
recnn

Release 0.1

Dec 08, 2020

1	What	3
2	Getting Started	5
2.1	Examples Page	5
2.1.1	Getting Started with recnn	5
2.1.2	Working with your own data	8
2.1.3	Using Pandas Backends	12
2.2	NN	13
2.2.1	Models	13
2.2.2	Update	14
2.2.3	Algo	16
2.3	Data	16
2.3.1	env	16
2.3.2	dataset_functions	18
3	Indices and tables	21
	Python Module Index	23
	Index	25

CHAPTER 1

What

This is my school project. It focuses on Reinforcement Learning for personalized news recommendation. The main distinction is that it tries to solve online off-policy learning with dynamically generated item embeddings. Also, there is no exploration, since we are working with a dataset. In the example section, I use Google's BERT on the ML20M dataset to extract contextual information from the movie description to form the latent vector representations. Later, you can use the same transformation on new, previously unseen items (hence, the embeddings are dynamically generated). If you don't want to bother with embeddings pipeline, I have a DQN embeddings generator as a proof of concept.

There are a couple of ways you can get started. The most straightforward is to clone and go to the examples section. You can also use Google Colab or Gradient Experiment.

How parameters should look like:

```
import torch
import recnn

env = recnn.data.env.FrameEnv('ml20_pca128.pkl', 'ml-20m/ratings.csv')

value_net = recnn.nn.Critic(1290, 128, 256, 54e-2)
policy_net = recnn.nn.Actor(1290, 128, 256, 6e-1)

cuda = torch.device('cuda')
ddpg = recnn.nn.DDPG(policy_net, value_net)
ddpg = ddpd.to(cuda)

for batch in env.train_dataloader:
    ddpd.update(batch, learn=True)
```

2.1 Examples Page

Welcome to the tutorial page. It is advised you'd use Local Jupyter/Google Colab/Gradient instead. Those are mostly copies of the notebooks.

2.1.1 Getting Started with recnn

Colab Version Here (clickable):

Offline example is in: RecNN/examples/[Library Basics]/1. Getting Started.ipynb

Let's do some imports:

```
import recnn

import recnn
import torch
import torch.nn as nn
from tqdm.auto import tqdm

tqdm.pandas()

from jupyterthemes import jtplot
jtplot.style(theme='grade3')
```

Environments

Main abstraction of the library for datasets is called environment, similar to how other reinforcement learning libraries name it. This interface is created to provide SARSA like input for your RL Models. When you are working with recommendation env, you have two choices: using static length inputs (say 10 items) or dynamic length time series with sequential encoders (many to one rnn). Static length is provided via FrameEnv, and dynamic length along with sequential state representation encoder is implemented in SeqEnv. Let's take a look at FrameEnv first:

In order to initialize an env, you need to provide embeddings and ratings directories:

```
frame_size = 10
batch_size = 25
# embeddings: https://drive.google.com/open?id=1EQ\_zXBR3DKpmJR3jBgLvt-xoOvArGMsL
dirs = recnn.data.env.DataPath(
    base="../../data/",
    embeddings="embeddings/ml20_pca128.pkl",
    ratings="ml-20m/ratings.csv",
    cache="cache/frame_env.pkl", # cache will generate after you run
    use_cache=True
)
env = recnn.data.env.FrameEnv(dirs, frame_size, batch_size)

train = env.train_batch()
test = env.train_batch()
state, action, reward, next_state, done = recnn.data.get_base_batch(train, ↵
    ↵device=torch.device('cpu'))

print(state)

# State
tensor([[ 5.4261, -4.6243,  2.3351, ...,  3.0000,  4.0000,  1.0000],
        [ 6.2052, -1.8592, -0.3248, ...,  4.0000,  1.0000,  4.0000],
        [ 3.2902, -5.0021, -10.7066, ...,  1.0000,  4.0000,  2.0000],
        ...,
        [ 3.0571, -4.1390, -2.7344, ...,  3.0000, -3.0000, -1.0000],
        [ 0.8177, -7.0827, -0.6607, ..., -3.0000, -1.0000,  3.0000],
        [ 9.0742,  0.3944, -6.4801, ..., -1.0000,  3.0000, -1.0000]])
```

