

# RBatchGenerator()

### Uproot+Awkward

### What are the problems?

 required more memory to test (maximum try 50 files for 15 GB RAM)

### Next step to try!

- save each file to numpy file
- try to use torch dataloader and load from dir.
- compare speed of training and memory using in each approach.

### •••

1	<pre>def load dataset(data path='/home/northnpk/Downloads/JetClass dataset/',</pre>
2	tree name = "tree",
3	batch size = 128,
4	max_num_particles=128,
5	chunk_size = 5_000,
6	<pre>vec_columns=['part_px', 'part_py', 'part_pz', 'part_energy'],</pre>
7	columns=['jet_pt', 'jet_eta', 'jet_phi', 'jet_energy'],
8	<pre>targets = ['label_QCD', 'label_Hbb', 'label_Hcc', 'label_Hgg'</pre>
9	'label_Hqql', 'label_Zqq', 'label_Wqq', 'label_Tbo
10	# Passing vector columns
11	<pre>max_vec_sizes = dict(zip(vec_columns, [max_num_particles]*len(vec_columns)</pre>
12	
13	# get files path
14	train_path = [data_path + 'train/' + p for p in os.listdir(data_path + 't
15	
16	# load RDataFrame from path
17	<pre>r_df = RDataFrame(tree_name, train_path[:10]) # Catting DBatabCanagatan from true DBatabCanagatan</pre>
18 19	<pre># Getting RBatchGenerator from tmva RBatchGenerator gen train = R00T.TMVA.Experimental.CreatePyTorchGenerators(r df,</pre>
20	<pre>gen_train = koor.nww.cxperimentat.createryrorchoenerators(r_dr, batch size,</pre>
20	chunk_size,
22	columns=columns
23	max vec sizes=n
24	target=targets,
25	validation_spli
26	drop remainder=
27	
28	# load Generator to torch dataloader
29	dataset = RBatchDataset(gen_train)
30	return DataLoader(dataset=dataset, collate_fn=dataset.collate_fn, batch_si

```
s+vec_columns+targets,
hax_vec_sizes,
t=0,
False)
.ze=None)
```

```
ain')]
```

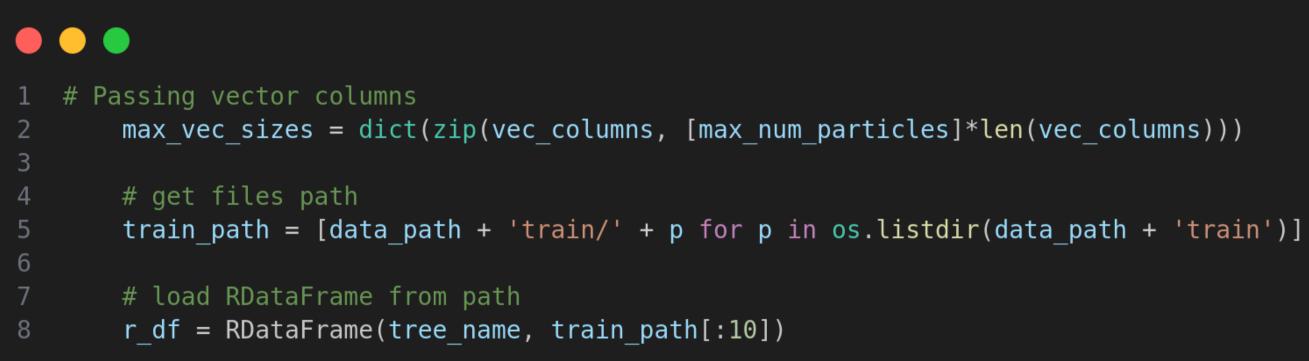
'label H4q',

```
q', 'label_Tbl']) -> DataLoader:
))
```

### •••

1	<pre># Define load_dataset method to loading TMVA RBatchGenerator into Torch Da</pre>
2	<pre>def load_dataset(data_path='/home/northnpk/Downloads/JetClass_dataset/',</pre>
3	<pre>tree_name = "tree",</pre>
4	batch_size = 128,
5	<pre>max_num_particles=128,</pre>
6	chunk_size = 5_000,
7	<pre>vec_columns=['part_px', 'part_py', 'part_pz', 'part_energ</pre>
8	columns=['jet_pt', 'jet_eta', 'jet_phi', 'jet_energy'],
9	<pre>targets = ['label_QCD', 'label_Hbb', 'label_Hcc', 'label_</pre>
10	'label_Hqql', 'label_Zqq', 'label_Wqq', 'label

ataloader gy'], \_Hgg', 'label\_H4q', l\_Tbqq', 'label\_Tbl']) -> DataLoader:



### • • • 1 # Getting RBatchGenerator from tmva RBatchGenerator gen train = ROOT.TMVA.Experimental.CreatePyTorchGenerators(r df, 3 5 6 8 10 11 # load Generator to torch dataloader dataset = RBatchDataset(gen train) 12 return DataLoader(dataset=dataset, collate fn=dataset.collate fn, batch size=None) 13

```
batch size,
chunk size,
columns=columns+vec_columns+targets,
max vec sizes=max vec sizes,
target=targets,
validation split=0,
drop remainder=False)
```

