

Application of Particle Net

Ting-Kai Hsu

Contents

1	ParticleNet Structure	2
1.1	Notification	2
1.2	stack_arrays	2
1.3	pad_arrays	3
1.4	Dataset.__init__()	5
1.5	Dataset.load()	6
1.6	Dataset.__len__()	8
1.7	Dataset.__getitem__(self, key)	9
1.8	What is @property	9
1.9	Dataset-X(self) and y(self)	10
1.10	Dataset-shuffle(self, seed=None)	10
1.11	Transforming Data	11
1.12	channel_last	11
1.13	Importing model	11
1.14	Inputing the model	12
1.15	Setting Training Parameters	12
1.16	lr_shedule	13
1.17	Model Information	14
1.18	Prepare Model and Its Saving Directory	14
1.19	Training	15
2	Appendices	16

Chapter 1

ParticleNet Structure

1.1 Notification

Note that the requirement for running Particle Net is important. One thing to note that the version of **numpy** should be below 1.14.1, and professor suggests to run the program on Linux system.

1.2 stack_arrays

First, it has defined two functions dealing with the data format,

- stack_arrays
- pad_arrays

For the first function, the definition code would be,

```
1 def stack_arrays(a, keys, axis=-1):
2     flat_arr = np.stack([a[k].flatten() for k in keys],
3                           axis=axis)
4     return
5     awkward.JaggedArray.fromcounts(a[keys[0]].counts,
6                                    flat_arr)
```

For the first, we have to know what is **np.stack()**, it is a function that can reconstruct some arrays and then combine (stack) them. The parameter **axis** would

be the orientation of the separation. In this case, **axis = -1**, which means the array would construct along their last axis, like below.

```
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.193
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> arr1 = np.array([[1,2,3],[4,5,6]])
>>> arr2 = np.array([[7,8,9],[10,11,12]])
>>> stacked_arr = np.stack([arr1, arr2], axis = -1)
>>> print(stacked_arr)
[[[ 1  7]
   [ 2  8]
   [ 3  9]]

  [[ 4 10]
   [ 5 11]
   [ 6 12]]]
>>> |
```

Figure 1.1: Example for `np.stack([,], axis = -1)`

Also, the function **flatten()** would flatten the array into 1 dimension array. Let's move on to the line 3, where we use the package **awkward**.

- **a[keys[0]].counts**: this is the first argument of the function call. It would define the structure of the jagged array. Counts function would specify the number of elements in the suggested array.
- **flat_arr**: this is the stack array created in line 2, and this argument would provide the data for the jagged array.

For example, if **a[keys[0]].counts** is [3, 2, 4], it means the JaggedArray will consist of three sublists, each with a different number of elements: 3, 2, and 4. The data in **flat_arr** will be distributed into these sublists based on this structure.

This is a common operation when working with structured data, such as particle physics data, where events may have varying numbers of particles, and you need to represent the data in a structured, variable-length format.

1.3 pad_arrays

```
1 def pad_array(a, maxlen, value=0., dtype='float32'):  
2     x = (np.ones((len(a), maxlen)) * value).astype(dtype)  
3     for idx, s in enumerate(a):  
4         if not len(s):  
5             continue  
6         trunc = s[:maxlen].astype(dtype)  
7         x[idx, :len(trunc)] = trunc  
8     return x
```

First of all, we should explain what is the term 'padding' means. Padding is a common operation that we adjust the length of sequences to make them uniform. In our code, we use this function that takes a list of sequences to pad them with a maximum limitation to ensure them to have same length. Padding is a pre-process that do the data to the ML model which requires the inputs to have same dimensions. Here we explain the code line by line, For inputs, we have

- **a**: The datatype would be array, which would provide the data that we should process.
- **maxlen**: The datatype would be integer, which would limit the length of the padded array.
- **value=0**: The datatype would be integer, and it would determine the initial content of padding with default value 0.
- **dtype='float32'**: This would determine the datatype stored in the array.