Recommending

Let's initialize main networks, and recommend something!

```

value_net = recnn.nn.Critic(1290, 128, 256, 54e-2)
policy_net = recnn.nn.Actor(1290, 128, 256, 6e-1)

recommendation = policy_net(state)
value = value_net(state, recommendation)
print(recommendation)
print(value)

# Output:

tensor([[ 1.5302, -2.3658,  1.6439, ...,  0.1297,  2.2236,  2.9672],
        [ 0.8570, -1.3491, -0.3350, ..., -0.8712,  5.8390,  3.0899],
        [-3.3727, -3.6797, -3.9109, ...,  3.2436,  1.2161, -1.4018],
        ...,
        [-1.7834, -0.4289,  0.9808, ..., -2.3487, -5.8386,  3.5981],
        [ 2.3813, -1.9076,  4.3054, ...,  5.2221,  2.3165, -0.0192],
        [-3.8265,  1.8143, -1.8106, ...,  3.3988, -3.1845,  0.7432]],
        grad_fn=<AddmmBackward>)
tensor([[ -1.0065],
        [  0.3728],
        [  2.1063],
        ...,
        [ -2.1382],
        [  0.3330],
        [  5.4069]], grad_fn=<AddmmBackward>)

```

Algo classes

Algo is a high level abstraction for an RL algorithm. You need two networks (policy and value) in order to initialize it. Later on you can tweak parameters and stuff in the algo itself.

Important: you can set writer to torch.SummaryWriter and get the debug output Tweak how you want:

```

ddpg = recnn.nn.DDPG(policy_net, value_net)
print(ddpg.params)
ddpg.params['gamma'] = 0.9
ddpg.params['policy_step'] = 3
ddpg.optimizers['policy_optimizer'] = torch.optim.Adam(ddpg.nets['policy_net'], your_
→lr)
ddpg.writer = torch.utils.tensorboard.SummaryWriter('./runs')
ddpg = ddpd.to(torch.device('cuda'))

```

ddpg.loss_layout is also handy, it allows you to see how the loss should look like

```

# test function
def run_tests():
    batch = next(iter(env.test_dataloader))
    loss = ddpd.update(batch, learn=False)
    return loss

value_net = recnn.nn.Critic(1290, 128, 256, 54e-2)
policy_net = recnn.nn.Actor(1290, 128, 256, 6e-1)

cuda = torch.device('cuda')
ddpg = recnn.nn.DDPG(policy_net, value_net)
ddpg = ddpd.to(cuda)

```

(continues on next page)

(continued from previous page)

```

plotter = recnn.utils.Plotter(ddpg.loss_layout, [['value', 'policy']],)
ddpg.writer = SummaryWriter(dir='./runs')

from IPython.display import clear_output
import matplotlib.pyplot as plt
%matplotlib inline

plot_every = 50
n_epochs = 2

def learn():
    for epoch in range(n_epochs):
        for batch in tqdm(env.train_dataloader):
            loss = ddpq.update(batch, learn=True)
            plotter.log_losses(loss)
            ddpq.step()
            if ddpq._step % plot_every == 0:
                clear_output(True)
                print('step', ddpq._step)
                test_loss = run_tests()
                plotter.log_losses(test_loss, test=True)
                plotter.plot_loss()
            if ddpq._step > 1000:
                return

learn()

```

Update Functions

Basically, the Algo class is a high level wrapper around the update function. The code for that is pretty messy, so if you want to check it out, I explained it in the colab notebook linked at the top.

