Chapter 14: Part 5

**Deep Learning**

**BM project - part 2: Building RBM model**

**14.5.1 RBM-model**

To create our first RBM object we just call the class RBM with the two parameters: *no. of visible node =* ***nv*** and *no. of hidden node =* ***nh***.

* The object will then initialized (***\_\_init\_\_()*** will invoked), with biases ***a***, ***b*** and weight ***W***.
* Other functions ***sample\_h()***, ***sample\_v()*** and ***train()*** will be used during the training, they will not take any action unless they are called.
* no. of visible node: Here ***nv*** *= no. of visible node = is the number of movies*; is a fixed parameter.
* At the start the *visible nodes* are the *ratings* of all the movies by a specific user and so we have *one visible node* for *each movie*.

nv = **len**(train\_set\_tensor[0])

***train\_set\_tensor[0]*** is the length of the *first column* of *train-tensor*, which corresponds to movies. We have 1682 movies so 1682 visible nodes.

* no. of hidden node: Here we can choose any number for ***nh***. The hidden nodes correspond to some features that are going to be detected by the RBM model. These features will be some *GENRE*, some *ACTORS*, some *DIRECTORS* whether the movie's got an *OSCAR*, etc.
* So the *number of* *hidden nodes* corresponds to the *number of features* we want to detect.

nh = 100

So let's say that we want to start by detecting ***100 features***. It's actually hard to say right now about the *optimal number of features*. Of course we can tune it later.

* Batch size: We already mentioned this concept of *batch* when we add extra dimension during *creation of batches a & b.* This ***dimension*** here represented by, **1**. So **1** corresponds to the batch.

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

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

* When we train our algorithm, we will *update* the *weights* after a *batch* *observations* (instead of each observation). Each batch will contain same number of observations.
* Also this parameter is tunable. We can adjust it to improve the model.
* Now, we can set ***batch\_size = 1***, in that case you're *updating* the *weights* after *each* *observation* going to the network. It will be very slow for large amount of data.
* For fast training we can take a large batch size for example ***batch\_size = 100***. Since we have 943 observations the training will go very fast with batch size = 100.

batch\_size = 100

* Finally we create the RBM object with parameters, ***nv*** & ***nh***.

# *creating RBM model*

nv = **len**(train\_set\_tensor[0])

nh = 100

batch\_size = 100

rbm = **RBM**(nv, nh)

**14.5.2 Training the RBM-model - part 1**

To train the model over each epoch, we use a for loop. This loop will run for specific number of epoch. We just need to include the different functions (that we made in this RBM class) inside this for loop.

* Epoch: We first set the number of epochs. Let's choose for now 10 epochs. Because we have only 943 observations and besides we only have binary value **0** and **1** so the *convergence* will be reached *pretty fast*.

# *Training the RBM*

nb\_epoch = 10

**for** epoch **in** **range**(1, nb\_epoch + 1):

    train\_loss = 0

    s = 0.0

* For each epoch, *all our observations* will go into the network.
* We will update the weights after the observations of *each batch* passed through the network.
* At the end we'll get our *final visible nodes* with the *new ratings* for the *movies that were not originally rated*.
* Inside the for loop: Note that, for any deep learning algorithm we need a ***loss function*** to *measure* the *error* between the *predictions* and the *real* *ratings*.
* Loss: In this training we will compare the predicted ratings to the ratings of the training set.
* Basically we will measure the *difference* between the *predicted ratings*, that is either ***0*** or ***1***, and the real ratings, ***0*** or ***1***.
* For measuring the loss we can use different methods. Here in RBM we use *simple difference* in *absolute values*. that measures, simply, the *absolute difference* between the *predicted rating* and the *real rating*.
* But the most common loss measuring method is RMSE, the Root Mean Square Error, which is the *root* of the *mean* of the *squared differences* between the *predicted* *ratings* and the *real* *ratings*. We'll use RMSE in next chapter to build Auto-Encoders.

**train\_loss = 0**

We will initialize it to ***zero***, because before we started training this ***loss*** is equal to ***0***. And then the ***loss*** will increase when we ***find*** some ***errors*** between the *predicted ratings* and the *real ratings*.

* Counter: Here we're gonna need a ***counter***, we name it "***s***". Using this counter ***s***, we're going to normalize the *train\_loss* ( simply divide the ***train\_loss*** by ***s***) . We will increment it after each epoch

**s = 0.0**

To make ***s*** a *floating-number* we use ***0.*** Or ***0.0***, either will work.

* Nested for loop for bath operation: When we made ***sample\_h()***, ***sample\_v()*** and ***train()*** functions regarding to one user.
* But the samplings and the *contrastive* of *divergence* algorithm have to be done over *all the users*, but remember over *all the users* in the *batch*.
* To get these batches of users, we need ***another for loop*** (a nested for loop).
* We loop over all the users and ***increment*** the user by ***batch\_size***, It will divide the dataset into batches.

**for** u\_id **in** **range**(0, nb\_users - batch\_size, batch\_size):

Since we increment the loop variable by ***batch\_size***, the upper bound must ***nb\_users - batch\_size***. This loop creates the batches, for user 0 to 99, 100 to 199 etc.