Next we initialize the padded array,

```
1 x = (np.ones((len(a), maxlen)) * value).astype(dtype)
```

We create a Numpy array **x** with dimension **(len(a), maxlen)** and then fill it with all zero. Note that the elements become zeros because the default value of **value** is zero, and **np.ones()** provides us flexibility to adjust the content of the array rather than making them all zero by **np.zeros()**. Then we make the datatype of the elements become float32. Next, let's move on to the for loop. In the for loop, we go through the input list of sequences **a**, and **idx** would be the index of sequence, and **s** would be the sequence itself. First, we would check whether the sequence **s** is empty or not. If is empty, then the loop would skip and go on to the next loop. If is not empty, then we see line 6 and 7 together.

```
1 trunc = s[:maxlen].astype(dtype)
2 x[idx, :len(trunc)] = trunc
```

First **trunc** means that we only take part of information of **s**, that is, we truncate the sequence for a certain length **maxlen**. For sequences that are shorter than **maxlen**, we pad the remaining part in **x** by **value**.

1.4 Dataset-__init__()

Dataset has been separated into many parts, first we would introduce a function called **__init__**.

```
1 def __init__(self, filepath, feature_dict = {},
2             label='label', pad_len=100, dataformat='channel_first'):
3     self.filepath = filepath
4     self.feature_dict = feature_dict
5     if len(feature_dict)==0:
6         feature_dict['points'] = ['part_etarel',
7                                   'part_phirel']
8         feature_dict['feature'] = ['part_pt_log',
9                                    'part_e_log', 'part_etarel', 'part_phirel']
10        feature_dict['mask'] = ['part_pt_log']
11    self.label = label
12    self.pad_len = pad_len
13    assert data_format in ('channel_first', 'channel_last')
14    self.stack_axis = 1 if data_format=='channel_first'
15                        else -1
16    self._values = {}
17    self._label = None
18    self._load()
```

This function is obviously initialize the object with some certain inputs.

Inputs:

- **self**: This is the parameter calling the object itself.
- **filepath**: This is the parameter that the data file comes from. It is associated with the instance variable **self.filepath** which store the path of the data file.

- **self.feature_dict = feature_dict**: This variable is used to store a dictionary¹ of features.

And then we go into a loop that initializes **feature_dict**. First, we have to check whether the dictionary is empty or not, if not, we must avoid to overwrite the content; if so, we would do some works as shown below,

- **feature_dict['points'] = ['part_etarel', 'part_phirel']**: We assign a list of strings to a key **'points'**, and this may be the information (coordinates) of the particle with angular information.
- **feature_dict['feature'] = ['part_pt_log', 'part_e_log', 'part_etarel', 'part_phirel']**: We assign the key **feature** to the some physical features² of the particle. In this case, we only provide some basic data, that is, the transverse momentum of the particle along the detector axis, energy of the particle, and the position of the particle we have mentioned previously.
- **feature_dict['mask'] = ['part_pt_log']**: This needs verification.
- **stack_axis**: This would tell us how transform the initial data format (may be channels_first) to channels_last by applying `numpy.stack()`. How does this affect to the data would ne

Next we need to review the meaning and function of **assert**. **assert** function is used to check at some certain time, what is the situation of the system. If the situation of assertion is different with the situation of the system, the computer would print out the error message. ³ So line 10 check whehter **dataformat** is in **'channel_first'** or **'channel_last'**.

At the last line of this function, that is, line 14, the function calls another function called **load()**, which we would explain next.

1.5 Dataset-load()

After initializing the dataclass object, we now go through the function that reads the data file.

¹Dictionary in Python is like map in C++, where there is key one-one to the data, and we can use the key to find the desired data.

²I don't know why but we neglect the information of charge carried by the particle here.

³Check this

```

1 def _load(self):
2     logging.info('Start loading file %s' % self.filepath)
3     counts = None
4     with awkward.load(self.filepath) as a:
5         self._label = a[self.label]
6         for k in self.feature_dict:
7             cols = self.feature_dict[k]
8             if not isinstance(cols, (list, tuple)):
9                 cols = [cols]
10            arrs = []
11            for col in cols:
12                if counts is None:
13                    counts = a[col].counts
14                else:
15                    assert np.array_equal(counts, a[col].counts)
16                    arrs.append(pad_array(a[col], self.pad_len))
17            self._values[k] = np.stack(arrs,
18                                     axis=self.stack_axis)
19     logging.info('Finished loading file %s' % self.filepath)

```

The first two lines,

```

1 logging.info('Start loading file %s' % self.filepath)
2 counts = None

```

logging shows the function would load the information of the file whose position is given by **self.filepath**. And then set the variable **counts** to **None**.

