Chapter 15 : Part 3

**Deep Learning**

**AutoEncoder : project**

**15.3.1 Data Preprocessing**

* We will *train* our *auto-encoders- AE* on this *training set*, and so it will try to identify some *patterns* to find some *groups* *of* *movies* that are liked by *similar* *segments* *of users*,
* *AE* will find these *patterns*, it will find some *specific features* of the *movies* which will be the *hidden nodes* in the *auto-encoders*, and these ***specific features*** can be some *genres*, some *actors* that are in the *movies*, or some *directors*.
* For example, *one hidden node* can be a *specific director*, like *Tarantino*, for example, or it can identify *some movies* with a *great actor* that is *liked by the same group of people* because they gave the *same high rating*.
* So basically it will identify some patterns, it will identify some features, based on which, in the future, the AE-recommender-model will be able to *predict* the *rating* of a movie that *one user hasn't seen yet*, and it will be able to *predict* that *rating* based on the *features* that the *auto-encoder's* *detected*, and based on the *history* of this *user*.
* The AE will take the features that it detected, and it will take also the *ratings* of that one same user to *predict* the *rating* of the *new movie* that the same *user hasn't seen*, and therefore has *not rated yet*.
* There is *no common rating* of the *same movie* by the *same user* between the training sets and the test set, however, we have the *same* *users*, for example in train-set we start with *user one*, as in the *training set*. But, for this same *user one*, we won't have the *same movies* because the *ratings* are *different in the test-set*.
* It is not a supervised technique: *Test set* is just to get the *real results* on one's side, and so what will happen is that we will make our *predictions* using *our auto-encoder's model* to predict the rating, for example, *AE* could *predict* rating ***4*** of ***movie 14*** by ***user-1***, but in test-set the real rating is ***5***.
* Then, we will measure the *error* between the *real rating, 5*, and the *prediction, 4*, and that will allow us to measure the MSE that is the Mean Squared Error.
* We are going to *convert* our *training sets* and *test set* into *2 matrices*, where the ***lines/rows*** are going to be the ***users***, the ***columns*** are going to be the ***movies***, and the ***cells*** are going to be the ***ratings***.
* In each of these *two* *matrices*, we want to include *all the users* and *all the movies* from the *original data set*.
* And if in the *training set*, a *user* *didn't rate* a *movie*, well we'll *put* a *zero* into the *cell* of the *matrix* that corresponds to this *user* and *that* *movie*.
* These *matrices* will have the *same number of users* and the *same number of movies* so they will have the *same number of lines/rows* and the *same number of* *columns*.
* And in these *two matrices*, each ***cell*** of indexes ***U, I***, where ***U*** is the ***user*** and ***I*** is the ***movie***, each ***cell UI*** will get a ***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***.

For this reason we get first *total number of users* and *total number of movies*.

* We need to make those *matrices* for a *specific structure of data* that will fit to the *structure* of the *AE* and the *AEs* are like *neural* *networks*, where you have some *input nodes* that are the *features* and you have some *observations* going one by one into the *NN* starting with the *input nodes*.
* So we created 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* (exactly same as RBM-model).
* We use *tensors* instead of *NumPy array*. A *tensor* is a *multidimensional matrix*, but instead of being a NumPy array, this is a PyTorch array. And in fact, we could build a neural network with *NumPy* *arrays*, but that would be much *less efficient* and that's why we're using *tensors*, as what we're about to do with the torch.tensors.
* So, the ***training\_set*** is going to be one ***torch.tensor*** and the ***test\_set*** is going to be another ***torch.tensor***. Two separate multidimensional matrices based on PyTorch.

|  |
| --- |
| * Note that, with ***TensorFlow***, we work with *tensors*, that are *TensorFlow Tensors*. * Those are another kind of *tensor*, another kind of *multidimensional* *matrix*. * We could also *implement* our *AE* from *scratch* with *TensorFlow*. But it turned out that for *AE*, *PyTorch* give *better results*, and also it is much more simple. |

* We copy all code from our RBM-recommender. We use the code *before* the *binary-rating conversion*. Because in our AE-model we actually predict ratings from ***1*** to ***5***.

# *----------- AE : 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 -----------*

# *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)

* Next, we create the ***architecture*** of our ***NN***, that is our ***AE-class***. We'll make an AutoEncoder-class. And we'll use this *class* to build our *AE-recommender-model*. Inside this class we'll define 2 functions, one is ***\_\_init\_\_*** and other is ***forward()***.
* We can use it afterwards to change the architecture of the AE to try other AE-architectures.

**15.3.2 AE-architecture : StackedAutoEncoders class - \_\_init\_\_()**

The class that we're going to make will be the model that will contain the instructions on how to build the auto-encoder.

* We're making this class, because to make an auto-encoder, we need to define multiple things:
* How many layers we want to have,
* How many nodes in the layers,
* We also need an activation function,
* A criterion, and
* An optimizer function.
* To make an auto-encoder, we not only need some *variables*, that's will get, for example, the *info of the layers*, and *some functions*, for the *activation* and the *optimizer*. And to get all this in one same recipe, well, we can only use a *class*, or, at least, that's the *simple* *solution*.
* We could also a *module* contains *several classes*,
* Or even some *libraries*, which contain *several modules*.
* But the *simplest solution* is to make a *class*. And a *simple function wouldn't be enough*.
* So the *first reason* to make this class is: a ***class*** can gather all these features, variables and functions to make the ***auto-encoder***.
* But then there is a *second reason*, and this is very important, because this is related to ***PyTorch***.
* Inheritance: We're going to create a class that's we're gonna call ***StackedAutoEncoders*** (SAE in short).
* That is actually going to be the child class of an existing ***parent*** class in ***PyTorch***. This parent class is called ***Module***.
* And this *class* ***Module*** is taken from the ***nn*** *module* that we imported here: ***torch.nn as nn***.
* We are doing this so that we can use ***all*** the ***variables*** and ***functions*** from the parent class ***Module***, and that's what *inheritance* is all about.
* Because this parent class *Module* contains all the tools to make an *auto-encoder*. Basically, it contains everything we need to make an *auto-encoder*.
* It contains an *optimizer* function,
* It contains a *criterion*,
* It contains tools to make *full connections* between the layers.
* Our auto-encoder is a stacked auto-encoder, because we will have *several hidden layers*, so we will have *several* *encodings* of the *input vector* *features*.
* We're gonna call this child class ***StackedAutoEncoders***, with capital S-A-E, because we use capitals for classes.
* In the parentheses of ***StackedAutoEncoders()***, we're going to input the parent class which is Module.

**class** StackedAutoEncoders(nn.Module):

* ***\_\_init\_\_():*** We now define the ***\_\_init\_\_()*** function because we need it to *initialize* the *objects* that are created from this SAE-class. We define it as:

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

* We add a comma, and then just nothing **(self, ):**, because this will just consider the variables of the ***Module*** class, because we are doing inheritance (it is something like *variable arguments*).
* Now using the ***super*** function we get the inherited methods from the ***Module*** class.

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

**super**(StackedAutoEncoders, self).**\_\_init\_\_**()

**super**(StackedAutoEncoders, self).**\_\_init\_\_**() will make sure we get all the inherited classes and methods of the parent class ***nn.Module***.