That’s how we implement *batch learning* from *scratch*. Everything happens in this loop.

|  |
| --- |
| * For lop syntax:   **for** i **in** **range**(lower\_bound, upper\_bound, increment): |

* Inside the nested loop:

1. We will get separately, our ***input*** and our ***target***.

* Our *input* is the *ratings* of all the *movies* by the *specific user* we're dealing with right now in the loop.
* The target is going to be at the beginning the same as the input.
* Since the ***input*** is gonna go into the *Gibbs chain* and will be *updated* to get the *new ratings* in *each visible node* then the ***input*** is going to change, but the ***target*** will keep its same initial value.
* input vector: We name the *input vector* ***vk***, it will go through the *Gibbs chain* and will be *updated* at each round trip. ***vk*** is going to be the *output* of *Gibbs sampling*, after the *K steps* of the random walk.
* But at the start this ***vk*** is actually *same a*s the *input batch* of observations (the *ratings* of the *users*), in the *batch*. The ratings that *already existed*.
* To get is all the users in a batch starting from ***u\_id*** up to ***u\_id + batch\_size***, we use following code:

**vk = train\_set\_tensor[u\_id:(u\_id + batch\_size)]**

We selected the users from ***train\_set\_tensor***, from ***u\_id*** up to ***u\_id + batch\_size***.

* Target: target is the same as the initial value of input at the beginning but it remain unchanged.
* It's the *batch* of *original ratings* that we want to *compare*, in the end, to our *predicted ratings*.
* We need it to *measure* the ***error*** between the *predicted ratings* and the *real ratings* to calculate ***train\_loss***.
* So, we're gonna call this target ***v0***, so ***v0*** are our ratings of the movies that were *already rated* by the *100 users* in this *batch*.

v0 = train\_set\_tensor[u\_id:(u\_id + batch\_size)]

* Initial Probabilities: Before starting Contrastive Divergence with Gibbs Sampling we need to specify our *initial probabilities*.
* The initial probabilities, is actually ***ph0*** is the probabilities that the hidden node at the start equal **1** given the real ratings **ph0**
* Now we use our ***sample\_h()*** function of RBM class. Since we are getting the probabilities that the hidden node = **1** given the visible nodes at the beginning.
* Because ***sample\_h()*** returns ***p\_h\_given\_v***, that is here it will return ***p\_h\_given\_v0***.
* Using \_ : Since this ***sample\_h()*** function also returns both *probabilities* and the *samples* (Bernoulli samples) "**return** **p\_h\_given\_v, torch.bernoulli(p\_h\_given\_v)**", well we have to use a Python trick to only get the function returned first element, i.e. ***p\_h\_given\_v***, and the Python trick to do that is to add here a ***comma*** and then an ***underscore***,

ph0,\_ = **rbm.sample\_h**(v0)

* The parameter of ***sample\_h()*** is x, and x corresponds to the *visible node*, because we want to *sample* the *hidden nodes* given the *visible* *nodes*.
* Which visible nodes: We want the visible nodes at the start, that is, **v0**. That is the *original ratings* of the movies for all the *users* of our *batch*.

# *Training the RBM*

nb\_epoch = 10

**for** epoch **in** **range**(1, nb\_epoch + 1):

    train\_loss = 0

    s = 0.0

**for** u\_id **in** **range**(0, nb\_users - batch\_size, batch\_size):

        vk = train\_set\_tensor[u\_id:(u\_id + batch\_size)]

        v0 = train\_set\_tensor[u\_id:(u\_id + batch\_size)]

        ph0,\_ = **rbm.sample\_h**(v0)

**for** k **in** **range**(10):

**14.5.3 Training the RBM-model - part 2 : k-step CD**

* Another nested for loop: This for loop for the *K steps of contrastive divergence*. In this for loop that we're gonna make the Gibbs chain, that we're gonna do our k steps of the random walk. Let's call the looping variable ***k*** in ***range(10)***.

**for** k **in** **range**(10):

* Actually our *k-steps* of the *Random* *Walk* in *Gibbs sampling* is an MCMC technique (Markov Chain Monte Carlo technique).
* Here we'll explain what's going on with this random walk.
* Basically, Gibbs sampling consists of making Gibbs chain. There are simply several round trips from the *visible nodes* to the *hidden nodes*, and then from the *hidden nodes* to the *visible nodes*.
* In each round trip of this *Gibbs chain* of *Gibbs sampling* well, the *visible nodes* are *updated*. and step after step, we get closer to our good *predicted ratings*.
* At the ***beginning*** we start with our ***input batch*** of observations. That is our ***input ratings*** in ***v0***, in our ***batch*** of ***100 users***.
* Sampling Hidden Nodes: In the first step of Gibbs sampling, from this batch input vector ***v0*** of original ratings, we are going to sample the first hidden nodes using ***Bernoulli sampling*** using our ***p\_h\_given\_v0*** distribution from ***sample\_h()*** function.
* So the first step of *k-steps contrastive divergence* is to call ***sample\_h()*** on the visible nodes, to get the *first sampling* of the *first* *hidden nodes*.
* Since we actually want to get this second element returned by ***sample\_h()***, we use the similar python trick using '**\_**' as we did before.
* In this time, we are going to start with an **underscore** '**\_**' and then **hk**, to be the *hidden nodes* obtained at the *k-th step of contrastive* *divergence*.

