.

activation = wx + self**.a.expand\_as**(wx)

* We need to do something more. Remember that *each input vector* will *not be treated individually*, but inside batches. i.e. the new dimension that we created by using ***1's*** in **randn**(1, nh)and **randn**(1, nv).
* Even if the batch contains one ***input vector*** or you know, one ***vector of bias***, well that input vector is still in the batch. And in that case we call it a ***mini batch***.
* So when we add the bias *(bias of the hidden nodes =* ***a****)*, well we want to make sure that this ***bias*** is *applied* to *each line of the mini batch*.
* To make sure of that, we need to use a *function* called **expand\_as()** that will again, *add a new dimension* for these ***bias*** (bias **a** & **b**) that we're adding.
* In *parenthesis*, we need to specify how to expand the ***bias "a"*** by giving the *tensor* that we want to *add with*. Since we are adding ***a*** with ***wx*** we used **expand\_as**(wx).
* It is about making "a" as the *similar format* of "wx" before adding. That makes sure that the *bias* are *applied* to each line of the *mini batch*.
* It is this linear combinations of the the *visible neurons* ***x***, with coefficient ***W*** (*weights*) and added the ***bias***.
* Sigmoid-Activation-Function: Now we can *compute* the *activation function* that will *activate* the *hidden node*. But remember this *activation* function *represents* a *probability*. It will be the *probability* that the *hidden node* will be *activated* according to the *value* of the *visible node*.
* We give it a name ***p\_h\_given\_v*** i.e. .
* For example, let's say that we have a *user* that *likes* only *drama-movies*.
* Well, if there is a *hidden node* that *detected* a specific *feature* corresponding to that *drama genre* of the movies, those drama-movies that are rated **1** by the user.
* Then the ***probability*** of that ***node*** specific to this *drama (feature) genre*, given the *visible node* of that *user* who has *all the nodes* of the *drama movies* is equal to ***one***.
* This probability: **p\_h\_given\_v** will be very *high*, because **v = 1** *(user rated 1)* for the *drama movies* and **h** corresponds to the *drama movie genre*. So i.ewill be very high.

p\_h\_given\_v = **torch.sigmoid**(activation)

To use the *sigmoid activation function* we used ***sigmoid()*** method of ***PyTorch***. And use the value (linear combination of the neurons) "**activation**" to make it sigmoid.

* Returning "probability" and "sample of h": The final step is to return not only the probability ***p\_h\_given\_v***, but of course a *sample of h*.
* *Sample of h* is the *sample of all the hidden nodes/neurons*, according to this probability ***p\_h\_given\_v***.
* Its important to understand is that we're making a Bernoulli RBM, because we're just predicting a *binary outcome* (users like yes or no, a movie).
* Hence we'll return ***Bernoulli samples*** of that ***distribution***. Of that probabilities ***p\_h\_given\_v***.
* What does that mean?
* Right now ***p\_h\_given\_v*** is a vector of **nh** elements.
* For example, suppose we have *100 hidden nodes*, while this ***p\_h\_given\_v*** vector is a vector of *100 elements*. Each of these *100 elements* corresponds to each of the *100 hidden nodes* and each of these ***elements*** is the ***probability*** that the ***hidden*** node is ***activated***.
* Let's take the **i'th** element of this *vector*, it is the probability that the **i'th** *hidden node* is *activated*. But remember that's *given* the *values* of the *visible nodes*. i.e given the ***ratings*** of the ***user*** we're dealing with.
* So that's what we have in this ***p\_h\_given\_v*** vector. And the idea is to use these ***probabilities*** to ***sample*** the ***activation*** of each of this *100 hidden nodes*.
* That is for each of these *100 hidden nodes*, while depending on that probability ***p\_h\_given\_v*** that we have for these hidden nodes, we will activate ***yes*** or ***no***, this hidden neuron.
* And so how are we going to activate it?

Let's suppose that for a ***hidden neuron i***, the probability corresponding to that hidden neuron in this ***p\_h\_given\_v*** vector is ***0.7***, ***70%***.

In that case we take a ***random number*** between ***zero*** and ***one***, If this *random number* is ***below*** ***0.7***, ***70%*** then we will ***activate*** the neuron. And if this *random number* is ***larger*** than ***0.7*** then we will ***not activate*** the neuron.

* That's how Bernoulli sampling works. And so we do that for *each* of the *hidden nodes* of our **100** *nodes*. And then in the end we get a ***vector*** of ***zeros*** and ***ones***.
  + - * The ***zeros*** correspond to the ***hidden nodes*** that were ***not activated*** *after the sampling*,
      * ***Ones*** correspond to the ***neurons*** that were ***activated*** by the sampling.
* How do we return that sampling of the hidden neurons?
* We have a torch function called **Bernoulli()**. In this function we have to input the distribution of which we are making that sampling, And i.e. ***p\_h\_given\_v***.
* So that will return all the ***probabilities*** of the ***hidden neurons***, given the values of the ***visible nodes*** (i.e. the ratings ***0*** or ***1***). and it will return also that *sampling* of the *hidden neurons*.

