RecoTour III: Variational Autoencoders for Collaborative Filtering with Mxnet and Pytorch

Javier Rodriguez Zaurin Follow
Jun 20 · 13 min read ★

This post and the code here are part of a larger repo called RecoTour, where I normally explore and implement some recommendation algorithms that I consider interesting and/or useful (see RecoTour and RecoTourII). In every directory, I have included a `README` file and a series of explanatory notebooks that I hope help explaining the code. I keep adding algorithms from time to time, so stay tuned if you are interested.

As always, let me first acknowledge the relevant people that did the hard work. This post and the companion repo are based on the papers "Variational Autoencoders for Collaborative Filtering" [1] and "Auto-Encoding Variational Bayes" [2]. The code here and in that repo is partially inspired by the implementation from Younggyo Seo. I have adapted the code to my coding preferences and added a number of options and flexibility to run multiple experiment.

The reason to take a deep dive into variational auto-encoders for collaborative filtering is because they seem to be one of the few Deep Learning based algorithms (if not the only one) that obtains better results that those using non-Deep Learning techniques [3].

Throughout this exercise I will use two dataset. The Amazon Movies and TV dataset [4] [5] and the Movilens dataset. The later is used so I can make sure I am obtaining consistent results to those obtained in the paper. The Amazon dataset is significantly more challenging that the Movielens dataset as it is ~13 times more sparse.

All the experiments in this post were run using a p2.xlarge EC2 instance on AWS.

The more detailed, ***original version*** of this post in published in my blog. This intends to be a summary of the content there and focuses more on the implementation/code and the corresponding results and less on the math.

1. Partially Regularized Multinomial Variational Autoencoder: the Loss function

I will assume in this section that the reader has some experience with Variational Autoencoders (VAEs). If this is not the case, I recommend reading Kingma and Welling's paper, Liang et al paper, or the original post. There, the reader will find a detailed derivation of the *Loss* function we will be using when implementing the Partially Regularised Multinomial Variational Autoencoder (Mult-VAE). Here I will only include the final expression and briefly introduce some additional pieces of information that I consider useful to understand the Mult-VAE implementation and the loss below in Eq (1).

Let me first describe the notational convention. Following Liang et al., 2018, I will use **u** ∈ {1,…,*U*} to index users and **i** ∈ {1,…,*I*} to index items. The user-by-item binary interaction matrix (i.e. the click matrix) is **X** ∈ ℕ^{*U* × *I*} and I will use lower case $\boldsymbol{x_u}$ = [*X*_{u1},…,*X*_{uI}] ∈ ℕ^I to refer to the click history of an individual user **u**.

With that notation, the Mult-VAE *Loss* function is defined as:

$$Loss = -\frac{1}{M}\sum_{u=1}^{M}\left[\mathbf{x}_u \log(\pi(\mathbf{z}_u)) + \frac{\beta}{2}\sum_{j}(1 + \log(\sigma_{uj}^2) - \mu_{uj}^2 - \sigma_{uj}^2)\right]$$

Eq 1. Mult-VAE Loss function

where *M* is the mini-batch size. The first element within the summation is simply the log-likelihood of the click history $\boldsymbol{x_u}$ conditioned to the latent representation $\mathbf{z_u}$, i.e. $log(p\theta(\boldsymbol{x_u}|\mathbf{z_u}))$ (see below). The second element is the Kullback–Leibler divergence for VAEs when both the encoder and decoder distributions are Gaussians (see here).

We just need a bit more detail before we can jump to the code. $\boldsymbol{x_u}$, the click history of user **u**, is defined as:

$$\mathbf{x}_u \sim \mathrm{Mult}(N_u, \pi(\mathbf{z}_u))$$

Eq 2. Click history for user **u**

where $\boldsymbol{N_u} = \sum_i \boldsymbol{N_{ui}}$ is the total number of clicks for user **u**. As I mentioned before, $\mathbf{z_u}$ is latent representation of $\boldsymbol{x_u}$, and is assumed to be drawn from a standard Gaussian prior $p\theta(\mathbf{z_u}) \sim \boldsymbol{N}(0, \mathrm{I})$. During the implementation of the Mult-VAE, $\mathbf{z_u}$ needs to be sampled from an approximate posterior $q\phi(\mathbf{z_u}|\mathbf{x_u})$ (which is also assume to be Gaussian). Since computing gradients when sampling is involved is…"*complex*", Kingma and Welling introduced the so-called reparameterization trick (please, read the original paper, original post, or any of the multiple online resources for more details on the reparameterization trick), so that the sampled $\mathbf{z_u}$ will be computed as:

$$\mathbf{z}_u^s = \mu(\mathbf{x}_u) + \sigma(\mathbf{x}_u) \cdot \epsilon$$

Eq 3. Sampled **zu** (hence the superscript 's') that will be used in the implementation of the Mult-VAE

$\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ in Eq 3 are functions of neural networks and $\epsilon \sim \boldsymbol{N}(0, \mathrm{I})$ is Gaussian noise. Their computation will become clearer later in the post when we see the corresponding code. Finally, $\pi(\mathbf{z_u})$ in Eq (2) is $\pi(\mathbf{z_u}) = \mathit{Softmax}(\mathbf{z_u})$.