## RBatchDataset(Iterabledataset)

### •••

	U. D. Charles Theorem 1. Destance from TNUA DD-to be an entropy
1	# Define Torch IterableDataset for TMVA RBatchGenerator
2	class RBatchDataset(IterableDataset):
3	<pre>definit(self, generator, vector_prep_fn=None):</pre>
	self.generator = generator
5	<pre>self.batch_size = self.generator.base_generator.batch_size</pre>
6	<pre>self.vector_prep_fn = vector_prep_fn</pre>
7	<pre>self.apply_vector_prep_fn = False</pre>
8	<pre>if self.vector_prep_fn != None:</pre>
9	<pre>self.apply_vector_prep_fn = True</pre>
10	<pre># print(f'Batch size: {self.batch_size}')</pre>
11	if self.generator.last_batch_no_of_rows != 0 and self.generator.number_of_batches > 1:
12	self.length = ((self.generator.number_of_batches-1) * self.batch_size) + self.generator.last_k
13	elif self.generator.number_of_batches == 1:
14	self.length = self.generator.last_batch_no_of_rows
15	else :
16	self.length = (self.generator.number_of_batches * self.batch_size)
17	
18	<pre>self.vec_columns = [c not in self.generator.base_generator.given_columns for c in self.generator.b</pre>
19	<pre>selfcolumns = [c in self.generator.base_generator.given_columns for c in self.generator.base_gene</pre>
20	<pre>self.label_columns = self.generator.base_generator.target_columns</pre>
21	<pre>self.vec_names = []</pre>
22	for n in self.generator.base_generator.given_columns:
23	if n not in self.label_columns and n not in self.generator.base_generator.train_columns:
24	<pre>self.vec_names.append(n)</pre>
25	
26	def collate fn(self, data):
27	tensors, targets = data
28	if self.apply vector prep fn :
29	return self.vector_prep_fn(tensors[:, self.vec_columns].reshape(len(tensors), 4, 128), self.vec
30	
31	else :
32	return tensors[:, self.vec columns].reshape(len(tensors), 4, 128), tensors[:, self. columns],
33	
34	def iter (self):
35	return self.generator. iter ()
36	
37	def len (self):
38	return self.generator.number of batches
39	

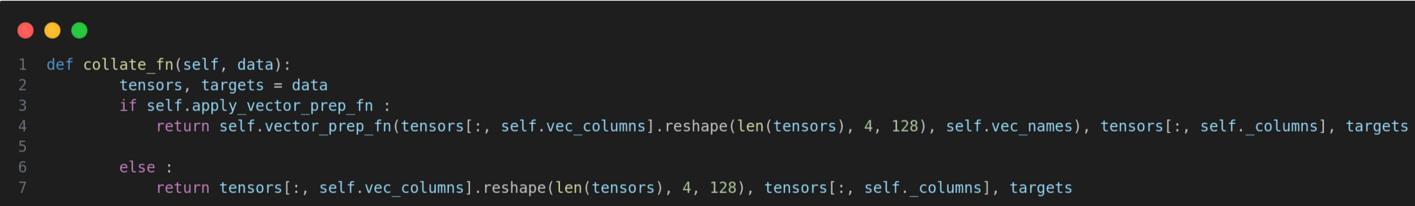
\_batch\_no\_of\_rows

.base\_generator.train\_columns] enerator.train\_columns]

vec\_names), tensors[:, self.\_columns], targets

, targets

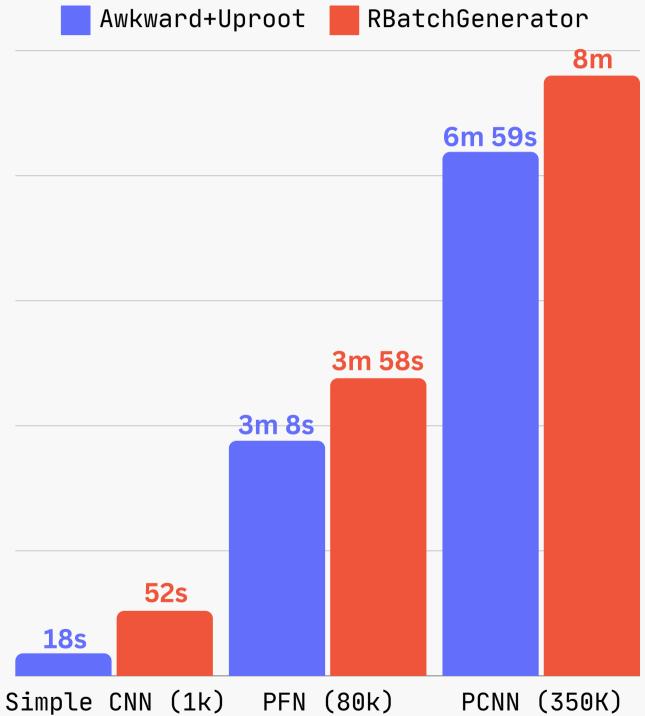
## RBatchDataset(Iterabledataset)



## RBatchGenerator()

training time with Pytorch	500s
CPU only for 1 epoch	
(Lower : Better)	400s
• 10 files	300s —
• batch_size : 128	5005
• dataset : JetClass	200s -

100s



## **RBatchGenerator()**

Next step to try!

- profiling with perf, memray to see What, Where, When, and How the memory use.
- Training Particle Transformers with **RBatchGenerator**



# Thank you