**def** **sample\_h**(self, x):

        wx = **torch.mm**(x, self**.W.t**())

        activation = wx + self**.a.expand\_as**(wx)

        p\_h\_given\_v = **torch.sigmoid**(activation)

**return** p\_h\_given\_v, **torch.bernoulli**(p\_h\_given\_v)

That's the first function we need for Gibbs sampling. But then we need another function, which is ***sample\_v*** and basically we will do the same, but this time for the *visible neurons*.

**14.4.9 RBM-architecture : RBM class – sample\_h()**

We just implemented our ***sample\_h*** function to *sample* the *hidden node* according to the probabilities ***p\_h\_given\_v***.

* We're gonna do the same for the *visible node*, from the values in the hidden nodes, that is whether they were activated or not.
* We will also estimate the *probabilities* of the *visible nodes*, that is the *probabilities* that ***each*** of the ***visible node equals one***.
* Our goal in the end is to *output* the *predicted ratings*, **1** or **0**, of the movies that were *not originally rated* by the *user*, and these *new* *ratings* that we get in the end are taken from what we *obtained* in the *hidden node*, that is from the *samples* of the *hidden node*.
* also, it's not the only reason to make sample\_v, we also need it for Gibbs sampling when we approximate the ***log likelihood gradients***.

**def** **sample\_v**(self, y):

        wy = **torch.mm**(y, self.W)

        activation = wy + self**.b.expand\_as**(wy)

        p\_v\_given\_h = **torch.sigmoid**(activation)

**return** p\_v\_given\_h, **torch.bernoulli**(p\_v\_given\_h)

* We just need to copy this ***sample\_h()*** function we will replace just one or two things. Our goal is:

1. To return some *samples* of the *visible node*
2. We also return the *probability* of ***v\_given\_h***,

* We're gonna call our function ***sample\_v*** that make some samples of the ***visible*** ***node*** according to the probabilities **p\_v\_given\_h** .
* **p\_v\_given\_h** means, given the values of the *hidden* *nodes* (i.e. whether the hidden nodes are activated or not)we return the *probabilities* that each of the *visible* *nodes* equals ***one***.
* As before we return some ***samples*** of the ***visible*** ***node*** based on that ***Bernoulli sampling***. It is *vector* of *probabilities* of the *visible nodes*, since we have ***1,682 movies***, then we have ***1,682 visible nodes***, a vector of 1,682 probabilities, one probability for each of the visible nodes,
* So *each* of these *probabilities* is the probability that the corresponding *visible node* is equal to *one* but that remember is *given* the *activations* of the *hidden nodes* (activated or not).
* From the vector of probabilities ***p\_v\_given\_h*** we return some ***sampling*** of the *visible nodes*.
* Let's say the **i**-th visible node has a probability of ***0.25***, then we take a random number between ***zero*** and ***one***.
* If this number is below 0.25, then this *visible node* will get the value *one*. So that means that we *predict* that the movie *corresponding* to that *visible node* will get a *like* by the *user*,
* If this random number is larger than ***0.25***, then this visible node will get the value ***zero***. And, therefore, we predict that the *movie* corresponding to that visible node will *not get a like* by the *user*.
* We need to change what's inside the ***activation function***,
* first, we will replace variable ***x*** by ***y***.

**def** **sample\_v**(self, y):

* Previously, **x** represented the ***visible node*** because we apply the ***sample\_h()*** function to the *visible node* because we want to *return* the *probabilities* that the ***hidden nodes = 1*** according to the value of the ***visible nodes***.
* But ***sample\_v()*** function will return the *probabilities* of the *visible nodes* given the values of the *hidden nodes*.
* The variable ***y*** is this time the *values* of the *hidden nodes*.
* We take the *torch product* of *matrices*, of tensors, with ***y*** and the weight ***w***, the *product* of the *hidden nodes* with *weight* w, for the probabilities ***p\_v\_given\_h***.
* No transpose: This time since we're making the product of the ***hidden nodes*** and the torch tensor of ***weight*** w, for the probabilities ***p\_v\_given\_h***, well here we must not take the transpose.
  + Before, we had to the take the transpose because we were computing **p\_h\_given\_v**.
* The reason is "Matrix-Product": W = ***matrix*** of size ***nh*** number of hidden *nodes*, and ***nv*** the number of visible *nodes*.

**x=** *vector of visible nodes* ***nv***.

So is not possible because ***no. of rows of W*** need to same with ***no. of columns of x***.