\_,hk = **rbm.sample\_h**(vk)     # *sampling hidden nodes*

* Since we are doing the sampling of the *first hidden nodes*, given the values of the *first visible nodes*, that is our original ratings, the first *input* for our ***sample\_h()*** function in this first step of Gibbs sampling is ***v0***.
* But be careful, ***v0*** is the ***target***, which we *don't wanna change*. So we have to take this ***vk***, because at the first step ***v0 = vk***. ***vk*** so far is our input batch of observations, and then ***vk*** will be *updated*.
* Sampling Visible Nodes: We'll update ***vk*** right after we update ***hk*** with the other sampling function which is ***sample\_v()***.
* So right now , but in the next step we update ***vk*** so that .
* ***vk*** is going to be the *sampled visible nodes* after the first step of *Gibbs sampling*, i.e after getting ***hk***.
* We'll get it using ***Bernoulli sampling*** using our ***p\_v\_given\_h*** distribution from ***sample\_v()*** function on the *first sample* of our *hidden nodes* i.e. ***hk***.
* Again we get this second element returned by ***sample\_v()***, we use the similar python trick using *underscore* '**\_**' .

\_,vk = **rbm.sample\_v**(hk)     # *sampling visible nodes*

* So we get our *first update* of the *visible nodes* after the *first sampling*.

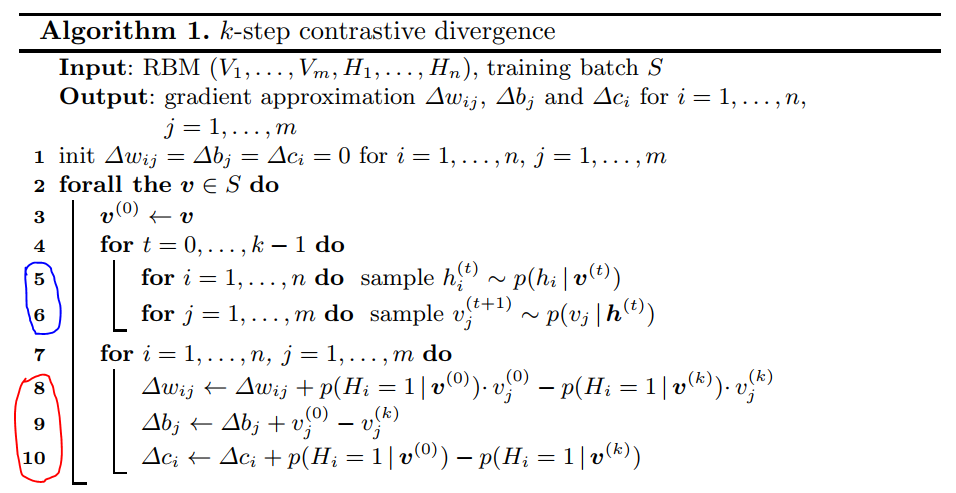
That's the first *step* of our *random walk*, that is the first step of *Gibbs sampling*.

**for** k **in** **range**(10):

            \_,hk = **rbm.sample\_h**(vk)     # *sampling hidden nodes*

            \_,vk = **rbm.sample\_v**(hk)     # *sampling visible nodes*

* So as the loop continues ***hk*** and ***vk*** update themselves repeatedly.



* In continues until the end of the loop, when we'll get the *10th sample* of *hidden nodes* and the *10th sample* of *visible nodes*.
* Getting *hk* and *vk* after the *last* *step* of the *random walk*, we can now ***approximate*** the ***gradients*** using 8, 9, and 10th steps of *k-step-Contrastive-divergence algorithm*.
* What we did above is the step 5 & 6 of *k-step-Contrastive-divergence algorithm*.
* We want to get the *last sample* after the *last step* of the *random walk*, the *last sample of* hiddennodes and the *last sample of* visible nodes.
* We need those last samples to update the ***weight*** and the ***bias*** to approximate the ***gradient***.

|  |
| --- |
| * Actually here we need the *last sample of* visible nodes. * But we don't directly use the *last sample of the* hiddennodes, that was just to get the *last sample* *of the* visible nodes. * We just got our ***vk***, we can now *approximate* the *gradient* to update the ***weights*** and the ***bias***. * We use ***tarin()*** function to update the ***weights*** and the ***bias***, from our ***RBM*** class. |

* Ignore unrated movies during weight update: But before we apply this ***train()*** function to update the ***weight***, well we need to do something very important:
* We don't wanna learn from the movies which has no rating, that is for the ***cells (movies)*** that have a ***minus one -1*** (no rating).
* We'll ignore these cells that contain the ***-1 ratings*** in the training process,
* We're going to freeze these visible nodes that contain the ***-1 ratings***. So that it *won't be possible* to *update* them during *Gibbs* *sampling*.
* How can we freeze these visible nodes containing the minus one ratings? We need to take our ***vk*** - visible nodes, and inside the for-loop, we keep these -1 values same as follows:

**for** k **in** **range**(10):

            \_,hk = **rbm.sample\_h**(vk)     # *sampling hidden nodes*

            \_,vk = **rbm.sample\_v**(hk)     # *sampling visible nodes*

            vk[v0 **<** 0] = v0[v0 **<** 0]

* That is, we use **v0**'s **-1**'s to make **vk**'s unrated movies to *keep away from updating*.
* vk[v0 **<** 0] gonna get the nodes that have a ***-1*** rating in ***vk***, that were *not originally rated* by the *users*.
* v0[v0 **<** 0] will replace those nodes by v0's original/unchanged values that were *not originally rated* by the *users*. The *original minus one ratings* from the target ***v0***, because it is *not updated*.
* We used v0 **<** 0, to get the ***minus one ratings*** because our ratings are either ***-1***, ***0***, or ***1***.
* By doing this, we make sure that the *training* is *not done* on these *ratings* that were *not actually existed* (not-rated by users).

So now we can get out of this third **for loop** and soon enough we will start the training.