* We're gonna use the ***nn.Linear()*** class, to make the different ***full connections*** between the ***layers***.
* Full Connection - ENCODING (first hidden layer): Now we start creating the *architecture* of the *neural network*, by choosing the *number of layers* and the *hidden neurons* in each of the *hidden layers*.
* The first part of the *neural network* that is the *full connection* between the *input vector* of features (ratings of all the movies for one specific user) and *first hidden layer*.
* The *first hidden* *layer* in *AutoEncoders* is a *shorter vector* than the *input vector*, that's the technique of *auto-encoders*.
* We're encoding the *input vector* into a *shorter vector*. That's will take place in the *first hidden layer*.
* Now we're going to create an *object* of the *class* that is inherited from the ***nn*** module. This object will represent the *full connection* between this *first input vector features* and the *first encoded vector*.
* We're gonna call this first full connection ***fc1***, and it is associated to our object so we used ***self.fc1***,
* Then we use the inherited class ***Linear()*** from the ***nn*** module. So, ***self.fc1 = nn. Linear()***

self.fc1 = **nn.Linear**(nb\_movies, 20)

* Parameters of ***nn.Linear()***:

1. The first input is the number of features in the input vector, i.e. number of movies ***nb\_movies***. Since the *features* are actually *movies*, because *one observation (one-user)* contains all the *ratings* of all the *movies*.
2. The second input is going to be the *number of nodes/neurons* in the *first hidden layer*. i.e the *number of elements* in the *first encoded* *vector*.

* We tried several values for this vector. The value we're gonna use is based on experiments, and the value is 20. But this is still not tuned. To get a better score we can tune it later.
* *20 nodes* in *first hidden layer* led us to a pretty good score. It means that our first *encoded vector* is a vector of *20 elements*.

|  |
| --- |
| * If we want to make the parallel with the problem, well, what would represent this first hidden layer composed of 20 neurons? * Remember that the *neurons* in the *auto-encoder* represent actually *some features* that, through *unsupervised learning*, the auto-encoder detects. * So, these *20 features* that are in the *first hidden layer* can represent some *features of movies* that are liked by *similar people*. * For example, one of these *20 nodes* could be a *specific* ***genre*** of a movie. One of the detected feature could be the ***horror*** movie ***genre***. * In that case, when a *new user* comes in the *system* then, if this new user gave some *good ratings* to *all* the *horror movies* of the *database*, then this will activate this ***horror genre neuron/node*** and therefore a big weight will be attributed to this neuron/node in the *final prediction* to predict the *rating* of a *horror movie*.   Of course this is a very simple example and of course, the features can be much more complex that even would be difficult to define, but *that's how it works*.   * With this *number 20* here, we're trying to detect *20* *features*. |

* Second full connection - ENCODING (Second layer): Since we're making some *stacked auto-encoders*, we're gonna make *another layer*. We have to make a *second full connection*.

self.fc2 = **nn.Linear**(20, 10)

* It is same as previous layer, we just need to give this layer a name "**fc2**" i.e. full-connection-2.
* Again we're gonna use the ***Linear*** class that will make this *full connection*.
* To make the full connection between the *first hidden layer*, composed of the *20 neurons* and this second hidden layer, then we need to specify the number of *nodes of previous layer* (which is 20) and since we are still *encoding*, we choose *smaller* *number* of nodes for this *layer*, by experiment this number is ***10***.
* In this *second hidden layer* with *10 neurons*. it will *detect* even more *features*, *based* on the *previous features* that were detected.
* Third full connection - DECODING (3rd layer): Since we're doing deep learning, let's add a third layer. But now, we start DECODING.

self.fc3 = **nn.Linear**(10, 20)

* We name this third hidden layer ***fc3***, the third *full connection* between the *second hidden layer* and the *third hidden layer*.
* Since we are now *Decoding*, the number of *nodes* will be *higher* than the *previous layer*. We specify ***10*** *neurons* of *previous* layer and ***20*** for this *third layer*.
* Here we're just starting to ***reconstruct*** the ***original input vector***. So, let's make things *symmetrical*.
* Fourth full connection - DECODING (4th layer): This is the last part of the ***decoding***, the last full connection we have to make. We name the *fourth full connection* as ***fc4***. Here we input ***20***, because we had ***20*** ***nodes*** in the third layer.

self.fc4 = **nn.Linear**(20, nb\_movies)

* Now this layer is the *last layer*, and the number of *neurons* in the *output layer* must be *same* as *input layer*, because, in *auto-encoders*, we are *re-constructing* the *input vector*.
* Hence the *output vector* should have the *same dimension* as the *input vector*.
* Therefore, the *number of neurons* in the *output layer* is also going to be ***nb\_movies***.
* Activation function: Now we have to specify an *activation* function that will, *activate* *the neurons* when the *observation* *enters* into the *network*.
* For example, if someone gives some *good ratings* for *horror movies*, well, this will *activate* the *specific feature* for the *horror* *movie* *genre*. And this activation will be done with a certain function,
* There are *different activation functions*, we tried the ***Rectifier Activation*** function and also, the ***Sigmoid Activation*** function, and it turns out that we got better results with a Sigmoid Activation function between the *four* *full* *connections*.

self.activation = **nn.Sigmoid**()

* So, to get the sigmoid activation function, we use ***nn.Sigmoid()***, from the ***nn*** module, we don't need any arguments for now.
* Later we can try some combinations of the *Rectifier Activation* function and the *Sigmoid Activation* function, to get better result.

**class** StackedAutoEncoders(nn.Module):

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

**super**(StackedAutoEncoders, self).**\_\_init\_\_**()

        self.fc1 = **nn.Linear**(nb\_movies, 20)

        self.fc2 = **nn.Linear**(20, 10)

        self.fc3 = **nn.Linear**(10, 20)

        self.fc4 = **nn.Linear**(20, nb\_movies)

        self.activation = **nn.Sigmoid**()

* So that's all for the **\_\_init\_\_()** function. That's the *architecture* of our *stacked auto-encoder*. You're totally welcome to try to change it.
* Technically speaking, we just created *five objects* of twodifferentclasses, *four* *objects* of the ***Linear*** class, and *one* *object* of this ***Sigmoid*** class.

**15.3.3 AE-architecture : StackedAutoEncoders class - forward()**

Now, we're gonna make another function that is required to build our auto-encoders. This second function going to make the action *that takes place* in *auto-encoder*. This action is basically the *encoding*, and the *decoding*.

* We're going to name this function ***forward()***. And that will proceed to the different ENCODINGS and DECODINGS when the observation is forwarded into the network. We spell it correctly because PyTorch will use it (so ***foRWard()*** may not work it has to be ***forward()*** ).
* It will not only do the action of encoding and decoding, but also will apply to different activation functions inside the full connections.
* Also, the main purpose of making this function is that it will *return* in the end the *vector of predicted ratings* (output-vector) that we will *compare* to the *vector of real ratings* (input-vector).
* We call this function ***forward***, like *forward propagation*, because during the forward propagation the *encoding* and *decoding* take place.
* Arguments: In this function, we have to input two arguments, ***self*** and ***x*** *(x is our input vector: the features with all the ratings for the movies in one specific user)*,
* then we'll ***transform*** this input vector ***x***, by *encoding* it *twice* and then *decoding* it *twice* again to get the *final output vector* i.e. the decoded vector that was reconstructed.