At this stage we have almost all the information we need to implement the Mult-VAE and its loss function in Eq (1): we know what $\boldsymbol{x_u}$ is, $\mathbf{z_u}$, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ will be functions of our neural networks, and π is just the *Softmax* function. The only "letter" left to discuss from Eq (1) is β.

Looking at the loss function in Eq (1) within the context of VAEs, we can see that the first term is the "reconstruction loss", while the *KL* divergence act as a regularizer. With that in mind, Liang et al add a factor β to control the strength of the regularization, and propose β<1. For a more in-depth refection of the role of β, and in general a better explanation of the form of the loss function for the Mult-VAE, please read the original paper or the original post.

Without further ado, let's move to the code:

2. Preparing the data.

As I mentioned earlier in the post, this intends to be a summary of the original post. With that in mind, I will briefly describe here how the data is prepared without including the code. The reader can find the code related to the data preparation in the original implementation, the original post or in the repo.

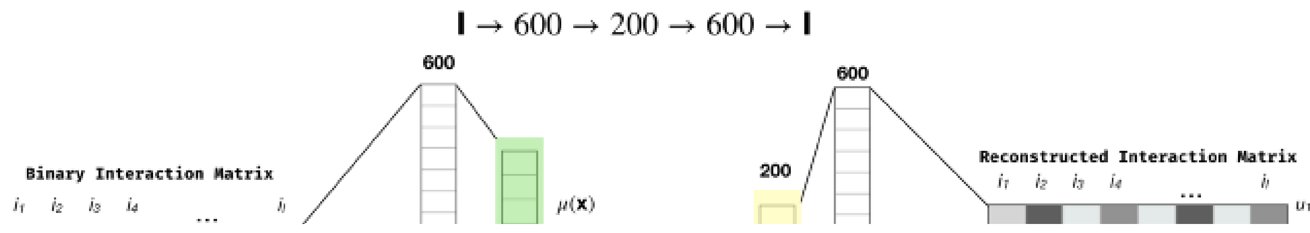
Basically, the authors split the data into training, validation and test users and their corresponding interactions. Then they split validation and test interactions into so-called "*validation and test train and test sets*".

I know that this sounds convoluted, but is not that complex. The "*validation_train and test_train sets*", comprising here 80% of the total validation and test sets, will be used to build what we could think as an input binary *"image"* (i.e. the binary matrix of clicks) to be "*encoded → decoded*" by the trained auto-encoder. On the other hand the "*validation_test and test_test sets*", comprising here 20% of the total validation and test sets, will be used to compute the ranking metrics at validation/test time. If you want more details along with a toy example please go to the corresponding notebook in the repo.

3. Partially Regularized Multinomial Variational Autoencoder: the code

I have implemented the Mult-VAE using both `Mxnet` 's `Gluon` and `Pytorch` . In this section I will concentrate only on the `Mxnet` implementation. Please go to the repo in case you are interested in the `Pytorch` implementation.

As with any Autoencoder architecture, the main two elements are (hold your breath…) the encoder and the decoder. In the original publication the authors used a one hidden layer MLP as generative model. There they say that deeper architectures do not improve the results, which I find it to be true after having run over 60 experiments. With that it mind, let's first have a look the model [*I→600→200→600→I*] where *I* is the total number of items:



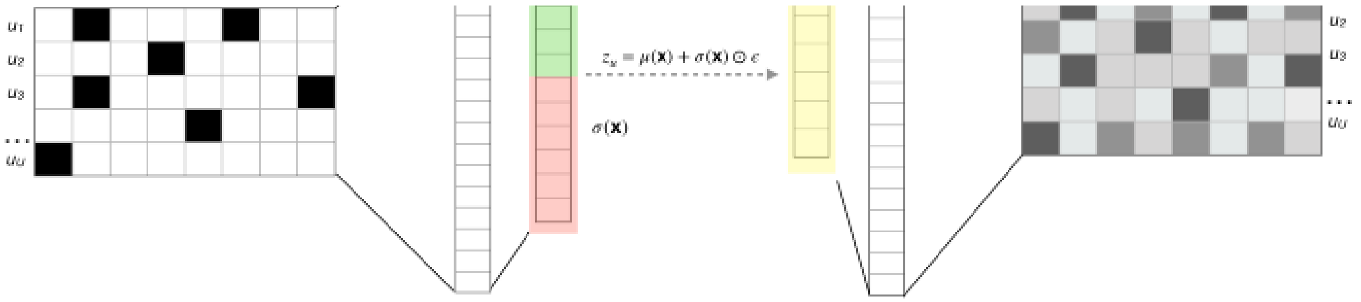


Fig 1. Mult-VAE architecture. The colours are my attempt to guide the eye through the reparameterization trick process.