**14.5.4 Training the RBM-model - part 3 : train()**

* **phk:** We want to apply the ***train()*** function to update the ***weight*** and the ***bias***. But, you notice that in this ***train()*** function, we need

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

1. target, ***v0***,
2. our *sampled visible nodes* at the *last step* of the random walk, ***vk***,
3. The initial probabilities, ***ph0***
4. And *k-step probabilities,* ***phk***.

* Notice, we don't have any ***phk***. We didn't calculated it because ***vk*** wasn't updated at start.
* So before applying the ***train()*** function we compute ***phk***.
* We want to get the *first element* returned by the ***sample\_h()*** function.
* We apply ***sample\_h()*** on ***vk***, the last sample of the visible nodes (which we got from the end of 3rd **for**-loop), the last sample visible nodes, at the 10th step of the *random* *walk*, *Gibbs* *sampling*.

        phk,\_ = **rbm.sample\_h**(vk)    # *probabilities after k-step*

Now we got everything we need to apply the ***train()*** function.

* ***train():*** Now let's apply the ***train()*** function. It's going to be kid stuff.
* Note that, the ***train()*** function doesn't return anything. It just updates the weights according to the steps 8, 9 and 10 of *k-step-Contrastive-divergence algorithm*.
* So we don't need any new variable because it doesn't return anything.
* We just take our ***rbm*** object of the RBM class and apply ***train()***, using ***v0***, ***vk***, ***ph0*** and ***phk*** values for our four arguments.

# *creating RBM model*

nv = **len**(train\_set\_tensor[0])

nh = 100

batch\_size = 100

rbm = **RBM**(nv, nh)

# *Training the RBM*

nb\_epoch = 10

**for** epoch **in** **range**(1, nb\_epoch + 1):

    train\_loss = 0

    s = 0.0

**for** u\_id **in** **range**(0, nb\_users - batch\_size, batch\_size):

        vk = train\_set\_tensor[u\_id:(u\_id + batch\_size)]     # *input vector*

        v0 = train\_set\_tensor[u\_id:(u\_id + batch\_size)]     # *target vector*

        ph0,\_ = **rbm.sample\_h**(v0)    # *initial probabilities*

            # *applying k-step contrastive divergence*

**for** k **in** **range**(10):

            \_,hk = **rbm.sample\_h**(vk)     # *sampling hidden nodes*

            \_,vk = **rbm.sample\_v**(hk)     # *sampling visible nodes*

            vk[v0 **<** 0] = v0[v0 **<** 0]     # *preventing updates to unrated nodes*

        phk,\_ = **rbm.sample\_h**(vk)    # *probabilities after k-step*

**rbm.train**(v0, vk, ph0, phk) # *train RBM model*

* Now the *training* is going to happen. The *weight* and the *bias* are going to be *updated* *towards* the direction of maximum likelihood.
* And therefore, all our *probabilities* , of *visible node* given the states of the *hidden nodes* will be more and *more relevant*,
* It will get the *largest* *weights* for the *probabilities* that are the *most significant* and eventually that will *lead* *us* to some *predicted* *ratings* that are going to be *close* to the *real ratings*.

**14.5.5 Training the RBM-model - part 4 :** measuring error- **train\_loss**

We now calculate ***train\_loss*** to measure how *close* the *predicted* *ratings* to the *real ratings*.

* So we now update the ***train\_loss***, which previously set to ***0***. Because right now we actually have our *predictions*, after updating the *weights* using ***train()***.
* Here **torch.abs**(vk[v0 **>=** 0] - v0[v0 **>=** 0]) finds the tensor *of absolute values* of the difference **vk – v0**.
* We only take the non-negative-values of both **vk** and **v0**, using condition **"v0 >= 0"**
* Finally we use **torch.mean**() to find the *simple difference* in the *tensor of absolute values*.
* Here in RBM we use *simple difference* in *absolute values*. that measures, simply, the *absolute difference* between the *predicted rating* and the *real rating*.
* But the most common loss measuring method is RMSE, the Root Mean Square Error, which we'll use in next chapter to build Auto-Encoders.
* Also we increment counter **s** and ***train\_loss*** inside the 2nd ***for loop***.
* Outside the 2nd ***for loop*** we calculate the normalized *train\_loss*. And ***print*** this value and epoch no.

        # *Error rate*

        train\_loss += **torch.mean**(**torch.abs**(vk[v0 **>=** 0] - v0[v0 **>=** 0]))

        s += 1.0

    loSS = train\_loss/s

**print**(f"Epoch no. {epoch}.\t loss = {loSS}")

* At the last steps, the *weights* are going *close* to the *optimal weights*. The *optimal sample visible* nodes after the *10 steps* of *Gibbs sampling* contains our best *predicted ratings*.
* We compared the *non-negative* values of ***vk*** (the last visible nodes after the last batch of users that went through the network; after the last step of contrastive divergence) and ***v0*** (the target vector), take their *absolute values* and calculated the simple mean-distance of those absolute values. The *simple* *distance* in *absolute values* between the *predictedn* and the *real rating*.
* Remember in the training we didn't include the ratings that were actually non-existent, the -1 ratings were ignored. Hence we considered the non-negative values here.
* When we compute this difference between the *real original ratings* and the *predictions*, well, we have to take them for the ones that *actually exist*.