**def** **forward**(self, x):

* First-Encoding: First, we are about to encode our input vector ***features x*** into a first *shorter vector* composed of *20 elements* in our *first hidden layer*.

**def** **forward**(self, x):

        x = self**.activation**(self**.fc1**(x))

* we're going to take our AE-object (i.e. ***self***) and then we'll use our ***activation*** object that we created using the ***Sigmoid*** class, because this *sigmoid activation function* will activate the *neurons* of this first *encoded vector of 20 elements*.
* The *parameter* of this *activation* function will not be directly the input vector **x**. We've used ***fc1(x)***, because we are *encoding*, and to do the *encoding* we have to apply the *activation* function on the *first full connection* **fc1**. Hence to include that information of the *first full connection* we don't input **x** *directly*.
* Since ***fc1*** is an object of the ***Linear*** class it will apply the full-connection over ***x***. And it becomes the first-full-connection.
* Since ***forward()*** will return the *output vector* in the end using *several encoding-decoding*, and we'll *compare* it to the *real rating*.
* For this reason, we're going to *modify* ***x*** after each *encoding* or *decoding*, hence we used the variable ***x*** again so that it will be *modified* by the *activation* function.
* The *first encoded vector* in the first hidden layer will be the new modified ***x***.

x = self**.activation**(self**.fc1**(x))

* This ***x*** in self**.fc1**(x) is the input vector features of the *first full connection*
* and this ***x*** in "x = self**.activation**(" will be the *new first encoded vector* resulting from this *first encoding* that happens here with the *activation function* in the *first full connection*.

Now basically we need to do the *same* for the *other* *full connections* and so this is going to be *exactly* the *same*.

* Second-Encoding: We *update* ***x*** of the *first hidden layer*, then on this vector in the first hidden layer, we made the *second full connection* which will *encode* this vector of ***20*** elements into a *shorter vector* of ***10*** elements.

x = self**.activation**(self**.fc2**(x))

* At the same time, we apply the *sigmoid* activation function to *activate* the *neurons* and then eventually ***x*** becomes this *new encoded* vector of ***10*** elements in this *second hidden layer*.
* first-Decoding: We do the same for the third full connection represented by ***fc3***. Now we are DECODING.

x = self**.activation**(self**.fc3**(x))

* The third full connection ***fc3*** corresponds to the decoding from an *input vector* of ***10*** *elements* in the second hidden layer to a *larger output vector* *composed of* ***20*** *elements*.
* Final Decoding: To get our final output vector we use ***fc4*** for 2nd (final) DECODING. It will be our reconstructed output vector, similar to input vector.

x = self**.fc4**(x)

* But we *don't apply* the *activation* function because this is the final part of the decoding.
* We only have to use our fourth full connection, ***fc4*** *without* the *activation* function.
* From the ***forward()*** we ***return*** the reconstructed output ***x***. Then ***x*** is our vector of *predicted ratings*.

**def** **forward**(self, x):

        x = self**.activation**(self**.fc1**(x))

        x = self**.activation**(self**.fc2**(x))

        x = self**.activation**(self**.fc3**(x))

        x = self**.fc4**(x)

**return** x

* Finally ***StackedAutoEncoders*** class is done. It was composed of two functions,

1. the ***\_\_init\_\_()*** defines the *architecture* of our *auto encoders* and
2. the ***forward()*** does the action of establishing different full connections by applying at the *activation functions* to *activate* the *right neurons* in the network.

* In ***forward()***, we did *two encodings* and then *two decodings* to get our reconstructed output vector in the output layer.

Next we'll compare output to the real ratings, to measure the loss so that we can then obtain the weight to *reduce* this *loss*.

**15.3.4 SAE model : criterion & optimizer for SAE architecture**

Now we create an object of ***StackedAutoEncoders*** class. We name this object "***sae***". Since we didn't specify any arguments in the ***\_\_init\_\_()*** function of this ***StackedAutoEncoders*** class, we don't have to input any arguments here.

sae = **StackedAutoEncoders**()

* Criterion: Which we'll need this criterion to train the model.

criterion = **nn.MSELoss**()

* It's basically the *criterion* for the *loss function*, and the *loss function* is going to be the *mean squared error* so we used **nn.MSELoss**() class from ***nn*** module.
* Optimizer: Now the last thing that we need is an *optimizer*. It is much more like in *Keras*. The optimizer will apply *Stochastic* *Gradient Decent* to update the *different weights* in order to reduce the *error* at each *epoch*.
* We'll use ***torch.optim*** module to import our optimizer. We can use ADAM-optimizer or RMSProp-optimizer. By experiment RMSProp gives better result.

optimizer = **optim.RMSprop**(sae.parameters(), lr = 0.01, weight\_decay= 0.5)

* Arguments: We will actually input three things.

1. We have to input all the parameters of our ***AutoEncoders***, that is the parameters to build our ***StackedAutoEncoders*** architecture.

* We don't have to rewrite all that again. The easy way is to use ***sae.parameters()*** that gets all the parameters from our ***sae*** object.
* **lr**: Is the learning rate. It depends on experiments, a good value I found was ***0.01***. Of course we can tune it to get better result.
* **weight\_decay**: Specifies the decay. Decay is used to *reduce* the ***learning rate*** after every few *epochs* and that's in order to *regulate* the *convergence*. We can also tune it. Based on my experimenting a good value I found was ***0.5***.
* Note: We apply ***parameters*** attribute on ***sae*** object , not the class ***StackedAutoEncoders***.

# *-------- Creating the architecture of the Neural Netwark for SAE --------*

**class** StackedAutoEncoders(nn.Module):

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

**super**(StackedAutoEncoders, self).**\_\_init\_\_**()

        self.fc1 = **nn.Linear**(nb\_movies, 20)

        self.fc2 = **nn.Linear**(20, 10)

        self.fc3 = **nn.Linear**(10, 20)

        self.fc4 = **nn.Linear**(20, nb\_movies)

        self.activation = **nn.Sigmoid**()

**def** **forward**(self, x):

        x = self**.activation**(self**.fc1**(x))

        x = self**.activation**(self**.fc2**(x))

        x = self**.activation**(self**.fc3**(x))

        x = self**.fc4**(x)

**return** x

sae = **StackedAutoEncoders**()

criterion = **nn.MSELoss**()

optimizer = **optim.RMSprop**(sae.parameters(), lr = 0.01, weight\_decay= 0.5)

* Now that’s the end of the *architecture* of our *neural networks*. Next we're gonna implement the training,

**15.3.5 Train the SAE model : Nested-for-loop & input**

Our code will be bit technical to optimize our code so that you can use it on high dimensional datasets. Here we'll use a dataset with 200,000 ratings. If you want to use this code for the 1 million ratings dataset or even a larger dataset, you need an optimized code that saves up the memory as much as possible.

|  |
| --- |
| * We won't go in too much details about the technique and besides this is some technique related to PyTorch. The most important thing that matters is that you understand * How the architecture of the Neural Network works. * How to manipulate our ***StackedAutoEncoders*** architecture by changing the *number of layers* and the *number of neurons* in the *hidden layers* in the defined StackedAutoEncoders class. * You can try *different* *combinations* of the *activation functions* by manipulating the *objects* related to the *activation functions*. |

