Foundations for Machine Learning

L. Y. Stefanus
TU Dresden, June-July 2019

Slide 03p

Tensors in PyTorch (cont.)

Reference

- Eli Stevens and Luca Antiga. Deep Learning with PyTorch. Manning Publications, 2019/2020.
- Ian Goodfellow and Yoshua Bengio and Aaron Courville. Deep Learning. MIT Press, 2016.

Numeric Types in Tensors

- What kind of numeric types we can store in a tensor? The dtype argument to tensor constructors (like tensor, zeros, ones) specifies the numerical "data type" that will be contained in the tensor.
- The data type specifies the possible values the tensor can hold and the number of bytes per value.
- The possible values for the dtype argument:

- torch.float32 or torch.float: 32-bit floating point
- torch.float64 or torch.double: 64-bit, double precision fp
- torch.float16 or torch.half: 16-bit, half precision floating point
- torch.int8: signed 8-bit integers
- torch.uint8: unsigned 8-bit integers
- torch.int16 or torch.short: signed 16-bit integers
- torch.int32 or torch.int: signed 32-bit integers
- torch.int64 or torch.long: signed 64-bit integers

- Each of torch.float, torch.double, etc. have a corresponding concrete class of torch.FloatTensor, torch.DoubleTensor, etc.
- The class for torch.int8 is torch.CharTensor and for torch.uint8 is torch.ByteTensor.
- torch.Tensor is an alias for torch.FloatTensor
- 32-bit floating point is the default data type

Numeric Types in Tensors

- In order to allocate a tensor of the right numeric type, we can specify the proper dtype as an argument to the constructor, for examples:

```
# In[47]:
double_points = torch.ones(10, 2, dtype=torch.double)
short_points = torch.tensor([[1, 2], [3, 4]], dtype=torch.short)
```

- We can find out about the dtype for a tensor by accessing the corresponding attribute:

```
# In[48]:
short_points.dtype

# Out[48]:
torch.int16
```

Numeric Types in Tensors

- We can also cast the output of a tensor creation function to the right type using the to method:

```
# In[50]:
double_points = torch.zeros(10, 2).to(torch.double)
short_points = torch.ones(10, 2).to(dtype=torch.short)
```

- We can always cast a tensor of one type into a tensor of another type using the type method:

```
# In[51]:
points = torch.randn(10, 2)
short_points = points.type(torch.short)
```

The function randn initializes the tensor elements to random numbers from the standard normal distribution.

Indexing Tensors

- How can we obtain a 2D tensor containing all points but the first? That's easy using the range indexing or slicing notation, the same that applies to standard Python lists:

```
# In[53]:
some_list = list(range(6))
some_list[:]        ❶
some_list[1:4]      ❷
some_list[1:]       ❸
some_list[:4]       ❹
some_list[:-1]      ❺
some_list[1:4:2]    ❻
```

1. all elements in the list
2. from element 1 to element 3
3. from element 1 to the end of the list
4. from the start of the list to element 3
5. from the start of the list to one before the last element
6. from element 1 to element 3 in steps of 2

Indexing Tensors

- We can use the same notation for PyTorch tensors, with the added benefit that we can use slicing for each of the dimensions of the tensor:

```
# In[54]:
points[1:]          ❶
points[1:, :]       ❷
points[1:, 0]       ❸
```

1. All rows after first, implicitly all columns
2. All rows after first, all columns
3. All rows after first, first column only

Serializing Tensors

- Creating a tensor in RAM is all well and fine, but if the data inside the tensor is of any value to us, we will want to save it to a file and load it back at some point.
- After all, we don't want to have to retrain a model from scratch every time we start running our program!
- PyTorch uses pickle under the hood to serialize the tensor object, plus dedicated serialization code for the storage. Here's how we can save our points tensor to a ourpoints.dt file:

```
with open('c:/kuliah/machineLearning2019/data/ourpoints.dt','wb') as f:
    torch.save(points, f)
```

Serializing Tensors

- Loading our points back:

```
with open('c:/kuliah/machineLearning2019/data/ourpoints.dt','rb') as f:
    points = torch.load(f)
```

Caveat

- While this is a way we can quickly save tensors in case we only want to load them with PyTorch, the file format itself is not interoperable. We can't read the tensor with software other than PyTorch.
- Depending on the use case, this may or may not be a limitation, but we should learn how to save tensors interoperably with other softwares.