2.1.2 Working with your own data

Colab Version Here (clickable):

Some things to know beforehand:

When you load and preprocess data, all of the additional data preprocessing happens in the 'prepare_dataset' function that you should pass. An example of that is in the your own data notebook. Also if you have inconsistent indexes (i.e. movies index in MovieLens looks like [1, 3, 10, 20]), recnn handles in on its own, reducing memory usage. There is no need to worry about mixing up indexes while preprocessing your own data.

Here is how default ML20M dataset is processed. Use this as a reference:

```

def prepare_dataset(args_mut: DataFuncArgsMut, kwargs: DataFuncKwargs):
    # get args
    frame_size = kwargs.get('frame_size')
    key_to_id = args_mut.base.key_to_id
    df = args_mut.df

    # rating range mapped from [0, 5] to [-5, 5]
    df['rating'] = try_progress_apply(df['rating'], lambda i: 2 * (i - 2.5))

```

(continues on next page)

(continued from previous page)

```

# id's tend to be inconsistent and sparse so they are remapped here
df['movieId'] = try_progress_apply(df['movieId'], lambda i: key_to_id.get(i))
users = df[['userId', 'movieId']].groupby(['userId']).size()
users = users[users > frame_size].sort_values(ascending=False).index

if pd.get_type() == "modin":
    df = df._to_pandas() # pandas groupby is sync and doesnt affect performance
    ratings = df.sort_values(by='timestamp').set_index('userId').drop('timestamp',
↳axis=1).groupby('userId')

# Groupby user
user_dict = {}

def app(x):
    userid = int(x.index[0])
    user_dict[userid] = {}
    user_dict[userid]['items'] = x['movieId'].values
    user_dict[userid]['ratings'] = x['rating'].values

try_progress_apply(ratings, app)

args_mut.user_dict = user_dict
args_mut.users = users

return args_mut, kwargs

```

Look in reference/data/dataset_functions for further details.

Toy Dataset

The code below generates an artificial dataset:

```

import pandas as pd
import numpy as np
import datetime
import random
import time

def random_string_date():
    return datetime.datetime.strptime('{} {} {} {}'.format(random.randint(1, 366),
                                                            random.randint(0, 23),
                                                            random.randint(1, 59),
                                                            2019), '%j %H %M %Y').
↳strptime("%m/%d/%Y, %H:%M:%S")

def string_time_to_unix(s):
    return int(time.mktime(datetime.datetime.strptime(s, "%m/%d/%Y, %H:%M:%S").
↳timetuple()))

size = 100000
n_emb = 1000
n_usr = 1000
mydf = pd.DataFrame({'book_id': np.random.randint(0, n_emb, size=size),
                    'reader_id': np.random.randint(1, n_usr, size=size),
                    'liked': np.random.randint(0, 2, size=size),

```

(continues on next page)

(continued from previous page)

```

        'when': [random_string_date() for i in range(size)])
my_embeddings = dict([(i, torch.tensor(np.random.randn(128)).float()) for i in
↳range(n_emb)])
mydf.head()

# output:
   book_id  reader_id  liked  when
0         0         919     130  0  06/16/2019, 11:54:00
1         1         850     814  1  11/29/2019, 12:35:00
2         2         733     553  0  07/07/2019, 05:45:00
3         3         902     695  1  02/03/2019, 10:29:00
4         4         960     993  1  05/29/2019, 01:35:00

# saving the data
! mkdir mydataset
import pickle

mydf.to_csv('mydataset/mydf.csv', index=False)
with open('mydataset/myembeddings.pickle', 'wb') as handle:
    pickle.dump(my_embeddings, handle)