* We're gonna use some pretty *advanced techniques* in PyTorch.
* No. of Epochs: The first step is to define a number of EPOCH, it is based on experimenting. We're gonna train our StackedAutoEncoders on ***200*** EPOCHs.

nb\_epoch = 200

* Outer-loop, train\_loss: Second step is to make a ***nested-for-loop***, outer for loop runs over the epochs and the *inner for-loop* will loop over all our *observations* in each EPOCH (i.e. all our users because each observation corresponds to the ratings of a user).
* Before we introduce the second loop, we need to do is initialize ***train\_loss*** and counter ***s***. (As we did in RBM. We also will make the ***test\_loss*** for the loss of the ***test set***.)
* Counter s: Counter ***s*** will count the number of users that *rated* at least *one movie*. The purpose is to optimize the memory, we won't do the computations for the *users* who *didn't rate any movies*.
* To do this we need to keep track of the number of users who *rated at least one movie*. Hence, we use **s**. We use ***0.*** or ***0.0*** to make is *floating-point-data-type*.
* Since we're gonna use s to compute the RMSE in the end, therefore floating-point is used to make sure that int-type and float-type do not cause any issue.
* Since the *root-mean-squared error* is a *float*, then all the *elements* to compute the root-mean-squared error should be a *float*. It's not compulsory, but that's just to avoid a warning.
* The next step is to start the *second loop*. It will loop over *all the observations*, i.e *all the users*. So this inner-for loop will *introduce* all the *actions* that will take *place* in a single *EPOCH*.
* Inside 2nd for loop: We're gonna get our predicted rating using our ***StackedAutoEncoders*** class with ***sae*** object that we created earlier.
* We're gonna compute the *loss error* on one EPOCH. To observe how the *training evolving*. Because we want to *optimize* this *loss*, so we keep track if it's *decreasing* over the EPOCHs.
* We will apply our optimizer **optim.RMSprop** to apply Stochastic Gradient Descent to *update* the *weights* and lead to the *convergence*.
* The loop variable we call ***id\_user***. We set the ***range(nb\_users)***, without upper/lower bound, it just *loop for-each user*.
* Because actually it needs to be the *indexes of the observations* of our *training set* and the indexes of the observations of our training set don't go from ***1*** to ***943***. They go from ***0*** to ***942***. ***range(nb\_users)*** will make this ***range*** from ***0*** to ***942***.

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

* Now we apply the *input vector*, that is, the *input vector* of *features* that contains *all* the *ratings* of all the *movies* given by this *particular user* inside the loop.

input = **Variable**(train\_set\_tensor[id\_user]).**unsqueeze**(0)

* ***train\_set\_tensor[id\_user]*** will get the ratings for all movies that are given by ***id\_user***. ***id\_user*** is the user with which we are dealing right now in the loop.
* But ***train\_set\_tensor[id\_user]*** is a vector and a *network* in *PyTorch* or even on ***Keras*** can not accept a single vector of one dimension they accepts a Batch of input vectors.

For example, our ***forward()*** function will not take a ***simple vectors*** of ***one dimension*** as ***input***. So we need to add an extra fake dimension which will correspond to a ***batch*** (we also did it in RBM, but in different way).

* Also remember, we did it already in CNN, when we used the *predict* function to *predict* if the *image* contains a *cat* or a *dog*. we had to add *one dimension* and that *additional dimension* was *for* the *batch*. We put our input image of a *cat* or a *dog* into a *batch* and that added *one dimension* that then would make the whole thing accepted by the predict method.
* So we'll create a *batch*. This *batch* will *contain* *one vector*, but we will be into a *new dimension* corresponding to the *batches*.
* We now have to use the ***Variable*** *module* that we imported at the beginning from ***torch.autograd***. It's just a **PyTorch** technique.

1. We put ***train\_set\_tensor[id\_user]*** inside ***Variable()***.
2. To create this additional dimension, we use the function ***unsqueeze()*** on ***Variable()***, like this.
3. To specify the *index* of this *new dimension*. We use ***unsqueeze(0)***. Because we are gonna put this new dimension in first position (as we did in Keras). And so this dimension will have ***index zero***.

This will create a batch of a single input vector.

# *---- Training the SAE model ----*

nb\_epoch = 200

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

    train\_loss = 0

    s = 0.0

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

        input = **Variable**(train\_set\_tensor[id\_user]).**unsqueeze**(0)

|  |
| --- |
| * Note: The *batch* can have *several input vectors*. Remember we called this Batch Learning. * But *here* we're gonna do Online Learning. That means that *we're going to update* the *weights* after *each observation* going to the *network*. And therefore, we are creating a *batch* of *one input vector*. But we have to create this batch, otherwise it won't work. |

**15.3.6 Train the SAE model : target-vector**

* Target-vector: As we did in *RBM*, we need a *target vector*, which is a *copy* of the *input vector*. Since we're going to *modify* the *input*, and *target* remain the *same*, so that we can *compare* the *modified input* and the original data in *target*.
* We use ***clone()*** function to make the copy of original-input to ***target***.
* Filter the user: In this step we're gonna introduce an ***if*** ***condition*** to find the *users* who rated *at least one movie*. So if an observation *contains only zeros*, which means that the user *didn't rate* any *movies*, then we *won't care* of this *observation*.

# *---- Training the SAE model ----*

nb\_epoch = 200

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

    train\_loss = 0

    s = 0.0

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

        input = **Variable**(train\_set\_tensor[id\_user]).**unsqueeze**(0)

        target = **input.clone**()

**if** **torch.sum**(target.data **>** 0) **>** 0:

            output = **sae**(input)

* Here ***target.data*** will take all the values of ***target***, ***target.data*** will be all the ratings of this user here at the loop right now, it's just all the ratings.
* ***target.data > 0***, consider all the ratings that are *larger than zero*. We wanna check if **torch.sum**(target.data **>** 0) is larger than zero.
* And if that's the case, **torch.sum**(target.data **>** 0) **>** 0 means that the observation contains *at least one rating* that is *not* *zero*. In that way we consider the users that *rated* *at* *least one movie*.
* Output: To get our vector of predicted ratings, i.e. our output at the very right of the network. We're gonna introduce a new variable named ***output***.
* We have to use our ***sae*** object. Because this object is an object of the ***StackedAutoEncoders*** class. In this ***StackedAutoEncoders*** class, the action of *forwarding* the *input vector* into the *network* takes place.
* ***forward()*** function returns the output of the network, i.e. the *vector* of *predicted ratings*. So since this ***forward()*** function is part of the ***StackedAutoEncoders*** class, and we did use *variable-arguments* in **def** **\_\_init\_\_**(self, ):, we can use a parameter with ***sae*** object i.e ***sae(x)***, this ***x*** will then passed to ***forward(x)***, and ***forward()*** function will be applied and will return modified ***x***.
* In the ***forward()*** function *encodings* and *decoding* will take place with the *input*. And this will return eventually to the ***vector*** of *predicted ratings*.

output = **sae**(input)

* For simplicity we rewrite the if condition:

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

    train\_loss = 0

    s = 0.0

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

        input = **Variable**(train\_set\_tensor[id\_user]).**unsqueeze**(0)

        target = **input.clone**()

        non\_zero\_ratings = **torch.sum**(target.data **>** 0)

**if** non\_zero\_ratings **>** 0:

            output = **sae**(input)

**15.3.7 Train the SAE model : loss & mean\_corrector**

Now we have our Real Ratings in ***target*** and our Predicted Ratings in ***output***. In this step we optimize the memory and the computations.