# *Training the RBM*

nb\_epoch = 10

**for** epoch **in** **range**(1, nb\_epoch + 1):

    train\_loss = 0

    s = 0.0

**for** u\_id **in** **range**(0, nb\_users - batch\_size, batch\_size):

        vk = train\_set\_tensor[u\_id:(u\_id + batch\_size)]     # *input vector*

        v0 = train\_set\_tensor[u\_id:(u\_id + batch\_size)]     # *target vector*

        ph0,\_ = **rbm.sample\_h**(v0)    # *initial probabilities*

            # *applying k-step contrastive divergence*

**for** k **in** **range**(10):

            \_,hk = **rbm.sample\_h**(vk)     # *sampling hidden nodes*

            \_,vk = **rbm.sample\_v**(hk)     # *sampling visible nodes*

            vk[v0 **<** 0] = v0[v0 **<** 0]     # *preventing updates to unrated nodes*

        phk,\_ = **rbm.sample\_h**(vk)    # *probabilities after k-step*

**rbm.train**(v0, vk, ph0, phk) # *train RBM model*

        # *Error rate*

        train\_loss += **torch.mean**(**torch.abs**(vk[v0 **>=** 0] - v0[v0 **>=** 0]))

        s += 1.0

    loSS = train\_loss/s

**print**(f"Epoch no. {epoch}.\t loss = {loSS}")

|  |
| --- |
| * One Fix before executing the model: For the latest update of **PyTorch** we found the following error:   RuntimeError: The expanded size of the tensor (1682) must match the existing size (100) at non-singleton dimension 1  19 return p\_v\_given\_h, torch.bernoulli(p\_v\_given\_h)  20 def train(self, v0, vk, ph0, phk):  ---> 21 self.W += torch.mm(v0.t(), ph0) - torch.mm(vk.t(), phk)  22 self.b += torch.sum((v0 - vk), 0)  23 self.a += torch.sum((ph0 - phk), 0)   * Now, **print(rbm.W.size())** will show you **torch.Size([100, 1682])**   print((torch.mm(v0.t(), ph0)-torch.mm(vk.t(), phk)).size()) will show you **torch.Size([1682, 100])** |

|  |
| --- |
| * So it looks like we should take transpose of (torch.mm(v0.t(), ph0)-torch.mm(vk.t(), phk)), so it should be something like (**torch.mm**(**v0.t**(), ph0) - **torch.mm**(**vk.t**(), phk)).**t**(): we simply take transpose using **t()**. * So inside our RBM class we update this code:   **class** RBM():  **def** **\_\_init\_\_**(self, nv, nh) :  . . . .  . . . .  . . . .  **def** **train**(self, v0, vk, ph0, phk):          self.W += (**torch.mm**(**v0.t**(), ph0) - **torch.mm**(**vk.t**(), phk)).**t**() |

# *-------- 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)).**t**()

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

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

# *creating RBM model*

nv = **len**(train\_set\_tensor[0])

nh = 100

batch\_size = 100

rbm = **RBM**(nv, nh)

# *Training the RBM*

nb\_epoch = 10

**for** epoch **in** **range**(1, nb\_epoch + 1):

    train\_loss = 0

    s = 0.0

**for** u\_id **in** **range**(0, nb\_users - batch\_size, batch\_size):

        vk = train\_set\_tensor[u\_id:(u\_id + batch\_size)]     # *input vector*

        v0 = train\_set\_tensor[u\_id:(u\_id + batch\_size)]     # *target vector*

        ph0,\_ = **rbm.sample\_h**(v0)    # *initial probabilities*

            # *applying k-step contrastive divergence*

**for** k **in** **range**(10):

            \_,hk = **rbm.sample\_h**(vk)     # *sampling hidden nodes*

            \_,vk = **rbm.sample\_v**(hk)     # *sampling visible nodes*

            vk[v0 **<** 0] = v0[v0 **<** 0]     # *preventing updates to unrated nodes*

        phk,\_ = **rbm.sample\_h**(vk)    # *probabilities after k-step*

**rbm.train**(v0, vk, ph0, phk) # *train RBM model*

        # *Error rate*

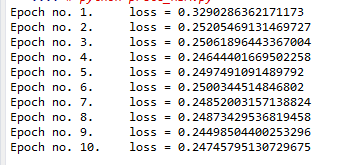
        train\_loss += **torch.mean**(**torch.abs**(vk[v0 **>=** 0] - v0[v0 **>=** 0]))

        s += 1.0

    loSS = train\_loss/s

**print**(f"Epoch no. {epoch}.\t loss = {loSS}")

* Training the model: After executing the above codes we get following result:



* And we end up with a ***train loss*** of ***0.25*** which is pretty good because that means that in the ***training set***, well we get the correct predictive rating, ***three*** times out of ***four***.

|  |
| --- |
| * Next to apply on test-set: Now, we need to ***evaluate*** our ***model*** on new observations and that is what the ***test\_set*** is for. * Next, we will make our predictions on the test set ***without*** doing any ***training***, of course. * And also we will compute a ***test\_loss***, which will be the same *mean of the absolute distance between* the *predictions* and *the ratings*. * We're hoping to get a ***test\_loss*** that is around this ***0.25*** value. If it is around it, that means that even on *new observations* well we predict *correctly*, *three* *ratings* out of *four*. So that would be amazing (no-overfitting). |

**14.5.6 RBM-model : Evaluating on Test-set**

Now we test our RBM on ***test\_set***. We will see if the results are *close* to the *training set results*. That is if even on *new observations*, we can predict *three* *correct* ratings out of *four*.