That’s why we need to take transpose,

* But in the case of **y**, it is *vector of hidden nodes* ***nh***.

So is possible. And we *not need to take transpose*.

**def** **sample\_v**(self, y):

        wy = **torch.mm**(y, self.W)

* Activation-value:

activation = wy + self**.b.expand\_as**(wy)

* We use ***b*** for the ***bias*** of the visible nodes. And we use **expand\_as**(wy) to make ***b*** as same format of ***wy*** to apply the bias to each line of the mini-batch.
* We take the sigmoid activation function, and we return the *probabilities* that the ***visible nodes = one*** given the ***activations*** of the ***hidden nodes***. And we also return some *sample* using Bernoulli distribution.

        p\_v\_given\_h = **torch.sigmoid**(activation)

**return** p\_v\_given\_h, **torch.bernoulli**(p\_v\_given\_h)

**14.4.10 RBM-architecture : RBM class – train()**

The last function of this ***RBM*** class is ***tarin()*** it about Contrastive Divergence,

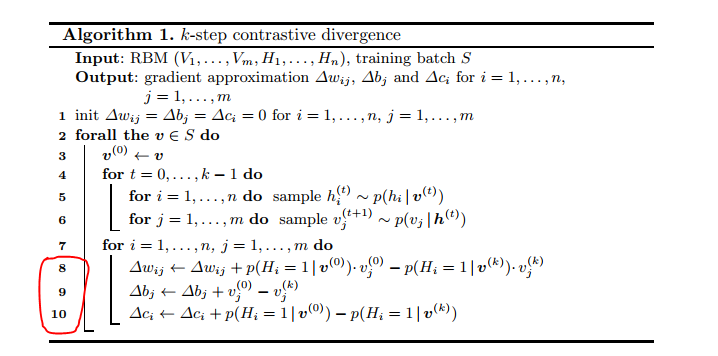
* Contrastive Divergence used to approximate the likelihood gradient. Before we implement it lets review the paper:

|  |
| --- |
|  |

* Contrastive divergence is all about *approximating* this *log-likelihood gradient*.
* Remember, the RBM is an *energy-based model*. We have this *energy function* that we are trying to *minimize*.
* This *energy function* depends on the *weight* of the model, i.e. *tensor of weights* in our model.
* We need to *optimize* these *weights* to *minimize* the *energy*.
* And not only it can be seen as an Energy-Based Model but it can also be seen as a Probabilistic Graphical Model.
* In that case, the goal is to maximize the ***log-likelihood*** of the *training set* (equivalent to *minimizing* the *energy*).
* In general, for any Deep-Learning/Machine-Learning model, we'll *minimize the* energy or *maximize the* log-likelihood to compute the ***gradient***.
* Because the *direct* *computations* of the *gradient* are *too heavy* and therefore, *instead of computing it directly*, we are gonna try to *approximate* it.
* These *gradient approximations* helps us to make these *tiny adjustments* in the direction of the *minimum energy*.

|  |
| --- |
| * It is similar what we did in ANN-part: trying to *minimize* a *loss function* and through Stochastic Gradient Descent (SGD) we were ***updating*** the ***weight*** in the ***direction*** of this ***minimum loss***. * But here in RBM, the *computations* are *too heavy* to compute these gradients *directly,* and so we need to come up with another solution, which is to *approximate* these *gradients*. * And the algorithm that will allow us to this is Contrastive Divergence. This comes with Gibbs sampling. |

* Gibbs sampling: ***Gibbs sampling*** consists of creating this ***Gibbs chain*** in ***k-steps*** and this *Gibbs chain* in *created* exactly by *sampling K times* the *hidden nodes* and the *visible nodes*.
* That is, we start with our *input vector* , based on the *probabilities* , we *sample* the *first* *hidden* nodes (first iteration).
* Then we take these *sampled hidden nodes* as *input*. Let's call them to *sample* the *visible nodes* with the probabilities .
* Then again we use these sample visible nodes for 2nd iteration, let's call them , to *sample again* the *hidden nodes* with the *probabilities* and then again we ***sample the visible nodes*** and we ***sample the hidden nodes*** and we do this ***K times***.
* And that's exactly what this CD-K algorithm (K-step contrastive divergence) is about. The algorithm is below:



* So, in our ***train()*** function of the RBM class, we will simply ***implement*** these three lines ***8, 9, 10***. And we will simply *update* our *tensor* of *weights,* ***W***, our *visible-node-bias* ***b***, and our *hidden-node-bias* ***a***.
* In the paper 's are *hidden-node-bias*, but in our algorithm we call them ***a***. That's the bias of ***p\_h\_given\_v***.
* **train():** ***train()*** function has several arguments. (five arguments total).