In code, Fig (1) is:

3.1 The Encoder

```
1   class VAEEncoder(HybridBlock):
2       def __init__(self, q_dims: List[int], dropout: List[float]):
3           super().__init__()
4   
5           # last dim multiplied by two for the reparameterization trick
6           q_dims_ = q_dims[:-1] + [q_dims[-1] * 2]
7           with self.name_scope():
8               self.q_layers = nn.HybridSequential(prefix="q_net")
9               for p, inp, out in zip(dropout, q_dims_[:-1], q_dims_[1:]):
10                  self.q_layers.add(nn.Dropout(p))
11                  self.q_layers.add(nn.Dense(in_units=inp, units=out))
12  
13      def hybrid_forward(self, F, X):
14          h = F.L2Normalization(X)
15          for i, layer in enumerate(self.q_layers):
16              h = layer(h)
17              if i != len(self.q_layers) - 1:
18                  h = F.tanh(h)
19              else:
20                  mu, logvar = F.split(h, axis=1, num_outputs=2)
21          return mu, logvar
```

Mult-VAE_encoder.py hosted with ♡ by GitHub view raw

Snippet 1. Mult-VAE Encoder. Note the μ , σ split in line 20 as part of the reparameterization trick. This split corresponds to the green and red colours in Fig 1

3.2 The Decoder

```python
class Decoder(HybridBlock):
    def __init__(self, p_dims: List[int], dropout: List[float]):
        super().__init__()

        with self.name_scope():
            self.p_layers = nn.HybridSequential(prefix="p_net")
            for p, inp, out in zip(dropout, p_dims[:-1], p_dims[1:]):
                self.p_layers.add(nn.Dropout(p))
                self.p_layers.add(nn.Dense(in_units=inp, units=out))

    def hybrid_forward(self, F, X):
        h = X
        for i, layer in enumerate(self.p_layers):
            h = layer(h)
            if i != len(self.p_layers) - 1:
                h = F.tanh(h)
        return h
```

Mult-VAE_decoder.py hosted with ♡ by GitHub view raw

Snippet 2. Mult-VAE Decoder

3.3 The full model

```python
class MultiVAE(HybridBlock):
    def __init__(
        self,
        p_dims: List[int],
        dropout_enc: List[float],
        dropout_dec: List[float],
        q_dims: List[int] = None,
    ):
        super().__init__()

        self.encode = VAEEncoder(q_dims, dropout_enc)
        self.decode = Decoder(p_dims, dropout_dec)

    def hybrid_forward(self, F, X):
        mu, logvar = self.encode(X)
        std = F.exp(0.5 * logvar)
        eps = F.random.normal_like(std)
        sampled_z = mu + float(autograd.is_training()) * eps * std
        return self.decode(sampled_z), mu, logvar
```

Mult-VAE_model.py hosted with ♡ by GitHub view raw

Snippet 3. Mult-VAE model. Note the reparameterization happening in line 15–18. Line 18 corresponds to the yellow colour in Fig 1

Before I move on, let me mention (and appreciate) one of the many nice "*little*" things that `Mxnet` 's `Gluon` has to offer. You will notice the use of `HybridBlock` and the use of the input `F` (the backend) when we define the forward pass, or more precisely, the `hybrid_forward` pass. One could write a full post on the joys of `HybridBlocks` and how nicely and easily the guys that developed `Gluon` brought together the flexibility of imperative frameworks (e.g. `Pytorch`) and the speed of declarative frameworks (e.g. `Tensorflow`) together. If you want to learn the details go here, but believe me, this is FAST.

Having said that, there is only one more piece that we need to have the complete model, the loss function in Eq (1).

```
def vae_loss_fn(inp, out, mu, logvar, anneal):
    # first term
    neg_ll = -nd.mean(nd.sum(nd.log_softmax(out) * inp, -1))
    # second term without beta
    KLD = -0.5 * nd.mean(nd.sum(1 + logvar - nd.power(mu, 2) - nd.exp(logvar), axis=1))
    # "full" loss (anneal is beta in the expressions above)
    return neg_ll + anneal * KLD
```

Mult-VAE_loss.py hosted with ♡ by GitHub view raw

Snippet 4. Implementation of the Loss for the Mult-VAE (i.e. Eq (1))

In the paper the authors also use a Multinomial Denoising Autoencoder (Mult-DAE). The architecture is identical to that of the Mult-VAE apart from the fact that there is no variational aspect. I have implemented the Mult-DAE and run multiple experiments with it. However, given its simplicity and an already lengthy post, I will not discuss the corresponding code here.

Note that following the original implementation, I apply dropout at the input layer for both Mult-VAE and Mult-DAE to avoid overfitting. I also include the option for applying dropout throughout the network.

Also note that even though I have explored different dropouts, the best way of addressing the interplay between dropout, weight decay, β, etc, and the architecture is using "*proper*" hyperparamter optimization.