Then we open the file with function **with** to make sure the file would close after being used. We call the function **awkward.load(self.filepath)** into a variable **a**. In **__init__()** we assign **None** to the variable **self._label**.

However, the difference between **self.label** and **self._label**.

```

1     self._label = a[self.label]

```

In the code, it is clear to see that **self._label** would be the value corresponding to the **key** of **self.label**.⁴

```

1 for k in self.feature_dict:

```

⁴For more information of letting a variable to be load() function, see this.

```

2     cols = self.feature_dict[k]
3     if not isinstance(cols, (list, tuple)):
4         cols = [cols]
5     arrs = []

```

Now we iterate the keys stored in **self.feature_dict** dictionary. If the datatype of the value for a certain is not **list** or **tuple**⁵, then we convert **cols** to a list with single elements. Line 5 would initialize a variable called **arrs**.

We then go through a contained for loop, which loops through the contents of the variable **cols** which is the value correspond to key of feature dictionary.

```

1     for col in cols:
2         if counts is None:
3             counts = a[col].counts
4         else:
5             assert np.array_equal(counts, a[col].counts)
6             arrs.append(pad_array(a[col], self.pad_len))

```

If **counts** remains its initial value, then we set it to be the correct value we wish it to be, that is, **a[col].counts**. Otherwise, we check if it is equal to **a[col].counts**, and then we append *the padded version of a[col]* to **arrs**.

```

1 self._values[k] = np.stack(arrs, axis=self.stack_axis)

```

Then we arrange the layout of **arrs** by calling **np.stack()**, and put it back into **self._values[k]**, which we now see that it stores the revised data.

The last line would show the completion of loading the file at **filepath**.

1.6 Dataset-__len__()

The code would be,

```

1 def __len__(self):
2     return len(self._label)

```

In **__init__** or **load(self)**, we can guess the datatype of **self.label** would be string, thus in this function we return its length. The reason should be shown later when we need to **randomly shuffle the order of training data**.

⁵For the use of function **isinstance()**, check this

1.7 Dataset-__getitem__(self, key)

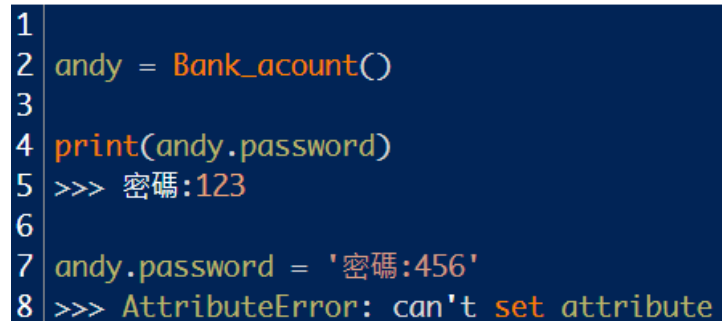
```
1 def __getitem__(self, key):
2     if key==self.label:
3         return self._label
4     elss:
5         return self._values[key]
```

In this peice of code we can see the difference between **label** and **_values[]**. **label** is used to identify where the data belongs to, and **_values[]** would be the data. This function merely combines two calling contents function into one, that is, if **key** is equal to **self.label**, that means we like to use this function to get the content of **self.label**, else otherwise.

1.8 What is @property

Before going into next peice of code in **class Dataset**, we must learn what is so-called @property in python.

- Turning the methods in class into **read-only** property.



```
1
2 andy = Bank_account()
3
4 print(andy.password)
5 >>> 密碼:123
6
7 andy.password = '密碼:456'
8 >>> AttributeError: can't set attribute
```

Figure 1.2: Example Code

- There should be **getter**, **setter**, and **deleter**, if one would like to modify the data of **property** function.

If one touched C++ first like me, think about the difference of **private** and **public**.

1.9 Dataset-X(self) and y(self)

```
1 @property
2 def X(self):
3     return self._values
4
5 @property
6 def y(self):
7     return self._label
```

Now we can see what `_values[]` and `_label` stand for. Recall that the function of ParticleNet would be labeling and identifying particles. Then we should recognize that `_label` here represents the label of training data, and `_values[]` would be the original data got from detector. These shouldn't be able to be modified by the model during the training process, thus we use `@property` to protect them.