* These are binary ratings, and we would definitely succeed at making a recommended system.
* Getting the ***test\_set*** results is going to be quite similar as getting the ***train\_set*** results. The only difference is that there is not gonna be a training.
* So we will *remove* at least one *for loop*. And we also use *MCMC techniques*. MCMC-Markov chain Monte Carlo techniques is the essential, the *crucial point to understand* here.
* Now we copy above training-code and we will make the required change.
* There is no training, so no epochs and first for-loop. We re-align everything.
* We rename ***train\_loss*** to ***test\_loss***,
* Then we will rename the counter to ***cnt*** we initialize at *zero*, and we will increment it by one at each step.
* Now we loop over all the users in ***test\_set***.
* We do not need a ***batch\_size***. Because the batch size is just a technique specific to the training. During training, *batch size* is a parameter that you *tune* to get *more* or *less* better *performance* on *results* on the *training set*, and therefore, on the test set.
* So we remove the ***batch\_size*** step of the for-loop and we only take our users, up to the last user.
* Our *model* will make some *predictions* for *each* of the *users* one by one.
* And therefore, we can also remove the **0** from **range()** of the for-loop
* Inside the for-loop:
* We rename ***vk*** to ***v*** and ***v0*** to ***vt***. So ***v*** will be out *input vector* and ***vt*** as *target vector*. ***v*** is the ***input*** on which we will make the prediction.
* Also we change **[u\_id:(u\_id + batch\_size)]** to  **[u\_id:(u\_id + 1)]**, notice we replace **batch\_size** with **1**. Since we are gonna make some predictions for each of the users one by one, we will replace this ***batch\_size*** by ***1***.
* Keep the train-set for input vector & test-set for target vector: Note that (important), we use the *test\_set* for the *target vector* but for the *input vector*, we use our *train\_set*.

# *Evaluating the RBM on Test-set*

test\_loss = 0

cnt = 0.0

**for** u\_id **in** **range**(nb\_users):

    v = **train\_set\_tensor**[u\_id:(u\_id + 1)]     # *input vector*

    vt = test\_set\_tensor[u\_id:(u\_id + 1)]     # *target vector*

|  |
| --- |
| * Since we are dealing with the ***test set*** so we are trying to ***predict*** the ***ratings*** in the ***test set***. So it would make sense to use ***test\_set*** for both ***v*** & ***vt***. * But that would be wrong. * Target vt: Indeed, we need ***test\_set*** for ***vt*** because we want to compare the *real ratings* of the *test set* to our *predictions*. Because ***vt*** contains the original ratings of the test set, so that is what we will *compare* to our *predictions* in the end. * Input v: But here for ***v***, the input, we actually need to keep the ***training set***. It's because, the ***training set*** is the ***input*** that will be ***used*** to ***activate*** the ***hidden neurons*** to get the ***output***. * *We need to understand this first crucial point:* We are using the *inputs* of the *training set* to *activate* the *neurons* of the *RBM* to get the *predicted ratings* of the *test set*. * Right now the *training set* contains the *ratings* of the *training set* and it *doesn't contain the answers* of the *test set*. * But, by using the *inputs* of the *training set* we will *activate* the *neurons* of our *RBM* to *predict* the *ratings* of the movies that were *not* *rated yet*, and those *will be the* ***ratings*** of the ***test set.*** * So we need this training-set, ***train\_set\_tensor*** as input to get the predicted ratings of the *test set*. Because *we* are *getting* these *predicted ratings* from the *inputs* of the *training set* that are used to *activate* the *neurons* of our *RBM*. |

* ph0,\_ = **rbm.sample\_h**(v0) computes probabilities that the hidden node equal **1** given the real ratings

**ph0**

* ***ph0*** was needed to ***train*** the ***model***, therefore, we don't need it for the ***test set***.
* MCMC & Blind walk: The *MCMC* techniques, *Markov chains Monte Carlo* techniques is related to the random walk and more precisely the blind walk.
* To get our predictions of the test set ratings, do we need to apply ***again*** the ***k step contrastive divergence***?
* More precisely, do we need to make *k steps* of the *random walk* (i.e. 10 step for-loop), that is 10 steps of the random walk?
* Actually we need one step of the *random walk (blind walk)*, hence we don’t need the *for-loop* for *k-step* (i.e 10 step) to get our *final* *prediction*. So we only need *only one step of contrastive divergence*.
* Also this is not exactly the random walk because in the *random walk* the *probabilities* are the *same*. Here, even if it's a *Markov chain* the *probabilities* are *not the same* so it's *not a random walk*, so it's rather a Blind Walk.
* The Principle Of The Blind Walk is that, imagine you were *blindfolded* and you had to make *100 steps* on a *straight line* without getting *out* of the straight line.
* You will be *trained* with *Gibbs sampling* to make *100 steps* by *staying* on the *straight line* but you're *blindfolded* so you know it's not easy to make some steps and always stay on the *straight line*, so, you may go a little bit on the left, on the right and after 100 steps it's hard for you to be on the straight line.
* But you were *trained* to make these *100 steps*, staying on the *straight line* being *blindfolded*. And that is by doing some *random* *steps*.
* It is close to the random walk technique and the MCMC but the difference is that in the random walk the *probabilities* are the *same* and here the *probabilities* are *different*. And that's the thing you are ***trained*** to make ***100 steps*** by ***staying*** on a ***straight line***.
* And so the ***principle*** of all this is that you are ***trained*** to do this for ***100 steps*** so that when you ***make*** ***one step***.
* When you have to take the challenge to make only one step and still be on that straight line, well you will have High Chance of SUCCESS.
* Same thing goes here, our RBM-recommender-model is ***trained*** for ***10-step (k-step)*** and now, for ***test-set*** it have to take ***only-one step***.
* That's the whole principle of the blind walk technique from MCMC, Markov chain Monte Carlo. That's close to the random walk but keep in mind that the probabilities are not the same, it is blind-walk with different probabilities.
* So our prediction will be *directly* the *result* of *one round trip* of *Gibbs* *sampling*. One iteration, one step of the blind walk. Then we'll get all our *predictions* of the *test set* in one shot.
* In this step we're gonna start with an ***if*** condition to ignore/filter the ***unrated-movies*** from the ***test-set*** (the ***-1*** are ratings that just never happened). And it's the same id we are dealing with for the train-set & the test-set.
* We use ***len()*** function to get the *real-ratings* (*original ratings* of the test set) that are existent from the target ***vt***,
* We take all the *ratings* that are *existent* using the condition "vt**>=**0".
* **len**(vt[vt**>=**0]) **>** 0 Specifies the number of the visible nodes containing the non-negative-ratings must be larger than zero, and in that condition we can make some predictions.