```

Writing custom preprocessing function

The following is a copy of the preprocessing function listed above to work with the toy dataset:

```

def prepare_my_dataset(args_mut, kwargs):

    # get args
    frame_size = kwargs.get('frame_size')
    key_to_id = args_mut.base.key_to_id
    df = args_mut.df

    df['liked'] = df['liked'].apply(lambda a: (a - 1) * (1 - a) + a)
    df['when'] = df['when'].apply(string_time_to_unix)
    df['book_id'] = df['book_id'].apply(key_to_id.get)

    users = df[['reader_id', 'book_id']].groupby(['reader_id']).size()
    users = users[users > frame_size].sort_values(ascending=False).index

    # If using modin: pandas groupby is sync and doesnt affect performance
    # if pd.get_type() == "modin": df = df._to_pandas()
    ratings = df.sort_values(by='when').set_index('reader_id').drop('when', axis=1).
↳groupby('reader_id')

    # Groupby user
    user_dict = {}

    def app(x):
        userid = x.index[0]
        user_dict[int(userid)] = {}
        user_dict[int(userid)]['items'] = x['book_id'].values
        user_dict[int(userid)]['ratings'] = x['liked'].values

    ratings.apply(app)

```

(continues on next page)

(continued from previous page)

```

args_mut.user_dict = user_dict
args_mut.users = users

return args_mut, kwargs

```

Putting it all together

Final touches:

```

frame_size = 10
batch_size = 25

dirs = recnn.data.env.DataPath(
    base="/mydataset",
    embeddings="myembeddings.pickle",
    ratings="mydf.csv",
    cache="cache/frame_env.pkl", # cache will generate after you run
    use_cache=True # generally you want to save env after it runs
)
# pass prepare_my_dataset here
env = recnn.data.env.FrameEnv(dirs, frame_size, batch_size, prepare_dataset=prepare_
↪my_dataset)

# test function
def run_tests():
    batch = next(iter(env.test_dataloader))
    loss = ddpq.update(batch, learn=False)
    return loss

value_net = recnn.nn.Critic(1290, 128, 256, 54e-2)
policy_net = recnn.nn.Actor(1290, 128, 256, 6e-1)

cuda = torch.device('cuda')
ddpg = recnn.nn.DDPG(policy_net, value_net)
ddpg = ddpq.to(cuda)
plotter = recnn.utils.Plotter(ddpg.loss_layout, [['value', 'policy']],)

from IPython.display import clear_output
import matplotlib.pyplot as plt
%matplotlib inline

plot_every = 3
n_epochs = 2

def learn():
    for epoch in range(n_epochs):
        for batch in tqdm(env.train_dataloader):
            loss = ddpq.update(batch, learn=True)
            plotter.log_losses(loss)
            ddpq.step()
            if ddpq._step % plot_every == 0:
                clear_output(True)
                print('step', ddpq._step)
                test_loss = run_tests()
                plotter.log_losses(test_loss, test=True)

```

(continues on next page)

```

        plotter.plot_loss()
    if ddpq._step > 100:
        return

learn()

```

2.1.3 Using Pandas Backends

RecNN supports different types of pandas backends for faster data loading/processing in and out of core

Pandas is your default backend:

```

# but you can also set it directly:
recnn.pd.set("pandas")
frame_size = 10
batch_size = 25
dirs = recnn.data.env.DataPath(
    base="../../../data/",
    embeddings="embeddings/ml20_pca128.pkl",
    ratings="ml-20m/ratings.csv",
    cache="cache/frame_env.pkl", # cache will generate after you run
    use_cache=False # disable for testing purposes
)

%%time
env = recnn.data.env.FrameEnv(dirs, frame_size, batch_size)

# Output:
100%|| 20000263/20000263 [00:13<00:00, 1469488.15it/s]
100%|| 20000263/20000263 [00:15<00:00, 1265183.17it/s]
100%|| 138493/138493 [00:06<00:00, 19935.53it/s]
CPU times: user 41.6 s, sys: 1.89 s, total: 43.5 s
Wall time: 43.5 s

```

IP.S. nstall Modin [here](#) , it is not installed via RecNN's deps

You can also use modin with Dask / Ray.