* When we apply *Stochastic Gradient Descent*, we make sure that the gradient is computed only with respect to the ***input*** and not the ***target-vector***.

        non\_zero\_ratings = **torch.sum**(target.data **>** 0)

**if** non\_zero\_ratings **>** 0:

            output = **sae**(input)

            target.require\_grad = **False**

            output[target **==** 0] = 0

            loss = **criterion**(output, target)

            mean\_corrector = nb\_movies/**float**(non\_zero\_ratings + 1e-10)

* **"target.require\_grad = False"** Here ***require\_grad*** will make sure that we *don't compute* the *gradient* with respect to the ***target*** and that will save a lot of computations and that optimizes the code.
* **"output[target == 0] = 0"** Considers *only non-zero values*. We don't wanna deal with the *movies* that the user didn't rate, where the *ratings* are *equal to zero*, but that is only for the ***output*** vector.
* We take the values of our output vector, those ratings that are ***0*** in the ***target***-vector, and we reset those to 0.
* We're just taking the same indexes of the ***ratings = 0*** in the ***target*** vector (original-rating). And for these indexes of the output vector, we will set the values corresponding to these indexes to zero. (Notice, we did it in previous chapter for our RBM model ).

[The ratings which were 0 at the start are changed after encoding & decoding. Se here we are resetting them to 0 by using ***target-vector***.]

* The reason is, we don’t want to *update* the *ratings* that are *initially* ***0***, not-rated by the user. So that these they *won't have impact* on the updates of the different weights right *after* *measuring* the *error*.
* After we've measured the ***error***, the ***weights*** will be ***updated*** by the ***RMSprop Optimizer*** and updating these ***weights*** require some computations and in these computations, these ***0****-ratings* here don't count.
* So, after updating, even if they're *not equal to* zero, they not being *counted*, and so, to save up *some memory*, again, we set them to ***zero***.
* **"loss = criterion(output, target) "** Computes the loss error using our ***criterion*** object. We inputted two arguments ***target*** - the vector of real ratings and ***output*** - the vector of predicted ratings. *First argument* is ***output***, *second argument* is ***target***.
* **"mean\_corrector = nb\_movies/float(non\_zero\_ratings + 1e-10) "** It's just a ratio where ***nb\_movies*** as numerator and ***non\_zero\_ratings*** as denominator.
* Notice we added a small number **1e-10** to the *denominator* to avoid *"undefined"* mathematical error, so that *denominator* always be a non-zero number. Because ***non\_zero\_ratings*** could be ***0*** in case the *user not-rated any movies*.
* ***non\_zero\_ratings*** means we're considering all the movies that have *non-zero ratings* by the *current user* in the *loop*.

|  |
| --- |
| * Why do we need to create this ***mean\_corrector***? * This actually represents the *average of the error*, but by only considering the *movies* that were *rated*. * We need to do this *because* we only considered here the *movies* that have *non-zero ratings*. When we will compute *mean*, this *mean* has to be *computed* only on the *movies* that we consider. That is, the *movies* that got *non-zero ratings*.   This ***mean\_corrector*** variable is just to adapt to this consideration of the movies that got non-zero ratings. We need to do this because this will then be mathematically *relevant* to compute the *mean* of the *errors*. |

**15.3.8 Train the SAE model : backward, train-loss, train**

* **backward():** Now we're gonna call the ***backward()*** method for the loss. This method comes from ***criterion*** object. By calling the ***backward*** method we just tell in *which direction* we need to *update* the *different weights*.
* To specify *increase* or *decrease* the *weight*. We still inside the if-condition:

**loss.backward**()

* train\_loss: We now compute the RMSE to update the ***train\_loss***.
* ***loss.data[0]*** gonna get the part of this loss object that contains the error. We excess to the ***data*** in the ***loss*** object and then we need to take the *index* of the *data* that contains this *train loss* which is ***0***.

            # *train\_loss += np.sqrt(loss.data[0] \* mean\_corrector)*

            train\_loss += **np.sqrt**(loss.data \* mean\_corrector)

* However in newer version of PyTorch we use ***loss.data*** without index. To avoid runtime error:

IndexError: invalid index of a 0-dim tensor. Use `tensor.item()` in Python or `tensor.item<T>()` in C++ to convert a 0-dim tensor to a number

That's because in PyTorch>=0.5, the index of 0-dim tensor is invalid. The master branch is designed for PyTorch 0.4.1, loss\_val.data[0] works well.

Try to change

total\_loss += loss\_val.data[0]

loss\_values = [v.data[0] **for** v **in** losses]

to

total\_loss += loss\_val.data

loss\_values = [v.data **for** v **in** losses]

might fix the problem.

* We multiply ***loss.data*** with our adjustment factor ***mean\_corrector***. We're just adjusting ***loss*** with ***mean\_corrector*** factor to compute the relevant *mean*.
* Since this ***loss.data*** is the Squared error and we want to get the Square-Root of this error. i.e one degree loss, we will take the root of this loss.data.mean\_corrector.

**np.sqrt**(loss.data \* mean\_corrector)

* We also *increment* the *counter* ***s*** here that corresponds to the *number of users* who rated at *least one movie* (***s*** just filtering the users who gives rating from the total users.). We used 1.0 because we want ***s*** to be a float-number.

s += 1.0

* Now we apply the ***optimizer*** that we defined before, which is **optim.RMSprop**(). We apply it here to update the weight.
* We need to use ***step()*** method of the ***optimizer*** object from the ***RMSprop*** to apply the *optimizer* to update the *weights*.

**optimizer.step**()

* ***optimizer.step()*** was the last step of both the ***if*** condition and the inner ***for loop***. so we're done dealing with our observation, taking care of all the actions that happened into the network.

|  |
| --- |
| * The difference between backward and optimizer: * ***backward()*** decides the direction to which-way the *weight* will be *updated* (i.e. increased or decreased). * And **optimizer.step**() decides intensity of the *updates*. That is, the amount by which the *weights* will be *updated*.   So, ***backward()*** decides the direction of weight-update. and **optimizer.step**() decides the intensity/amount weight-update. |

* Print: Finally, we print the ***epoch*** and ***train loss***, inside the *outer* ***for-loop***. We first calculate the *normalized train-loss* by dividing ***train\_loss*** by ***s***.

    train\_loss\_normalized = train\_loss/s

**print**(f"Epoch : {epoch}, Loss = {train\_loss\_normalized}")

* Our model is now ready to train!!! If you wanna build a different AutoEncoder model, just *change the architecture* in the ***StackedAutoEncoders*** class ***definition***. You can try some other *combinations* of *activation functions* there.
* To tune the model, also try different number of *epochs*, with different *activation functions*.