**if** **len**(vt[vt**>=**0]) **>** 0:

* To make the predictions we use **sample\_h**() and **sample\_v**() functions. And in this step we use ***h*** and ***v*** instead of ***hk*** and ***vk***.

        \_,h = **rbm.sample\_h**(v)     # *sampling hidden nodes*

        \_,v = **rbm.sample\_v**(h)     # *sampling visible nodes*

* For the ***input*** ***visible*** nodes we'll get only ***one*** ***hidden*** ***node*** because there is ***one step only***. i.e. one *vector of hidden nodes* to get our *final vector of predicted* *ratings*. And that's all for the only step of the blind walk.
* We don't need " vk[v0 < 0] = v0[v0 < 0]" here. We also do not use ***phk***, or ***train()***, so we get rid of those lines.
* Note that, now we have to update the ***test\_loss*** and the counter ***cnt*** inside this "if-block". These needs to be in the if-block because we are still computing the ***test\_loss*** only for the *existent ratings*.
* We are still taking the absolute distance between the *prediction* and the *target*, so we used ***torch.mean()*** and ***torch.abs()***.

test\_loss += **torch.mean**(**torch.abs**(vt[vt **>=** 0] - v[vt **>=** 0]))

* So, the *target* is ***vt*** and the *prediction* is ***v***. We take the existent ratings using the condition "vt **>=** 0" (to get the indexes of the cells that have the existent ratings).

**if** **len**(vt[vt**>=**0]) **>** 0:

        \_,h = **rbm.sample\_h**(v)     # *sampling hidden nodes*

        \_,v = **rbm.sample\_v**(h)     # *sampling visible nodes*

        # *Error rate*

        test\_loss += **torch.mean**(**torch.abs**(vt[vt **>=** 0] - v[vt **>=** 0]))

        cnt += 1.0

* At the end we calculate the normalized ***test\_loss*** using ***test\_loss/cnt***. And print the result.

eval\_loSS = test\_loss/cnt

**print**(f"Evaluation or Test loss = {eval\_loSS}")

# *Evaluating the RBM on Test-set*

test\_loss = 0

cnt = 0.0

**for** u\_id **in** **range**(nb\_users):

    v = train\_set\_tensor[u\_id:(u\_id + 1)]     # *input vector from training-set*

    vt = test\_set\_tensor[u\_id:(u\_id + 1)]     # *target vector from test-set*

**if** **len**(vt[vt**>=**0]) **>** 0:

        \_,h = **rbm.sample\_h**(v)     # *sampling hidden nodes*

        \_,v = **rbm.sample\_v**(h)     # *sampling visible nodes*

        # *Error rate*

        test\_loss += **torch.mean**(**torch.abs**(vt[vt **>=** 0] - v[vt **>=** 0]))

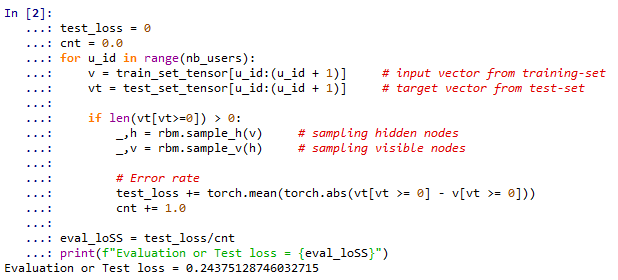
        cnt += 1.0

eval\_loSS = test\_loss/cnt

**print**(f"Evaluation or Test loss = {eval\_loSS}")

* In summery:

1. We start with a *test* *loss* of *zero* then the *counter* to *zero*.
2. We loop over all our users.
3. Then for all the ratings that are *existent* in the *test set*, we *sample* the *hidden nodes* first
4. then we use these *hidden nodes* as input to *sample* *the visible* *nodes*.
5. We update ***test\_loss*** and counter ***cnt*** inside if-block.
6. We did this for only one round trip according to the MCMC theory.



* After executing above code, we get a ***test loss*** of ***0.24***. Which is definitely excellent, because for *new observations* for *new movies* we managed to *predict* some *correct ratings* three times out of four and even better than that. Because we are slightly below ***25%***.
* Here we definitely managed to make a *robust recommended system*, but, remember this was the easy one. Predicting binary ratings ***0*** and ***1***.
* In the next chapter we'll take it to the next level because we will be predicting some ratings from ***1*** to ***5***, using the AutoEncoders.
* Obviously, working with some continuous values will increase the complexity of the problem. But it will help us to understand to build *two different recommended systems* by applying *two different deep learning models*.
* But, the good news is Auto Encoders – AE, is actually a much more simple model than Boltzmann machine.
* With AE we will get some amazing predictions even for ratings between ***1*** to ***5***.