1.10 Dataset-shuffle(self, seed=None)

```
1 def shuffle(self, seed=None):
2     if seed is not None:
3         np.random.seed(seed)
4     shuffle_indices = np.arange(self.__len__())
5     np.random.shuffle(shuffle_indices)
6     for k in self._values:
7         self._values[k] = self._values[k][shuffle_indices]
8     self._label = self._label[shuffle_indices]
```

Line 2 to 3, we should make sure that `seed` isn't empty, if so we should set it to be Numpy random seed⁶.

Line 4, it creates an array of indices from 0 to `self.__len__()`⁷ using `np.arange`. This array represents the indices of the elements to be shuffled, that is, in this function, we shuffle the order of training data numbered by `_label`.

Line 5, shuffle the array effectively, arranging the values to indice indicated

⁶`numpy.random.seed` function is used to initialize the random number generator with a seed. Seed ensures *reproducibility* of random number generation, that is, using the same seed would result in the same sequence of random number.

⁷Recall that `self.__len__()` is the function that measures the length of `_label`.

by **shuffle.indices**.

Line 6 to 7, going into for loop, it iterates the indices of **self._values**. Recall that it is a dictionary whose value would be **stacked array**. Thus we go into the for loop and shuffle the content of **_values** for each key according to **shuffle.indices**.

Line 8, we shuffle **self.label** according to **shuffle.indices**.

1.11 Transforming Data

```
1 train_dataset = Dataset('converted/train_file_0.awkd',
    data_format='channel_last')
2 val_dataset = Dataset('converted/val_file_0.awkd',
    data_format='channel_last')
```

First, we should know what **channel last** is. As implied by professor, it is the data format used in Numpy Array.

1.12 channel_last

This data format is a convention in operating image in deep learning, and there is an opposite data format, that is, **channel first**. The former more often appear in implementation. We knew that typically, an image can be stored in three-dimension array. Rows are for height, and columns are for width, and the remaining ones are for channel⁸.

- **Channels last:** [rows][cols][channels]
- **Channels first:** [channels][rows][cols]

We should know the order of storing data because we're not manipulating image. As appendix implies, **np.stack()** would preserve the data format.

1.13 Importing model

⁸Typically, there would be 3 channels, representing the colors, and one for blue, and one for red, and the other for yellow.

```
1 import tensorflow as tf
2 from tensorflow import keras
3 from tf_keras_model import get_particle_net,
  get_particle_lite
```

Note that the modulus package of ParticleNet model should be download locally and stored in the same folder in PC.

1.14 Inputing the model

```
1 model_type = 'particle_net_lite'
2 #choose between 'particle_net' and 'particle_net_lite'
3 num_classes = train_dataset.y.shape[1]
4 input_shapes = {k:train_dataset[k].shape[1:] for k in
  train_dataset.X}
5 if 'lite' in model_type:
6     model = get_particle_net_lite(num_classes, input_shapes)
7 else:
8     model = get_particle_net(num_classes, input_shapes)
```

I think the most confusing code would be line 3 and 4. Here are some explanation. Line 3, note that in the model, our purpose is to label the unknown particles based on their properties. We simulated the desired process and then make the simulated data into training data, that is, we now assume there would only be limited kinds of particles identified by label, which is stored in **train_dataset.y**. Thus, we classify the real data with same labels, that is line 3 doing, it equalizes the number of classes **num_classes** with the number of labels in training data, or simulated data.

Line 4 is setting the size of the inputs, which is correspond to training dataset values. Note that we ignore the first row of the training data, this is because of the convention in deep learning that this row would be preserved for **batch size** ⁹.

1.15 Setting Training Parameters

⁹This is the number of independent data sheet that is thrown into each training in each epoch.

```
1 #Training Parameters
2 batch_size = 1024 if 'lite' in model_type else 384
3 epochs = 30
```

1.16 lr_shedule

```
1 def lr_schedule(epoch):
2     lr = 1e-3
3     if epoch > 10:
4         lr *= 0.1
5     elif epoch > 20:
6         lr *= 0.01
7     logging.info('Learning rate: %f'%lr)
8     return lr
```

This is the function that stands for **learning schedule** for different epoch. As one can see in the function, the learning rate would slow down when the training process is almost done. The reason why the learning rate should slow down is due to several reasons.