Serializing Tensors

- To achieve interoperability, we can usethe HDF5 format and library.
- HDF5 is a portable and widely supported format for representing serialized multidimensional arrays, organized in a nested key-value dictionary.
- Python supports HDF5 through the h5py library, which accepts and returns data under the form of NumPy arrays.
- We can install h5py using anaconda:

  $ conda install h5py

Serializing Tensors

- Now we can save our points tensor by converting it to a NumPy array (at no cost) and passing it to the create_dataset function:

```
# In[61]:
import h5py
f = h5py.File('c:/kuliah/machineLearning2019/data/ourpoints.hdf5', 'w')
dset = f.create_dataset('coords', data=points.numpy())
f.close()
```

- Here 'coords' is a key into the HDF5 file. One of the interesting things in HDF5 is that we can index the dataset while on disk and only access the elements we're interested in.

Serializing Tensors

- Let's suppose we want to load just the last two points in our dataset:

```
# In[62]:
f = h5py.File('c:/kuliah/machineLearning2019/data/ourpoints.hdf5', 'r')
dset = f['coords']
last_points = dset[-2:]
```

- What happened here is that data has not been loaded when the file was opened. Rather, data stayed on disk until we requested the last two rows in the dataset.
- At that point, h5py has accessed those two rows and returned a NumPy array.

Serializing Tensors

- Owing to this fact, we can pass the returned object to the torch.from_numpy function to obtain a tensor directly. Note that in this case the data is copied over to the tensor's storage.
- Once we're finished loading data, we close the file.

```
# In[63]:
last_points_t = torch.from_numpy(dset[-2:])
f.close()
```

Moving Tensors to the GPU

- Every PyTorch tensor can be transferred to the GPU(s) in order to perform massively parallel, fast computations. All operations performed on the tensor will be carried out using GPU-specific routines that come with PyTorch.
- In addition to dtype, a PyTorch Tensor also has a notion of device, which is where on the computer the tensor data is being placed. Here is how we can create a tensor on the GPU (if it exists) by specifying the corresponding argument to the constructor:

```
# In[64]:
points_gpu = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]], device='cuda')
```

Moving Tensors to the GPU

- We could instead copy a tensor created on the CPU onto the GPU using the to method:

```
# In[65]:
points_gpu = points.to(device='cuda')
```

- Doing so returns a new tensor that has the same numerical data, but stored in the RAM of the GPU, rather than in regular system RAM.
- Now that the data is stored locally on the GPU, we'll see speedups when performing mathematical operations on the tensor.
- The class of this new GPU-backed tensor is also changed to be torch.cuda.FloatTensor (given our starting type of torch.FloatTensor).

The Tensor API

- It is worth taking a look at all the tensor operations that PyTorch offers. Check out the online documentation at pytorch.org/docs.
- The vast majority of operations on and between tensors are available under the torch module and can also be called as methods of a tensor object.
- For instance, the transpose function we've encountered earlier can be used from the torch module:

```
# In[71]:
a = torch.ones(3, 2)
a_t = torch.transpose(a, 0, 1)
```

or as a method of the a tensor

```
# In[72]:
a = torch.ones(3, 2)
a_t = a.transpose(0, 1)
```

The Tensor API

- There is no difference between the two forms; they can be used interchangeably.
- There's a caveat: a small number of operations only exist as methods of the tensor object. They are recognizable from a trailing underscore in their name, like zero_, which indicates that the method operates *in-place*, by modifying the input instead of creating a new output tensor and returning it. For instance, the zero_ method zeroes out all the elements of the input.
- Any method *without* the trailing underscore leaves the source tensor unchanged, and instead returns a new tensor.

The Tensor API

```
# In[73]:
a = torch.ones(3, 2)

# In[74]:
a.zero_()
a

# Out[74]:
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
```