Here is a little Ray example:

```

import os
import ray

if ray.is_initialized():
    ray.shutdown()
os.environ["MODIN_ENGINE"] = "ray" # Modin will use Ray
ray.init(num_cpus=10) # adjust for your liking
recnn.pd.set("modin")
%%time
env = recnn.data.env.FrameEnv(dirs, frame_size, batch_size)

100%|| 138493/138493 [00:07<00:00, 18503.97it/s]
CPU times: user 12 s, sys: 2.06 s, total: 14 s
Wall time: 21.4 s

```

Using Dask:


```

### dask
import os
os.environ["MODIN_ENGINE"] = "dask" # Modin will use Dask
recnn.pd.set("modin")
%%time
env = recnn.data.env.FrameEnv(dirs, frame_size, batch_size)

100%| 138493/138493 [00:06<00:00, 19785.99it/s]
CPU times: user 14.2 s, sys: 2.13 s, total: 16.3 s
Wall time: 22 s
<recnn.data.env.FrameEnv at 0x7f623fb30250>

```

Free 2x improvement in loading speed

2.2 NN

2.2.1 Models

class recnn.nn.models.**Actor** (*input_dim, action_dim, hidden_size, init_w=0.2*)

Vanilla actor. Takes state as an argument, returns action.

forward (*state, tanh=False*)

Parameters

- **action** – nothing should be provided here.
- **state** – state
- **tanh** – whether to use tahn as action activation

Returns action

class recnn.nn.models.**AnomalyDetector**

Anomaly detector used for debugging. Basically an auto encoder. P.S. You need to use different weights for different embeddings.

class recnn.nn.models.**Critic** (*input_dim, action_dim, hidden_size, init_w=3e-05*)

Vanilla critic. Takes state and action as an argument, returns value.

class recnn.nn.models.**DiscreteActor** (*input_dim, action_dim, hidden_size, init_w=0*)

forward (*inputs*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class recnn.nn.models.**bcqGenerator** (*state_dim, action_dim, latent_dim*)

Batch constrained generator. Basically VAE

class recnn.nn.models.**bcqPerturbator** (*num_inputs, num_actions, hidden_size, init_w=0.3*)

Batch constrained perturbative actor. Takes action as an argument, adjusts it.

2.2.2 Update

`recnn.nn.update.ddpg.ddpg_update` (*batch*, *params*, *nets*, *optimizer*, *device=device(type='cpu')*, *debug=None*, *writer=<recnn.utils.misc.DummyWriter object>*, *learn=False*, *step=-1*)

Parameters

- **batch** – batch [state, action, reward, next_state] returned by environment.
- **params** – dict of algorithm parameters.
- **nets** – dict of networks.
- **optimizer** – dict of optimizers
- **device** – torch.device
- **debug** – dictionary where debug data about actions is saved
- **writer** – torch.SummaryWriter
- **learn** – whether to learn on this step (used for testing)
- **step** – integer step for policy update

Returns loss dictionary

How parameters should look like:

```
params = {
    'gamma'      : 0.99,
    'min_value'  : -10,
    'max_value'  : 10,
    'policy_step': 3,
    'soft_tau'   : 0.001,
    'policy_lr'  : 1e-5,
    'value_lr'   : 1e-5,
    'actor_weight_init': 3e-1,
    'critic_weight_init': 6e-1,
}
nets = {
    'value_net': models.Critic,
    'target_value_net': models.Critic,
    'policy_net': models.Actor,
    'target_policy_net': models.Actor,
}
optimizer = {
    'policy_optimizer': some_optimizer
    'value_optimizer': some_optimizer
}
```

`recnn.nn.update.td3.td3_update` (*batch*, *params*, *nets*, *optimizer*, *device=device(type='cpu')*, *debug=None*, *writer=<recnn.utils.misc.DummyWriter object>*, *learn=False*, *step=-1*)

Parameters

- **batch** – batch [state, action, reward, next_state] returned by environment.
- **params** – dict of algorithm parameters.
- **nets** – dict of networks.
- **optimizer** – dict of optimizers

- **device** – torch.device
- **debug** – dictionary where debug data about actions is saved
- **writer** – torch.SummaryWriter
- **learn** – whether to learn on this step (used for testing)
- **step** – integer step for policy update