# *---- Training the SAE model ----*

nb\_epoch = 200

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

    train\_loss = 0

    s = 0.0

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

        input = **Variable**(train\_set\_tensor[id\_user]).**unsqueeze**(0)

        target = **input.clone**()

        non\_zero\_ratings = **torch.sum**(target.data **>** 0)

**if** non\_zero\_ratings **>** 0:

            output = **sae**(input)

            target.require\_grad = **False**

            output[target **==** 0] = 0

            loss = **criterion**(output, target)

            mean\_corrector = nb\_movies/**float**(non\_zero\_ratings + 1e-10)

**loss.backward**()

            train\_loss += **np.sqrt**((loss.data)\*mean\_corrector)

            s += 1.0

**optimizer.step**()

    train\_loss\_normalized = train\_loss/s

**print**(f"Epoch : {epoch}, Loss = {train\_loss\_normalized}")

* Before we execute our code, let's set some expectations of what we would like to get.
* First thing is, the ***loss*** represents the *average of the differences* between the *real rating* and the *predicted rating*, on the *training* *set*.
* Which means that, for example, if we get a ***loss*** of ***1*** at the last epoch, that will mean that the *average difference* between the *real ratings* of the *movies* by the *users* and the *predicted ratings* will be one.
* And that's not too bad because it means: when we predict if a user is going to like a movie, on average we will make an error of 1 star out of 5.
* We were hoping to get a loss that would be ***less*** than ***one star*** so that the average difference between the *real rating* and the *predicted rating* is *less than* ***1***. Therefore, on average, we will make better predictions of whether a user is going to like a movie or not.
* We are at least expecting to get a ***loss*** of less than ***1*** *(loss < 1 star)*,
* Second thing is we're now calculating the *loss* on the *training sets*, also we need to calculate the *test\_loss* on the *test set*. Then we can see is there a *high over-fitting* or not.
* Result: at the *last epoch*, we might expect a *final* *loss* of ***0.91***,
* So that's not *too bad*. Besides, we were *training* on *100,000* ratings and you will definitely get a *better* *loss error* if you train this on more ratings. For example *1 – million dataset*. Our code is *optimized* so that the training doesn't take *too much time*.
* Now let's hope that we will get around the same error on the test set.
* Increasing epochs may lead to Overfitting: Of course, by having *more epochs* you can get a *lower loss* on the *training set*, but then you might end up with a *larger difference* of the *loss* between the *training set* and the *test set* which leads to *high-overfitting*, and we want to avoid that, so ***200*** epoch is enough for our current dataset.

|  |
| --- |
| * **NotImplementedError:** Module [**StackedAutoEncoders**] is missing the required "**forward** " function. * This error is shown if we misspell the ***forward()*** function in our SAE-architecture the ***StackedAutoEncoders*** class. |

**All code at once (train only)**

# *----------- AE : Recommender. SAE Stacked-Auto-Encoder ---------------*

# *Importing the libraries*

**from** turtle **import** clone

**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 -----------*

# *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)

# *-------- Creating the architecture of the Neural Netwark for SAE --------*

**class** StackedAutoEncoders(nn.Module):

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

**super**(StackedAutoEncoders, self).**\_\_init\_\_**()

        self.fc1 = **nn.Linear**(nb\_movies, 20)

        self.fc2 = **nn.Linear**(20, 10)

        self.fc3 = **nn.Linear**(10, 20)

        self.fc4 = **nn.Linear**(20, nb\_movies)

        self.activation = **nn.Sigmoid**()

**def** **forward**(self, x):

        x = self**.activation**(self**.fc1**(x))

        x = self**.activation**(self**.fc2**(x))

        x = self**.activation**(self**.fc3**(x))

        x = self**.fc4**(x)

**return** x

sae = **StackedAutoEncoders**()

criterion = **nn.MSELoss**()

optimizer = **optim.RMSprop**(**sae.parameters**(), lr = 0.01, weight\_decay= 0.5)

# *---- Training the SAE model ----*

nb\_epoch = 200

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

    train\_loss = 0

    s = 0.0

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

        input = **Variable**(train\_set\_tensor[id\_user]).**unsqueeze**(0)

        target = **input.clone**()

        non\_zero\_ratings = **torch.sum**(target.data **>** 0)

**if** non\_zero\_ratings **>** 0:

            output = **sae**(input)

            target.require\_grad = **False**

            output[target **==** 0] = 0

            loss = **criterion**(output, target)

            mean\_corrector = nb\_movies/**float**(non\_zero\_ratings + 1e-10)

**loss.backward**()

            train\_loss += **np.sqrt**((loss.data)\*mean\_corrector)

            s += 1.0

**optimizer.step**()

    train\_loss\_normalized = train\_loss/s

**print**(f"Epoch : {epoch}, Loss = {train\_loss\_normalized}")

# *python prctc\_SAE.py*

**15.3.9 Test the model : use test-set to calculate "test\_loss"**

Here we try to compute the ***test-set loss*** and we hope we can make it below 1 (1 star error of ratings).

* Similar to our RBM model, we *don't need* a *nested-for-loop*, because we are not training the model.
* We don't need *200 epochs* to measure the ***test-set*** performance to measure the ***test-set*** loss. We of course need only *one* *epoch*. Because we're gonna measure the ***global*** ***loss***, one time. We only need ***inner*** ***for*** ***loop*** here because this for loop loops over ***all*** the ***users*** of the data set.
* For the **target***-vector,* we now choose **test***-set* and for the **input***-vector* we use the **train***-set*. Because
* ***test\_loss*** compute the ***loss*** on ***test-set***. We initialized it to ***0***.
* Here ***cnt*** is the ***counter*** it is the number of users that rated at least one movie in the test-set.

test\_loss = 0

cnt = 0.0

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

    v = **Variable**(train\_set\_tensor[u\_id]).**unsqueeze**(0)     # *input vector from training-set*

    vt = **Variable**(test\_set\_tensor[u\_id]).**unsqueeze**(0)     # *target vector from "test-set"*

* The *input* corresponding to the user is of course, all the ratings of movies this user watched. We need the ***training set*** in ***input-vector***. Because:
* We put this *input vector* into the *AE-network*, then the *AE* will look at the *ratings* of the *movies* and especially the *positive ratings*, and based on these ratings, it will *predict* the *ratings* of the *movies* that the user *hasn't watched* yet (i.e. ratings that are not present in training-set). That’s the prediction.
* Here the point of using the ***test-set*** as ***target-vector*** is that, it contains the real-ratings of the movies that were *not present* in ***training-set*** (we used AE to generate those), so that we can *compare* *predicted* *& real ratings* and measure the ***test\_loss***.
* ***test\_loss*** will indicate how our model performs on new data.
* For example, if in our *input vector* our user gave *five star* ratings to all the *action* *movies* he watched, then when we feed this input vector into the network, well, the *neurons* corresponding to the *specific features* related to *action* *movies*, will be *activated* with a *large weight* to *predict* high ratings for the other *action* movies that the user *hasn't watched yet*.
* And then what we'll do, is we will compare this *predicted* ratings to the ratings of the *test-set* because the *test set* contains these ratings that were not *part* of the *training set*. That is, these action movies that the user *hasn't watched yet*, in the *training set*.

test\_loss = 0

cnt = 0.0

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

    v = **Variable**(train\_set\_tensor[u\_id]).**unsqueeze**(0)     # *input vector from training-set*

    vt = **Variable**(test\_set\_tensor[u\_id]).**unsqueeze**(0)     # *target vector from "test-set"*

    non\_zero\_ratings\_tst = **torch.sum**(vt.data **>** 0)