4. Train, Validate and Test

4.1 Annealing schedule

As mentioned before, we can interpret the Kullback-Leiber divergence as a regularization term. With that in mind, in a procedure inspired by Samuel R. Bowman et al, 2016 [6], Liang and co-authors linearly annealed the KL term (i.e. increase β) slowly over a large number of training steps.

More specifically, the authors anneal the KL divergence all the way to $\beta = 1$, reaching that value at around 80% of the total number of epochs used during the process. They then identify the best performing β based on the peak validation metric, and retrain the model with the same annealing schedule, but stop increasing β after reaching that value.

If we go to their implementation, these are the specifics of the process: using a batch size of 500 they set the total number of annealing steps to 200000. Given that the training dataset has a size of 116677, every epoch has 234 training steps. Their `anneal_cap` value, i.e. the maximum annealing reached during training, is set to 0.2, and during training they use the following approach:

```
if total_anneal_steps > 0:
                anneal = min(anneal_cap, 1. * update_count / total_anneal_steps)
            else:
                anneal = anneal_cap
```

original_anneal_schedule.py hosted with ♡ by GitHub view raw

Snippet 5. Original anneal schedule

where `update_count` will increase by 1 every training step/batch. They use 200 epochs, therefore, if we do the math, the `anneal_cap` value will stop increasing when `update_count / total_anneal_steps` = 0.2, i.e. after 40000 training steps, or in other words, after around 170 epochs, i.e. ~80% of the total number of epochs.

Whit that in mind my implementation looks like this:

```python
batch_size = 500
anneal_epochs = None
anneal_cap = 0.2
constant_anneal = False
n_epochs = 200

training_steps = len(range(0, train_data.shape[0], batch_size))
try:
    total_anneal_steps = (
        training_steps * (n_epochs - int(n_epochs * 0.2))
    ) / anneal_cap
except ZeroDivisionError:
    assert (
        constant_anneal
    ), "if 'anneal_cap' is set to 0.0 'constant_anneal' must be set to 'True"
```

my_anneal_schedule.py hosted with ♡ by GitHub view raw

Snippet 6. My slightly adapted anneal schedule.

once the `total_anneal_steps` is set, the only thing left is to define the training and validation steps. If you are familiar with `Pytorch` , the next two functions will be look very familiar to you.

4.2 Training Step

```python
def train_step(model, optimizer, data, epoch):

    running_loss = 0.0
    global update_count
    N = data.shape[0]
    idxlist = list(range(N))
    np.random.shuffle(idxlist)
    training_steps = len(range(0, N, batch_size))

    with trange(training_steps) as t:
        for batch_idx, start_idx in zip(t, range(0, N, batch_size)):
            t.set_description("epoch: {}".format(epoch + 1))

            end_idx = min(start_idx + batch_size, N)
            X_inp = data[idxlist[start_idx:end_idx]]
            X_inp = nd.array(X_inp.toarray()).as_in_context(ctx)

            if constant_anneal:
                anneal = anneal_cap
            else:
                anneal = min(anneal_cap, update_count / total_anneal_steps)
            update_count += 1

            with autograd.record():
                if model.__class__.__name__ == "MultiVAE":
                    X_out, mu, logvar = model(X_inp)
                    loss = vae_loss_fn(X_inp, X_out, mu, logvar, anneal)
                    train_step.anneal = anneal
                elif model.__class__.__name__ == "MultiDAE":
                    X_out = model(X_inp)
                    loss = -nd.mean(nd.sum(nd.log_softmax(X_out) * X_inp, -1))
            loss.backward()
            trainer.step(X_inp.shape[0])
            running_loss += loss.asscalar()
            avg_loss = running_loss / (batch_idx + 1)

            t.set_postfix(loss=avg_loss)
```

mxnet_train_step.py hosted with ♡ by GitHub view raw

Snippet 7. Training step using Mxnet's Gluon

4.3 Validation Step

```
def eval_step(data_tr, data_te, data_type="valid"):

    running_loss = 0.0
    eval_idxlist = list(range(data_tr.shape[0]))
    eval_N = data_tr.shape[0]
    eval_steps = len(range(0, eval_N, batch_size))

    n100_list, r20_list, r50_list = [], [], []

    with trange(eval_steps) as t:
        for batch_idx, start_idx in zip(t, range(0, eval_N, batch_size)):
            t.set_description(data_type)

            end_idx = min(start_idx + batch_size, eval_N)
            X_tr = data_tr[eval_idxlist[start_idx:end_idx]]
            X_te = data_te[eval_idxlist[start_idx:end_idx]]
            X_tr_inp = nd.array(X_tr.toarray()).as_in_context(ctx)

            with autograd.predict_mode():
                if model.__class__.__name__ == "MultiVAE":
                    X_out, mu, logvar = model(X_tr_inp)
                    loss = vae_loss_fn(X_tr_inp, X_out, mu, logvar, train_step.anneal)
                elif model.__class__.__name__ == "MultiDAE":
                    X_out = model(X_tr_inp)
                    loss = -nd.mean(nd.sum(nd.log_softmax(X_out) * X_tr_inp, -1))

            running_loss += loss.asscalar()
            avg_loss = running_loss / (batch_idx + 1)

            # Exclude examples from training set
            X_out = X_out.asnumpy()
            X_out[X_tr.nonzero()] = -np.inf

            n100 = NDCG_binary_at_k_batch(X_out, X_te, k=100)
            r20 = Recall_at_k_batch(X_out, X_te, k=20)
            r50 = Recall_at_k_batch(X_out, X_te, k=50)
            n100_list.append(n100)
            r20_list.append(r20)
            r50_list.append(r50)

            t.set_postfix(loss=avg_loss)

        n100_list = np.concatenate(n100_list)
        r20_list = np.concatenate(r20_list)
        r50_list = np.concatenate(r50_list)

    return avg_loss, np.mean(n100_list), np.mean(r20_list), np.mean(r50_list)
```