**All Code at once (practiced) : RBM-Recommender model**

# *----------- 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. Following are the Tensors of ratings*

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

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

# *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

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

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

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

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

# *-------- 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)).**t**()

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

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

# *creating RBM model*

nv = **len**(train\_set\_tensor[0])

nh = 100

batch\_size = 100

rbm = **RBM**(nv, nh)

# *Training the RBM*

nb\_epoch = 10

**for** epoch **in** **range**(1, nb\_epoch + 1):

    train\_loss = 0

    s = 0.0

**for** u\_id **in** **range**(0, nb\_users - batch\_size, batch\_size):

        vk = train\_set\_tensor[u\_id:(u\_id + batch\_size)]     # *input vector*

        v0 = train\_set\_tensor[u\_id:(u\_id + batch\_size)]     # *target vector*

        ph0,\_ = **rbm.sample\_h**(v0)    # *initial probabilities*

            # *applying k-step contrastive divergence*

**for** k **in** **range**(10):

            \_,hk = **rbm.sample\_h**(vk)     # *sampling hidden nodes*

            \_,vk = **rbm.sample\_v**(hk)     # *sampling visible nodes*

            vk[v0 **<** 0] = v0[v0 **<** 0]     # *preventing updates to unrated nodes*

        phk,\_ = **rbm.sample\_h**(vk)    # *probabilities after k-step*

**rbm.train**(v0, vk, ph0, phk) # *train RBM model*

        # *Error rate*

        train\_loss += **torch.mean**(**torch.abs**(vk[v0 **>=** 0] - v0[v0 **>=** 0]))

        s += 1.0

    loSS = train\_loss/s

**print**(f"Epoch no. {epoch}.\t loss = {loSS}")

# *Evaluating the RBM on Test-set*

test\_set\_ratings\_list = []

pridicted\_rating\_list = []

test\_loss = 0

cnt = 0.0

**for** u\_id **in** **range**(nb\_users):

    v = train\_set\_tensor[u\_id:(u\_id + 1)]     # *input vector from training-set*

    vt = test\_set\_tensor[u\_id:(u\_id + 1)]     # *target vector from test-set*

**if** **len**(vt[vt**>=**0]) **>** 0:

        \_,h = **rbm.sample\_h**(v)     # *sampling hidden nodes*

        \_,v = **rbm.sample\_v**(h)     # *sampling visible nodes*

        # *Error rate*

        test\_loss += **torch.mean**(**torch.abs**(vt[vt **>=** 0] - v[vt **>=** 0]))

        cnt += 1.0

    # *creating list of original & predicted ratings*

    original\_test\_set\_ratings = **vt.numpy**()

**test\_set\_ratings\_list.append**(original\_test\_set\_ratings)

    predicted\_ratings = **v.numpy**()

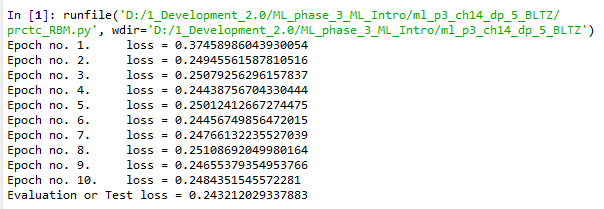
**pridicted\_rating\_list.append**(predicted\_ratings)

eval\_loSS = test\_loss/cnt

**print**(f"Evaluation or Test loss = {eval\_loSS}")

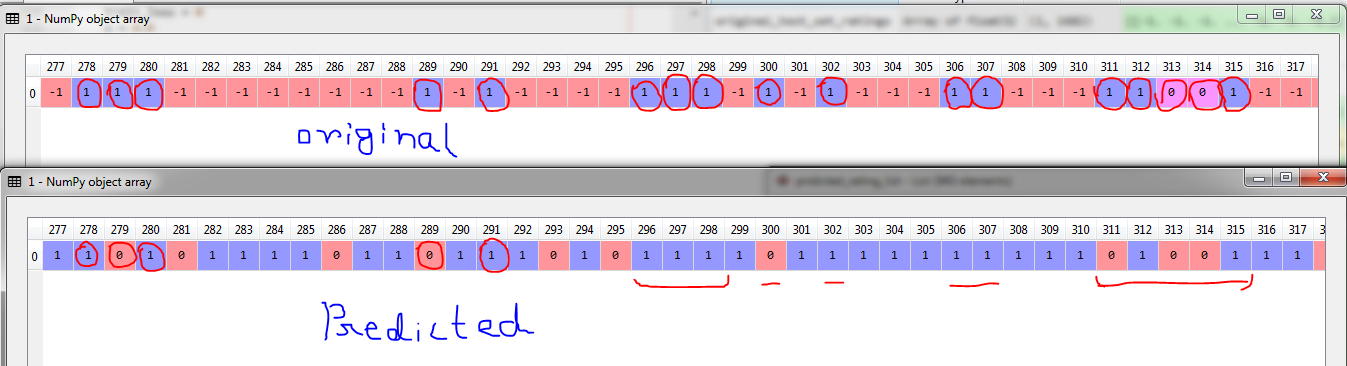
# *python prctc\_RBM.py*

Result



We can see that in ***test\_set*** prediction we have ***Test-Loss 0.2432***, i.e. more than ***75%*** *correct predictions*.

***Comparison between predicted and real results***



* From this comparison we can see that in the *predicted* result from *RBM*, we have some *ratings* that are *not originally rated*. Also we get *75% correct predictions* for the *movies* that are *originally rated* (marked red).