**if** non\_zero\_ratings\_tst **>** 0:

        tst\_output = **sae**(v)

        vt.require\_grad = **False**

        tst\_output[vt == 0] = 0

        loss\_tst = **criterion**(tst\_output, vt)

        mean\_corrector\_tst = nb\_movies/**float**(non\_zero\_ratings\_tst + 1e-10)

        test\_loss += **np.sqrt**((loss\_tst.data)\*mean\_corrector\_tst)

        cnt += 1.0

eval\_loSS = test\_loss/cnt

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

* The predictions happen right here: tst\_output = **sae**(v) . *The SAE model* ***sae*** *just making one step forward after training.*
* Our *forward* function that returns the *vector* of *predicted ratings* and therefore by *calling* our object on the *input* here, we will get our *vector* of *predicted ratings* for the movies that the user *hasn't watched yet*, and this will go into ***tst\_output***.
* We use " vt.require\_grad = **False** " to override the *computations* of the *gradient* with respect to the *target* because we don't need them.
* We don’t use **loss.backward**() or **optimizer.step**() either. Because those are for update weight and for training purpose.
* We use **tst\_output[vt == 0] = 0** to avoid the movies that are *not rated* by the user in the *test-set*.
* To calculate *loss*, between *real-ratings* in test-set and *predicted-ratings* from train-set:

**loss\_tst = criterion(tst\_output, vt)**

* We also need the ***mean\_cottector*** for test set:

**mean\_corrector\_tst = nb\_movies/float(non\_zero\_ratings\_tst + 1e-10)**

* Finally we compute the ***test\_loss*** and increment our counter ***cnt***:

**test\_loss += np.sqrt((loss\_tst.data)\*mean\_corrector\_tst)**

**cnt += 1.0**

* After the loop, we calculate the ***normalized-loss*** and print the result:

eval\_loSS = test\_loss/cnt

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

|  |  |
| --- | --- |
| * Result: Our test-loss is ***0.953***. That’s the error between the real & predicted ratings. On average, our model is going to *predict* a *rating* that will be *different* from the *real rating* by less than *one star*. If we manage to do this, that means that our recommended system will be pretty powerful because it can predict for new movies that you are going to like it or not. * We built a robust recommended system. The ***test loss*** is ***0.95*** stars, that is less than one star. So for example if you're applying this recommended system for the movie you're gonna watch tonight, and let's say that after watching the movie you give the rating *four stars*, then this *recommended system* would *predict* that you would give *between* *three and five stars* to this movie. |  |

**All code at once (practiced)**

# *----------- AE : Recommender. SAE Stacked-Auto-Encoder ---------------*

# *Importing the libraries*

**from** turtle **import** clone

**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 -----------*

# *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)

# *-------- Creating the architecture of the Neural Netwark for SAE --------*

**class** StackedAutoEncoders(nn.Module):

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

**super**(StackedAutoEncoders, self).**\_\_init\_\_**()

        self.fc1 = **nn.Linear**(nb\_movies, 20)

        self.fc2 = **nn.Linear**(20, 10)

        self.fc3 = **nn.Linear**(10, 20)

        self.fc4 = **nn.Linear**(20, nb\_movies)

        self.activation = **nn.Sigmoid**()

**def** **forward**(self, x):

        x = self**.activation**(self**.fc1**(x))

        x = self**.activation**(self**.fc2**(x))

        x = self**.activation**(self**.fc3**(x))

        x = self**.fc4**(x)

**return** x

sae = **StackedAutoEncoders**()

criterion = **nn.MSELoss**()

optimizer = **optim.RMSprop**(**sae.parameters**(), lr = 0.01, weight\_decay= 0.5)

# *---- Training the SAE model ----*

nb\_epoch = 200

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

    train\_loss = 0

    s = 0.0

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

        input = **Variable**(train\_set\_tensor[id\_user]).**unsqueeze**(0)

        target = **input.clone**()

        non\_zero\_ratings = **torch.sum**(target.data **>** 0)

**if** non\_zero\_ratings **>** 0:

            output = **sae**(input)

            target.require\_grad = **False**

            output[target **==** 0] = 0

            loss = **criterion**(output, target)

            mean\_corrector = nb\_movies/**float**(non\_zero\_ratings + 1e-10)

**loss.backward**()

            train\_loss += **np.sqrt**((loss.data)\*mean\_corrector)

            s += 1.0

**optimizer.step**()

    train\_loss\_normalized = train\_loss/s

**print**(f"Epoch : {epoch}, Loss = {train\_loss\_normalized}")

# *---- Testing the SAE model on Test-set ----*

# *Evaluating the SAE on Test-set*

test\_set\_ratings\_list = []

pridicted\_rating\_list = []

test\_loss = 0

cnt = 0.0

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

    v = **Variable**(train\_set\_tensor[u\_id]).**unsqueeze**(0)     # *input vector from training-set*

    vt = **Variable**(test\_set\_tensor[u\_id]).**unsqueeze**(0)     # *target vector from "test-set"*

    non\_zero\_ratings\_tst = **torch.sum**(vt.data **>** 0)

**if** non\_zero\_ratings\_tst **>** 0:

        tst\_output = **sae**(v)

        vt.require\_grad = **False**

        tst\_output[vt **==** 0] = 0     # *avoiding unrated movies in test-set*

        loss\_tst = **criterion**(tst\_output, vt)    # *comparing*

        mean\_corrector\_tst = nb\_movies/**float**(non\_zero\_ratings\_tst + 1e-10)

        test\_loss += **np.sqrt**((loss\_tst.data)\*mean\_corrector\_tst)

        cnt += 1.0

    # *creating list of original & predicted ratings : Tensor to NumPy-array conversion*

        """ Now we have to use detach() to convert tensor to Numpy-array

                 because our tensor requires "grad" now.

             We did it in RBM already but without detach() """

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

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

    predicted\_ratings = **tst\_output.detach**().**numpy**()

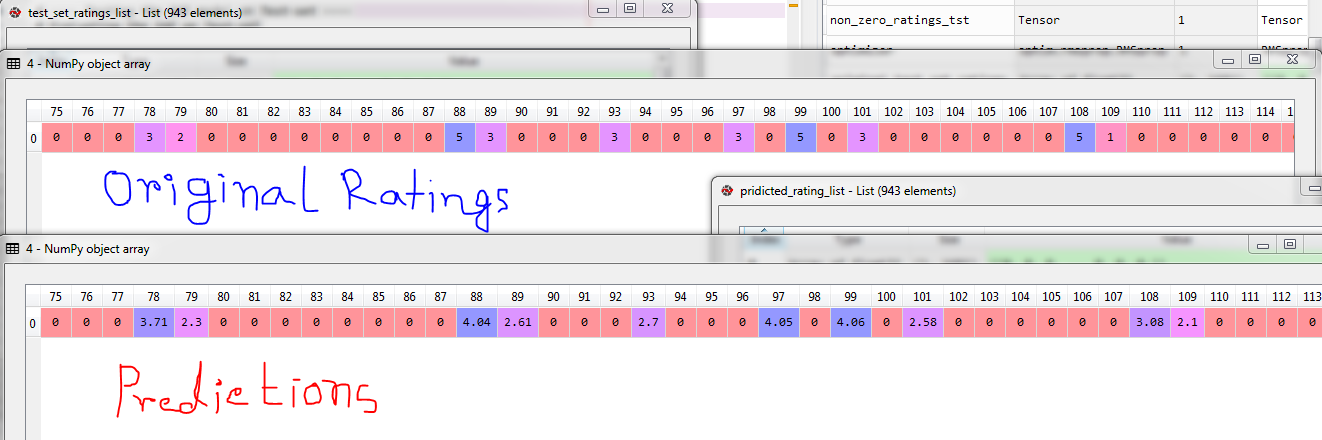
**pridicted\_rating\_list.append**(predicted\_ratings)

eval\_loSS = test\_loss/cnt

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

# *python prctc\_SAE.py*

* We compared the rating in the test-set & predicted-output for user no. 4.