- **Convergence improvement:** At the beginning of training, we set the high learning rate is because we want the model to converge faster. As the optimization process progresses and the model approaches a minimum or a good solution, reducing the learning rate prevents overshooting and helps the optimizer to fine-tune the model more delicately. This can lead to improved convergence and better generalization.
- **Escape local minima:** At high learning rate, the model soon converges to a minimum, but this minimum may be mere local minimum, so we should slow down the learning rate that help us learn more around that minimum.
- **Avoidance of Overshooting:** High learning rates might cause the optimization process to overshoot the minimum, leading to oscillations or divergence. Gradually reducing the learning rate helps prevent such overshooting as the optimization process progresses.

1.17 Model Information

```
1 model.compile(loss='categorical_crossentropy', optimizer =  
    keras.optimizers.Adam(learning_rate = lr_schedule(0)),  
    metrics = ['accuracy'])  
2 metrics=['accuracy']  
3 model.summary()
```

This piece of code setup the model using specified loss function, optimizer, and metrics.

1.18 Prepare Model and Its Saving Directory

```
1 # Prepare model model saving directory.  
2 import os  
3 save_dir = 'model_checkpoints'  
4 model_name = '%s_model.{epoch:03d}.h5' % model_type  
5 if not os.path.isdir(save_dir):  
6     os.makedirs(save_dir)  
7 filepath = os.path.join(save_dir, model_name)  
8  
9 # Prepare callbacks for model saving and for learning rate  
    adjustment.  
10 checkpoint =  
    keras.callbacks.ModelCheckpoint(filepath=filepath,  
        monitor='val_acc', verbose=1, save_best_only=True)  
11 lr_scheduler =  
    keras.callbacks.LearningRateScheduler(lr_schedule)  
12 progress_bar = keras.callbacks.ProgbarLogger()  
13 callbacks = [checkpoint, lr_scheduler, progress_bar]
```

This part would not be our focus.

1.19 Training

```
1 train_dataset.shuffle()
2 model.fit(train_dataset.X, train_dataset.y,
            batch_size=batch_size, epochs=epochs,
            validation_data=(val_dataset.X, val_dataset.y),
            shuffle=True, callbacks=callbacks)
```

One can see that we first shuffle the training data to prevent overfitting, and then we fit the model.

Chapter 2

Appendices

Here is the code that one can check that for the same input data but stored in different data format ¹. We can see from the message in the terminal, the converted data format would be preserved.

Here is the code:

```
1 import numpy as np
2
3 # Create random data
4 data_shape = (3, 2, 1) # Assuming 3 channels, 4 rows, and
5   5 columns
6 data_channel_first = np.random.rand(*data_shape)
7 data_channel_last = np.moveaxis(data_channel_first, 0, -1)
8   # Convert to channel_last
9
10 # Assume 'data_format' is set based on how the data is
11   stored
12 data_format_channel_first = 'channel_first'
13 data_format_channel_last = 'channel_last'
14
15 class DataProcessor:
16     def __init__(self, data_format):
17         self.stack_axis = 1 if data_format ==
18             'channel_first' else -1
```

¹That is, the code first assume the random data is stored in **channel.first** and then convert the data format into **channel.last**.

```

16     def process_data(self, data):
17         # Some processing code here
18         processed_data = np.stack([data, data, data],
19                                   axis=self.stack_axis)
19         return processed_data
20
21 # Create instances of DataProcessor for both formats
22 processor_channel_first =
23     DataProcessor(data_format_channel_first)
24 processor_channel_last =
25     DataProcessor(data_format_channel_last)
26
27 # Process data using both processors
28 result_channel_first =
29     processor_channel_first.process_data(data_channel_first)
30 result_channel_last =
31     processor_channel_last.process_data(data_channel_last)
32
33 # Print results
34 print("Check that they are really in channel first and
35       channel last:\n")
36 print(data_channel_first.shape)
37 # print("\n")
38 print(data_channel_last.shape)
39 print("Initial data (channel_first):")
40 print(data_channel_first)
41 print("\nResult after processing (channel_first):")
42 print(result_channel_first)
43
44 print("\nInitial data (channel_last):")
45 print(data_channel_last)
46 print("\nResult after processing (channel_last):")
47 print(result_channel_last)
48
49 print("\nThe difference can be seen by checking their
50       shape:")
51 print(result_channel_first.shape)
52 print("\n")
53 print(result_channel_last.shape)

```