1. The first one is, ***self*** to use object entities. We also add four more arguments, which are:
2. Input Vector: We name it **v0**. That's our *input vector* containing the *ratings* of all the movies by *one user*.
3. vk: ***vk*** represents *visible nodes* obtained *after K samplings* (after K round trips from the *visible nodes* to the *hidden nodes first* and then way *back* from the *hidden nodes* to the *visible nodes*).

* So that's the ***visible nodes*** obtained after ***K iterations*** and ***K contrastive divergence***.

1. **ph0**: ***ph0*** is *vector* of *probabilities* that at the *first iteration* the **hidden nodes = 1** given the values of **v0**, (input vector).
2. ***phk:*** ***phk*** will correspond to the *probabilities* of the *hidden nodes* after *K sampling* given the values of the *visible nodes*, ***vk***.

* Inside the function: We will first update our *tensor of weights*, ***W*** then our *bias* ***b*** and then our *bias* ***a***.
* Updating weights: We take our weights and then add the product of *rating of the movie j*  and the *probabilities* that the **hidden nodes = 1** given the values of **v(0)**.

And then we subtract the product of *probabilities* that the **hidden nodes = 1** given the values of **v(k)**after K iterations multiply by *rating of the movie*  (the *value* of the *visible node* corresponding to the *movie* **j** after **k** *iterations*.

* **torch.mm**(**v0.t**(), ph0). And  **torch.mm**(**vk.t**(), phk)).

**def** **train**(self, v0, vk, ph0, phk):

        self.W += (**torch.mm**(**v0.t**(), ph0) - **torch.mm**(**vk.t**(), phk))

* Updating bias b: We use the ***torch.sum()*** to calculate the difference between the *input vector* of observations and the *visible nodes* after *K samplings*, **vo - vk** .
* We also gonna add ***zero*** just to keep the format of **b** as a *tensor* of *two dimensions*.

self.b += **torch.sum**((v0 - vk), 0)

* Updating bias b: **a** contains the bias of the probabilities ***p\_h\_given\_v***.
* The probabilities that the ***hidden nodes = 1***, given the values of **v0**, the input vector of observations. **ph0**
* The probabilities that the ***hidden nodes = 1***, given the values of **vk**, that is, the *values* of the *visible nodes* after *K sampling*. **phk**
* We add ***zero*** to keep the format of **a** as a *tensor* of *two dimensions*.

self.a += **torch.sum**((ph0 - phk), 0)

**def** **train**(self, v0, vk, ph0, phk):

        self.W += (**torch.mm**(**v0.t**(), ph0) - **torch.mm**(**vk.t**(), phk))

        self.b += **torch.sum**((v0 - vk), 0)

        self.a += **torch.sum**((ph0 - phk), 0)

* Finally our ***RBM class*** is over. We now have what's at the heart of the RBM algorithm:

1. Our sampling functions, ***sample\_h()*** and ***sample\_v()***
2. Training function ***train()*** that will to *Contrastive divergence* *solution* with Gibbs sampling.

***RBM-class all at once***

# *Creating the architecture of the Neural Netwark*

**class** RBM():

**def** **\_\_init\_\_**(self, nv, nh) :

        self.W = **torch.randn**(nh, nv)

        self.a = **torch.randn**(1, nh)

        self.b = **torch.randn**(1, nv)

**def** **sample\_h**(self, x):

        wx = **torch.mm**(x, self**.W.t**())

        activation = wx + self**.a.expand\_as**(wx)

        p\_h\_given\_v = **torch.sigmoid**(activation)

**return** p\_h\_given\_v, **torch.bernoulli**(p\_h\_given\_v)

**def** **sample\_v**(self, y):

        wy = **torch.mm**(y, self.W)

        activation = wy + self**.b.expand\_as**(wy)

        p\_v\_given\_h = **torch.sigmoid**(activation)

**return** p\_v\_given\_h, **torch.bernoulli**(p\_v\_given\_h)

**def** **train**(self, v0, vk, ph0, phk):

        self.W += (**torch.mm**(**v0.t**(), ph0) - **torch.mm**(**vk.t**(), phk))

        self.b += **torch.sum**((v0 - vk), 0)

        self.a += **torch.sum**((ph0 - phk), 0)

* Now we have left to do is,
* Create an ***object*** of this ***RBM class***. This will be our ***model*** itself.
* Then we will train the ***model*** over *several epochs* so that we find these *optimal weights* that will *predict* the *ratings of the movies* that were *not originally rated*.
* Now RBM class is *ready*, we can create *object* of this *class*. Also we can create *several* RBM models using this *class*.
* We can *test* many of them with *different configurations*, with several numbers of *hidden nodes* because that's basically the main parameter.
* You can add some *more parameters* in your class, for example, a *learning rate* if you want to improve and tune your model.