***Another version from Github***

# *Stacked AutoEncoders: SAE*

# *Importing the libraries*

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

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

**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('ml-1m/movies.dat', sep='::', header=None, engine='python', encoding='latin-1')*

# *users = pd.read\_csv('ml-1m/users.dat', sep='::', header=None, engine='python', encoding='latin-1')*

# *ratings = pd.read\_csv('ml-1m/ratings.dat', sep='::', header=None, engine='python', encoding='latin-1')*

# *Preparing the training set and the test set*

training\_set = **pd.read\_csv**('movie\_lens\_100k/u1.base', delimiter='\t')

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

test\_set = **pd.read\_csv**('movie\_lens\_100k/u1.test', delimiter='\t')

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

# *Getting the number of users and movies*

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

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

# *Converting the data into an array with users in lines and movies in columns*

**def** **convert**(data):

    new\_data = []

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

        id\_movies = data[:, 1][data[:, 0] **==** id\_users]

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

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

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

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

**return** new\_data

# *Array with users in lines and movies in columns*

# *[user\_1, user\_2, ..., user\_943]*

# *[[movie\_1\_rating] [movie\_2\_rating] ... [movie\_1682\_rating]]*

training\_set = **convert**(training\_set)

test\_set = **convert**(test\_set)

# *Converting the data into Torch tensors*

training\_set = **torch.FloatTensor**(training\_set)#*.cuda()*

test\_set = **torch.FloatTensor**(test\_set)#*.cuda()*

# *Creating the architecture of the Neural Network*

**class** SAE(nn.Module):

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

**super**(SAE, self).**\_\_init\_\_**()

        self.fc1 = **nn.Linear**(nb\_movies, 20)

        self.fc2 = **nn.Linear**(20, 10)

        self.fc3 = **nn.Linear**(10, 20)

        self.fc4 = **nn.Linear**(20, nb\_movies)

        self.activation = **nn.Sigmoid**()

**def** **forward**(self, x):

        x = self**.activation**(self**.fc1**(x))

        x = self**.activation**(self**.fc2**(x))

        x = self**.activation**(self**.fc3**(x))

        x = self**.fc4**(x)

**return** x

sae = **SAE**()

criterion = **nn.MSELoss**()

optimizer = **optim.RMSprop**(**sae.parameters**(), lr=0.01, weight\_decay=0.5)

# *Training the SAE*

nb\_epoch = 200

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

    train\_loss = 0

    s = 0.  # *Number of users who rated at least 1 movie*

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

        # *Creates a 2D array instead of 1D, Pytorch only accept batch of data*

        input = **Variable**(training\_set[id\_user]).**unsqueeze**(0)

        target = **input.clone**()  # *Outputs should be the same as inputs*

**if** **torch.sum**(target.data) **>** 0:  # *User rated at least 1 movie*

            output = **sae.forward**(input)

            target.require\_grad = **False**  # *For code optimization*

            output[target **==** 0] = 0

            loss = **criterion**(output, target)

**loss.backward**()  # *Perform backpropagation*

**optimizer.step**()  # *Define the intensity of backward pass*

            # *1e-10 so that we never divide by 0*

            # *mean\_corrector corresponds to the average of the error of the rated movies only*

            # *it's not used in the backprop calculation, just for metrics*

            mean\_corrector = nb\_movies / **float**(**torch.sum**(target.data **>** 0) + 1e-10)

            # *train\_loss += np.sqrt(loss.data[0] \* mean\_corrector)*

            train\_loss += **np.sqrt**(loss.data \* mean\_corrector)

            s += 1.  # *Increment number of users who rated at least 1 movie*

**print**('epoch: ' + **str**(epoch) + ' loss: ' + **str**(train\_loss / s))

# *Testing the SAE*

test\_loss = 0

s = 0.

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

    # *The training set contains movies that the user has not yet watched*

    input = **Variable**(training\_set[id\_user]).**unsqueeze**(0)

    # *The test set contains the movies that the user watched*

    target = **Variable**(test\_set[id\_user])

**if** **torch.sum**(target.data) **>** 0:

        output = **sae.forward**(input)

        target.require\_grad = **False**

        # *output[target == 0] = 0*

        output[(target **==** 0).**unsqueeze**(0)]= 0

        loss = **criterion**(output, target)

        mean\_corrector = nb\_movies / **float**(**torch.sum**(target.data **>** 0) + 1e-10)

        # *test\_loss += np.sqrt(loss.data[0] \* mean\_corrector)*

        test\_loss += **np.sqrt**(loss.data \* mean\_corrector)

        s += 1.

**print**('test loss: ' + **str**(test\_loss / s))

""" IndexError: invalid index of a 0-dim tensor. Use `tensor.item()` in Python or `tensor.item<T>()` in C++ to convert a 0-dim tensor to a number

That's because in PyTorch>=0.5, the index of 0-dim tensor is invalid. The master branch is designed for PyTorch 0.4.1, loss\_val.data[0] works well.

Try to change

total\_loss += loss\_val.data[0]

loss\_values = [v.data[0] for v in losses]

to

total\_loss += loss\_val.data

loss\_values = [v.data for v in losses]

might fix the problem."""

""" IndexError: The shape of the mask [1682] at index 0 does not match the shape of the indexed tensor [1, 1682] at index 0.

Change:

output[target == 0] = 0      # I get error at this line

To:

output[(target == 0).unsqueeze(0)] = 0

Reason: The torch.Tensor returned by target == 0 is of the shape [1682].

(target == 0).unsqueeze(0) will convert it to [1, 1682] """

***Homework Challenge - Coding Exercise***

So far our training and test sets have the following format:

Column1: User

Column 2: Movie

Column 3: Rating

Column 4: Timestamp

Define a function that will convert this format into a list of horizontal lists, where each horizontal list corresponds to a user and includes all its ratings of the movies. In each list should also be included the movies that the user didn't rate and for these movies, just put a zero. So what you should get in the end is a huge list of 943 horizontal lists (because there are 943 users):

List of User 1: [Ratings of all the movies by User 1]

List of User 2: [Ratings of all the movies by User 2]

................................................................................

List of User 943: [Ratings of all the movies by User 943]

Why doing this ? Because we want to create a new structure of data, having the shape of a 2d array where:

the rows are the users,

the columns are the movies,

the cells are the ratings.

This coding exercise will be excellent practice for you because you will work with four important techniques in Python:

functions

lists and arrays

for loops

handling indexes

Try to complete this Homework as hard as you can, the more you try, the more you will progress.

The solution is in the next tutorial.

Good luck!