mxnet_eval_step.py hosted with ♡ by GitHub view raw

Snippet 8. Evaluation step using Mxnet's Gluon.

I have widely discussed the evaluation metrics (NDCG@k and Recall@k) in a number of notebooks in this repo (and corresponding posts). Therefore, with that in mind and with the aim of not making this a more "infinite post", I will not describe the corresponding implementation here. If you want details on those evaluation metrics, please go the `metrics.py` module in `utils`. The code there is a very small adaptation to the one in the original implementation.

4.4. Running the process:

Let's define the model, prepare the set up and run a small sample (of course, ignore the results printed. I only want to illustrate how to run the model)

```python
model = MultiVAE(
    p_dims=[200, 600, n_items],
    q_dims=[n_items, 600, 200],
    dropout_enc=[0.5, 0.0],
    dropout_dec=[0.0, 0.0],
)

ctx = mx.gpu() if mx.context.num_gpus() else mx.cpu()
model.initialize(mx.init.Xavier(), ctx=ctx)
model.hybridize()
optimizer = mx.optimizer.Adam(learning_rate=0.001, wd=0.)
trainer = gluon.Trainer(model.collect_params(), optimizer=optimizer)

stop_step = 0
update_count = 0
eval_every = 1
stop = False
for epoch in range(1):
    train_step(model, optimizer, train_data[:2000], epoch)
    if epoch % eval_every == (eval_every - 1):
        val_loss, n100, r20, r50 = eval_step(valid_data_tr[:1000], valid_data_te[:1000])
        print("=" * 80)
        print(
            "| valid loss {:4.3f} | n100 {:4.3f} | r20 {:4.3f} | "
            "r50 {:4.3f}".format(val_loss, n100, r20, r50)
        )
        print("=" * 80)
```

mxnet_running_process.py hosted with ♡ by GitHub view raw

Snippet 9. Running the model

And with a few more rings and bells (e.g. optional learning rate scheduler, early stopping, etc…) this is exactly the code that you will find in `main_mxnet.py` .

Before I move to the next, final section, just a quick comment about something I normally find in these scientific publications. Normally, once they have found the best hyperparameters on the validation set, they test the model on the test set.

In “real-life” scenarios, there would be one additional step, the one merging the train and validation sets, re-training the model with the best hyperparameters and then testing on the test set. In any case, since here my goal is not to build a real-life system, I will follow the same procedure to that found in the original implementation.

Time now to have a look to the results obtained with both `Pytorch` and `Mxnet` .

5. Summary of the results

let me remind again the annealing schedule described in Section 4.1. Basically, we gradually anneal to $\beta=1$, which is reached at around 80% of the total number of epochs, and record the best anneal parameter ($\beta best$). Then we apply the same annealing schedule but with $\beta best$, i.e. we anneal to $\beta best$ reaching that value at around 80% of the total number of epochs.