Returns loss dictionary

How parameters should look like:

```
params = {
    'gamma': 0.99,
    'noise_std': 0.5,
    'noise_clip': 3,
    'soft_tau': 0.001,
    'policy_update': 10,

    'policy_lr': 1e-5,
    'value_lr': 1e-5,

    'actor_weight_init': 25e-2,
    'critic_weight_init': 6e-1,
}

nets = {
    'value_net1': models.Critic,
    'target_value_net1': models.Critic,
    'value_net2': models.Critic,
    'target_value_net2': models.Critic,
    'policy_net': models.Actor,
    'target_policy_net': models.Actor,
}

optimizer = {
    'policy_optimizer': some_optimizer
    'value_optimizer1': some_optimizer
    'value_optimizer2': some_optimizer
}
```

`recnn.nn.update.bcq.bcq_update` (*batch*, *params*, *nets*, *optimizer*, *device=device(type='cpu')*,
debug=None, *writer=<recnn.utils.misc.DummyWriter object>*,
learn=False, *step=-1*)

Parameters

- **batch** – batch [state, action, reward, next_state] returned by environment.
- **params** – dict of algorithm parameters.
- **nets** – dict of networks.
- **optimizer** – dict of optimizers
- **device** – torch.device
- **debug** – dictionary where debug data about actions is saved
- **writer** – torch.SummaryWriter

- **learn** – whether to learn on this step (used for testing)
- **step** – integer step for policy update

Returns loss dictionary

How parameters should look like:

```
params = {
    # algorithm parameters
    'gamma'           : 0.99,
    'soft_tau'        : 0.001,
    'n_generator_samples': 10,
    'perturbator_step' : 30,

    # learning rates
    'perturbator_lr'  : 1e-5,
    'value_lr'        : 1e-5,
    'generator_lr'    : 1e-3,
}

nets = {
    'generator_net': models.bcqGenerator,
    'perturbator_net': models.bcqPerturbator,
    'target_perturbator_net': models.bcqPerturbator,
    'value_net1': models.Critic,
    'target_value_net1': models.Critic,
    'value_net2': models.Critic,
    'target_value_net2': models.Critic,
}

optimizer = {
    'generator_optimizer': some optimizer
    'policy_optimizer': some optimizer
    'value_optimizer1': some optimizer
    'value_optimizer2': some optimizer
}
```

`recnn.nn.update.misc.value_update` (*batch, params, nets, optimizer, device=device(type='cpu'), debug=None, writer=<recnn.utils.misc.DummyWriter object>, learn=False, step=-1*)

Everything is the same as in `ddpg_update`

2.2.3 Algo

2.3 Data

This module contains things to work with datasets. At the moment, utils are pretty messy and will be rewritten.

2.3.1 env

Main abstraction of the library for datasets is called environment, similar to how other reinforcement learning libraries name it. This interface is created to provide SARSA like input for your RL Models. When you are working with recommendation env, you have two choices: using static length inputs (say 10 items) or dynamic length time series

with sequential encoders (many to one rnn). Static length is provided via FrameEnv, and dynamic length along with sequential state representation encoder is implemented in SeqEnv. Let's take a look at FrameEnv first:

```
class recnn.data.env.DataPath (base: str, ratings: str, embeddings: str, cache: str = "", use_cache:
                                bool = True)
```

[New!] Path to your data. Note: cache is optional. It saves EnvBase as a pickle

```
class recnn.data.env.Env (path: recnn.data.env.DataPath, prepare_dataset=<function prepare_dataset>,
                           embed_batch=<function batch_tensor_embeddings>,
                           **kwargs)
```

Env abstract class

```
class recnn.data.env.EnvBase
```

Misc class used for serializing

```
class recnn.data.env.FrameEnv (path, frame_size=10, batch_size=25, num_workers=1, *args,
                               **kwargs)
```

Static length user environment.

```
test_batch ()
```

Get batch for testing

```
train_batch ()
```

Get batch for training

```
class recnn.data.env.UserDataset (users, user_dict)
```

Low Level API: dataset class user: [items, ratings], Instance of torch.DataSet

Reference

```
class recnn.data.env.UserDataset (users, user_dict)
```

Low Level API: dataset class user: [items, ratings], Instance of torch.DataSet

```
__getitem__ (idx)
```

getitem is a function where non linear user_id maps to a linear index. For instance in the ml20m dataset, there are big gaps between neighbouring user_id. getitem removes these gaps, optimizing the speed.

Parameters **idx** (*int*) – index drawn from range(0, len(self.users)). User id can be not linear, idx is.

Returns dict{'items': list<int>, rates:list<int>, sizes: int}

```
__init__ (users, user_dict)
```

Parameters

- **users** (*list<int>.*) – integer list of user_id. Useful for train/test splitting
- **user_dict** ((dict{ user_id<int>: dict{'items': list<int>, 'ratings': list<int>} }))) – dictionary of users with user_id as key and [items, ratings] as value

```
__len__ ()
```

useful for tqdm, consists of a single line: return len(self.users)

```
class recnn.data.env.Env (path: recnn.data.env.DataPath, prepare_dataset=<function prepare_dataset>,
                           embed_batch=<function batch_tensor_embeddings>,
                           **kwargs)
```

Env abstract class

```
__init__ (path: recnn.data.env.DataPath, prepare_dataset=<function prepare_dataset>, embed_batch=<function batch_tensor_embeddings>, **kwargs)
```

Note: embeddings need to be provided in {movie_id: torch.tensor} format!

Parameters

- **path** (*DataPath*) – DataPath to where item embeddings are stored.
- **test_size** (*int*) – ratio of users to use in testing. Rest will be used for training/validation
- **min_seq_size** (*int*) – (use as kwarg) filter users: len(user.items) > min seq size
- **prepare_dataset** (*function*) – (use as kwarg) function you provide.
- **embed_batch** (*function*) – function to apply embeddings to batch. Can be set to yield continuous/discrete state/action

```
class recnn.data.env.FrameEnv(path, frame_size=10, batch_size=25, num_workers=1, *args,
                               **kwargs)
```

Static length user environment.

```
__init__(path, frame_size=10, batch_size=25, num_workers=1, *args, **kwargs)
```

Parameters

- **embeddings** (*str*) – path to where item embeddings are stored.
- **ratings** (*str*) – path to the dataset that is similar to the ml20m
- **frame_size** (*int*) – len of a static sequence, frame

p.s. you can also provide ****pandas_conf** in the arguments.

It is useful if you dataset columns are different from ml20:

```
pandas_conf = {user_id='userId', rating='rating', item='movieId', timestamp=
↪ 'timestamp'}
env = FrameEnv(embed_dir, rating_dir, **pandas_conf)
```

```
test_batch()
```

Get batch for testing

```
train_batch()
```

Get batch for training

2.3.2 dataset_functions

What?

RecNN is designed to work with your data flow.

Set kwargs in the beginning of prepare_dataset function. Kwargs you set are immutable.

args_mut are mutable arguments, you can access the following: base: data.EnvBase, df: DataFrame, users: List[int], user_dict: Dict[int, Dict[str, np.ndarray]]

Access args_mut and modify them in functions defined by you. Best to use function chaining with build_data_pipeline.

recnn.data.prepare_dataset is a function that is used by default in Env.__init__ But sometimes you want some extra. I have also predefined truncate_dataset. This function truncates the number of items to specified one. In reinforce example I modify it to look like:

```

def prepare_dataset(args_mut, kwargs):
    kwargs.set('reduce_items_to', num_items) # set kwargs for your functions here!
    pipeline = [recnn.data.truncate_dataset, recnn.data.prepare_dataset]
    recnn.data.build_data_pipeline(pipeline, kwargs, args_mut)

# embeddings: https://drive.google.com/open?id=1EQ_zXBR3DKpmJR3jBgLvt-xoOvArGMSL
env = recnn.data.env.FrameEnv('.',
                              '...', frame_size, batch_size,
                              embed_batch=embed_batch, prepare_dataset=prepare_dataset,
                              num_workers=0)