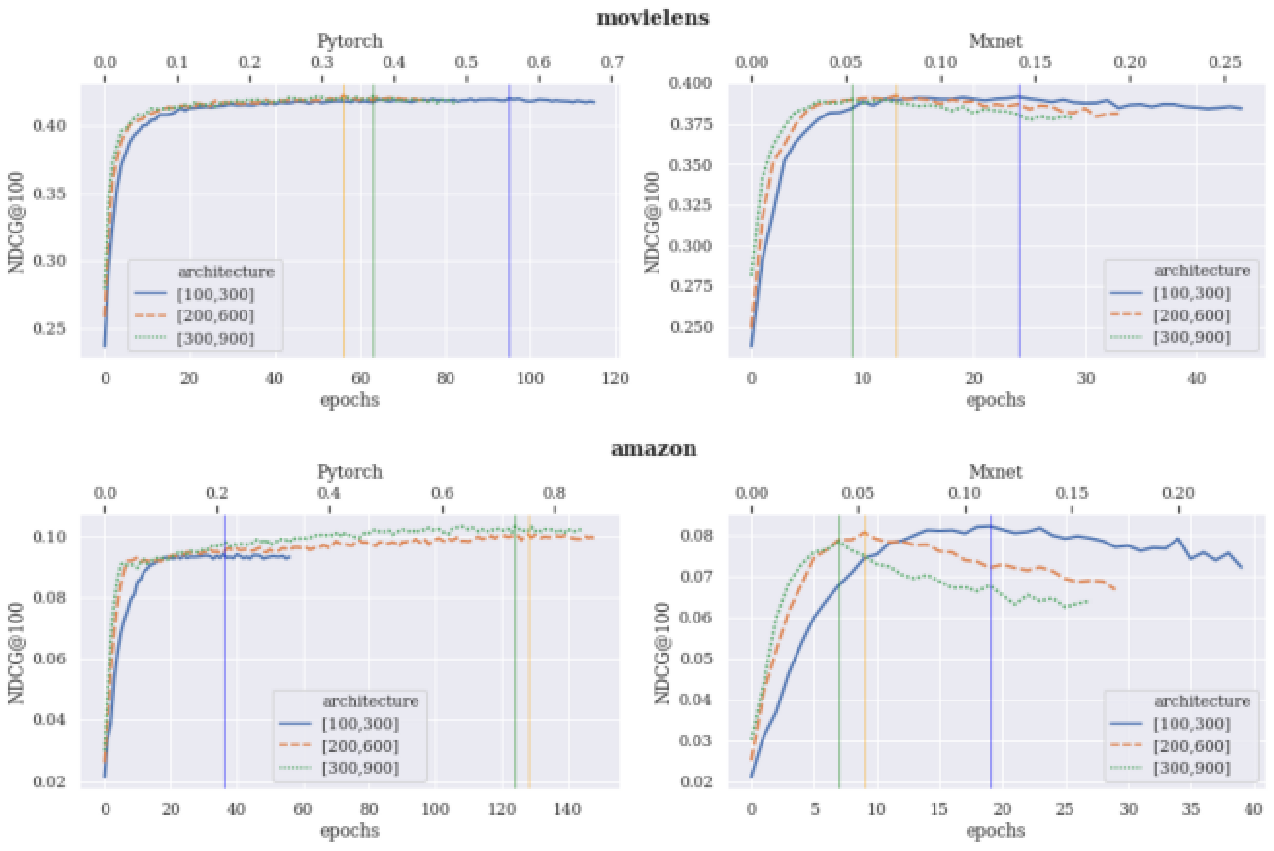


Figure 2. Same annealing schedule as on Liang et al 2018 for 3 different architectures and for the Movielens and the Amazon datasets using `Pytorch` and `Mxnet` . During the annealing schedule, β=1\beta=1β=1 is reached at 170 epochs (out of 200, i.e. 85%)

Search this file…

dataset	dl_frame	model	p_dims	weight_decay	lr	lr_scheduler	anneal_cap	best_epoch	loss
amazon	Pytorch	dae	'[50 -> 150]'	0.0	0.001	False	NA	28	87.588
amazon	Mxnet	dae	'[100 -> 300]'	0.0	0.001	False	NA	18	85.985
amazon	Pytorch	vae	'[300 -> 900]'	0.0	0.001	False	0.7	170	92.263
amazon	Mxnet	vae	'[200 -> 600]'	0.0	0.001	False	0	8	85.310
movielens	Pytorch	dae	'[200 -> 600]'	0.0	0.001	False	NA	136	349.714
movielens	Mxnet	dae	'[200 -> 600]'	0.0	0.005	True	NA	184	348.841
movielens	Pytorch	vae	'[200 -> 600]'	0.0	0.005	True	0.2	155	365.372
movielens	Mxnet	vae	'[200 -> 600]'	0.0	0.001	False	0	101	350.479

results_table.csv hosted with ♡ by **GitHub** **view raw**

Table 1. Best performing experiments (in terms of NDCG@10) among all the experiments I run (which can be found in the ` `run_experiment.sh` file in the repo). A csv file with the results for all experiments run can be found in the `all_results.csv` file in the repo.

Figure 2 reproduces the same annealing schedule for 3 different architectures and for the Movielens and the Amazon datasets using `Pytorch` and `Mxnet`. During the annealing schedule, β=1 is reached at 170 epochs (out of 200, i.e. 85%). In addition, I have also used early stopping with a "*patience*" of 20 epochs, which is why none of the experiments reaches the 200 epochs. The vertical lines in the figure indicate the epoch at which the best `NDGC@100` is reached, and the corresponding β value is indicated in the top x-axis.

On the other hand, Table 1 shows the best results I obtained for all the experiments I run, which you can find in this repo in the file `run_experiments.sh`.

At first sight it is apparent how different the two deep learning frames behave. I find `Pytorch` to perform a bit better than `Mxnet` and to be more stable across experiments. This is something that I keep finding every time I use these two frames for the same exercise. For example, here, using Hierarchical Attention networks. I actually believe this is due to the fact that I know (or have used) more `Pytorch` than `Mxnet`. Nonetheless, at this stage it is clear for me that I need to do a proper benchmark exercise between these two deep learning libraries.