```

recnn.data.dataset_functions.**build_data_pipeline** (*chain: List[Callable], kwargs: recnn.data.dataset_functions.DataFuncKwargs, args_mut: recnn.data.dataset_functions.DataFuncArgsMut*)

Higher order function :param chain: array of callable :param ****kwargs**: any kwargs you like

recnn.data.dataset_functions.**prepare_dataset** (*args_mut: recnn.data.dataset_functions.DataFuncArgsMut, kwargs: recnn.data.dataset_functions.DataFuncKwargs*)

Basic prepare dataset function. Automatically makes index linear, in ml20 movie indices look like: [1, 34, 123, 2000], recnn makes it look like [0,1,2,3] for you.

recnn.data.dataset_functions.**truncate_dataset** (*args_mut: recnn.data.dataset_functions.DataFuncArgsMut, kwargs: recnn.data.dataset_functions.DataFuncKwargs*)

Truncate #items to reduce_items_to provided in kwargs

CHAPTER 3

Indices and tables

- `genindex`
- `modindex`
- `search`

r

`recnn.data.dataset_functions`, 19
`recnn.data.env`, 17
`recnn.nn.algo`, 16
`recnn.nn.models`, 13
`recnn.nn.update.bcq`, 15
`recnn.nn.update.ddpg`, 14
`recnn.nn.update.misc`, 16
`recnn.nn.update.td3`, 14

Symbols

`__getitem__()` (*recnn.data.env.UserDataset method*), 17
`__init__()` (*recnn.data.env.Env method*), 17
`__init__()` (*recnn.data.env.FrameEnv method*), 18
`__init__()` (*recnn.data.env.UserDataset method*), 17
`__len__()` (*recnn.data.env.UserDataset method*), 17

A

Actor (*class in recnn.nn.models*), 13
 AnomalyDetector (*class in recnn.nn.models*), 13

B

`bcq_update()` (*in module recnn.nn.update.bcq*), 15
 bcqGenerator (*class in recnn.nn.models*), 13
 bcqPerturbator (*class in recnn.nn.models*), 13
`build_data_pipeline()` (*in module recnn.data.dataset_functions*), 19

C

Critic (*class in recnn.nn.models*), 13

D

DataPath (*class in recnn.data.env*), 17
`ddpg_update()` (*in module recnn.nn.update.ddpg*), 14
 DiscreteActor (*class in recnn.nn.models*), 13

E

Env (*class in recnn.data.env*), 17
 EnvBase (*class in recnn.data.env*), 17

F

`forward()` (*recnn.nn.models.Actor method*), 13
`forward()` (*recnn.nn.models.DiscreteActor method*), 13
 FrameEnv (*class in recnn.data.env*), 17, 18

P

`prepare_dataset()` (*in module recnn.data.dataset_functions*), 19

R

`recnn.data.dataset_functions` (*module*), 19
`recnn.data.env` (*module*), 17
`recnn.nn.algo` (*module*), 16
`recnn.nn.models` (*module*), 13
`recnn.nn.update.bcq` (*module*), 15
`recnn.nn.update.ddpg` (*module*), 14
`recnn.nn.update.misc` (*module*), 16
`recnn.nn.update.td3` (*module*), 14

T

`td3_update()` (*in module recnn.nn.update.td3*), 14
`test_batch()` (*recnn.data.env.FrameEnv method*), 17, 18
`train_batch()` (*recnn.data.env.FrameEnv method*), 17, 18
`truncate_dataset()` (*in module recnn.data.dataset_functions*), 19

U

UserDataset (*class in recnn.data.env*), 17

V

`value_update()` (*in module recnn.nn.update.misc*), 16