Focusing on the results shown in Figure 1, the first apparent result is that the `Mxnet` implementation performs better with little or no regularization. In fact, I have run over

60 experiments and, as shown in Table 1, the best results when using `Mult-VAE` and `Mxnet` are obtained with no regularization, i.e. a denoising autoencoder with the reparametrization trick. Furthermore, the best overall metrics with `Mxnet` are obtained using the `Mult-DAE` (NDCG@100 = 0.424).

If we focus in the differences between datasets, it is first apparent that the metrics are significantly smaller for the Amazon dataset relative to those obtained with the Movielens dataset. This was of course expected since the Amazon dataset is 13 times more sparse that the Movielens dataset, i.e. significantly more challenging. In addition, we see that the `Pytorch` implementation shows a very stable behavior for both datasets and architectures, reaching the best `NDCG@10` later in the training epochs in the case of the Amazon dataset. Again this is different in the case of the `Mxnet` implementation, where we see less consistency and the maximum `NDCG@10` being reached very early during training for both datasets.

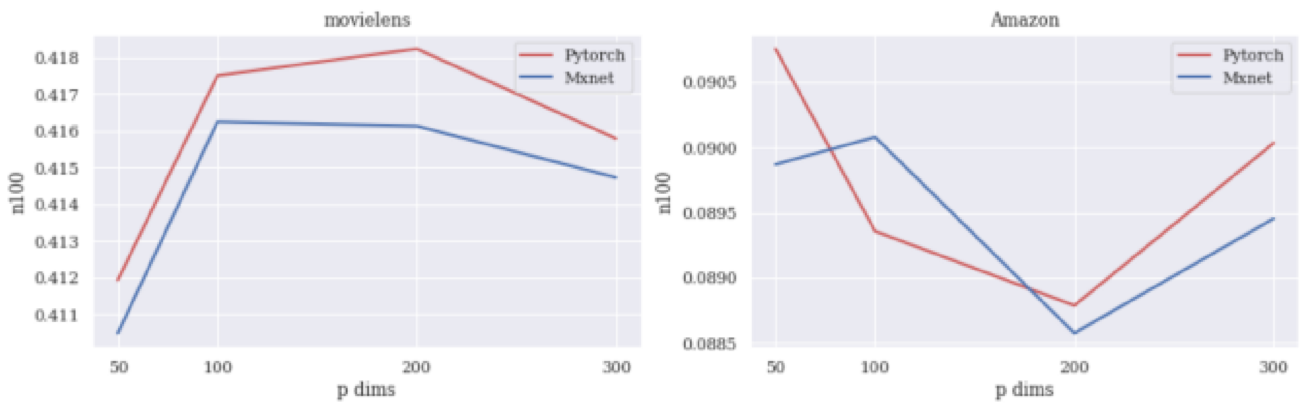


Figure 3. NDCG100 (referred as `n100 in the figure`) plotted against the first dimension of the decoder for the following architectures: i) I→150→50→150→I, ii) I→300→100→300→I, iii) I→600→200→600→I and iv) I→900→300→900→I

On the other hand, Liang et al mentioned in their paper that deeper architectures did not lead to any improvement. This is consistent with the results I found in my experiments. In fact, Figure 3 shows the NDCG100 (refereed in the figure as `n100`) vs the first dimension of the decoder for four different architectures. As we can see in the figure, even concentrating in architectures with the same number of layers, adding neurons per layer was not particularly helpful beyond a certain number (50 and 200 for the Movilens and the Amazon dataset respectively).

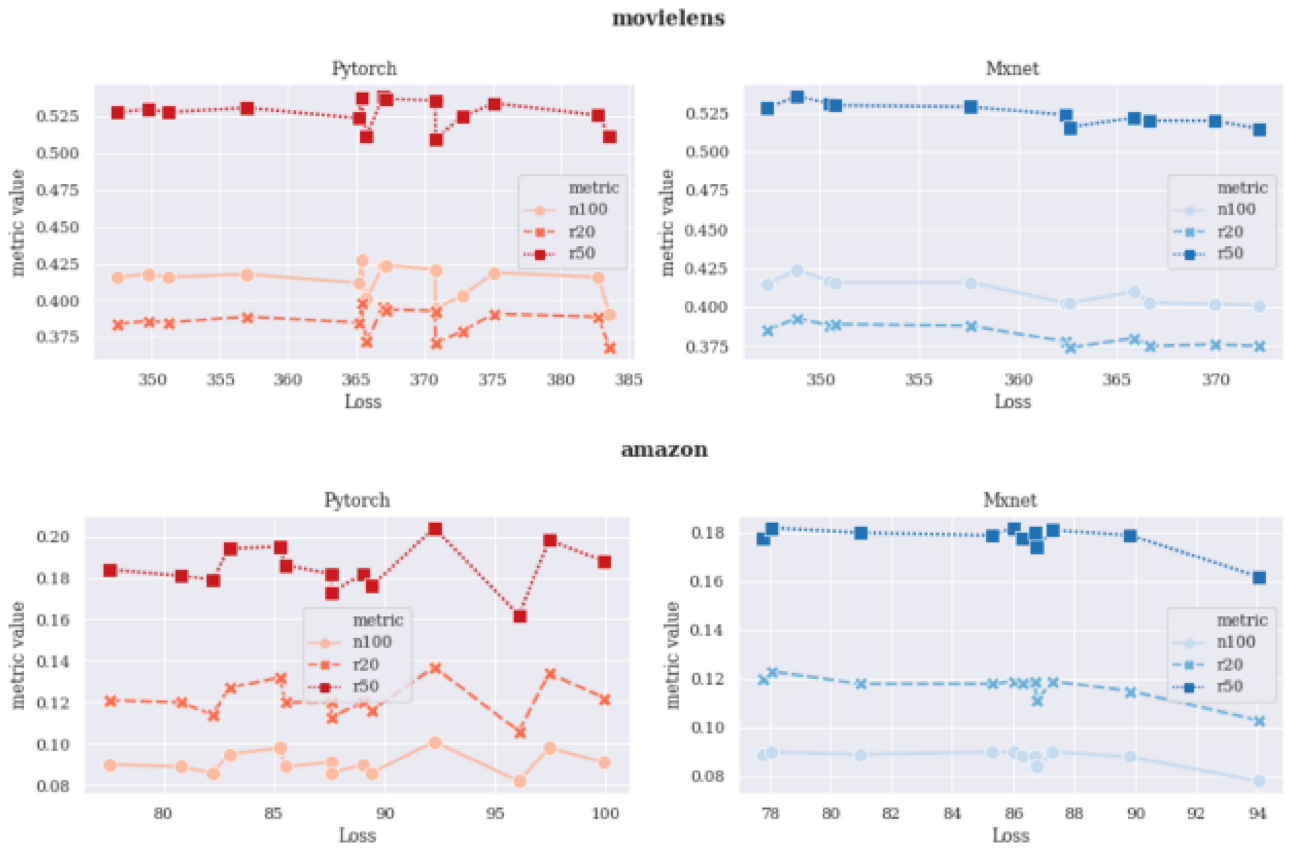


Figure 4. NDGC@100 (n100), and Recall@20 (r20) and Recall@50 (r50) plotted against the loss for all experiments I run

Before I end up this exercise I wanted to emphasise a result I have already discussed in the past (see here) and that is illustrated in Fig 4.

Fig 4 shows that, in general, the best ranking metrics do not correspond to the best loss values. Even though the the reconstruction of the input matrix of clicks might be worse, the ranking metrics might still improve. This is an important and not uncommon results, and something one has to bear in mind when building real world recommendation systems. When building recommendation algorithms we are not interested in achieving the best classification/regression loss, but in producing the best recommendations, which is more related to information retrieval effectiveness, and therefore ranking metrics. For more information on this and many other aspects of recommendation systems, I recommend this fantastic book. Chapter 7 in that book focuses on evaluation metrics.

And with this, I conclude my experimentation around the `Mult-VAE` with `Pytorch` and `Mxnet`

The next, most immediate projects I want to add to the repo are:

1. Sequential Variational Autoencoders for Collaborative Filtering [7]
2. LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation [8]

If you manage to read all that, I hope you found it useful.

References:

[1] Dawen Liang, Rahul G. Krishnan, Matthew D. Hoffman, Tony Jebara, 2018. Variational Autoencoders for Collaborative Filtering: arXiv:1802.05814v1

[2] Diederik P Kingma, Max Welling, 2014. Auto-Encoding Variational Bayes: arXiv:1312.6114v10

[3] Maurizio Ferrari Dacrema, Paolo Cremonesi, Dietmar Jannach. Are We Really Making Much Progress? A Worrying Analysis of Recent Neural Recommendation Approaches: arXiv:1907.06902v3

[4] J. McAuley, C. Targett, J. Shi, A. van den Hengel. 2015. Image-based recommendations on styles and substitutes. arXiv:1506.04757v1

[5] R. He, J. McAuley, 2016. Modeling the visual evolution of fashion trends with one-class collaborative filtering. arXiv:1602.01585v1

[6] Samuel R. Bowman, Luke Vilnis, Oriol Vinyals, Andrew M. Dai, Rafal Jozefowicz, Samy Bengio, 2016. Generating Sentences from a Continuous Space: arXiv:1511.06349v4

[7] Noveen Sachdeva, Giuseppe Manco, Ettore Ritacco, Vikram Pudi, 2018. Sequential Variational Autoencoders for Collaborative Filtering: arXiv:1811.09975v1

[8] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, Meng Wang, 2020. LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation arXiv:2002.02126v2

Variational Autoencoder Recommendation System Python Mxnet Pytorch