Geospatial Deep Learning

Tensors

Image from NASA: https://commons.wikimedia.org/wiki/File:Earth_Western_Hemisphere_transparent_background.png#filelinks

In this module we will discuss tensors since this is the primary data model used to perform deep learning (DL).

A decent understanding of tensors and tensor operations in Python and PyTorch can go a long way when learning to use DL techniques.

Module Topics

1. Tensor data model
2. The computational graph
3. Data types
4. Size, dimension, and shape
5. Indexing
6. Slicing
7. Reshaping
8. Reduction
9. Representing real data as tensors
10. PyTorch DataSets
11. PyTorch DataLoaders
12. Mini-batches

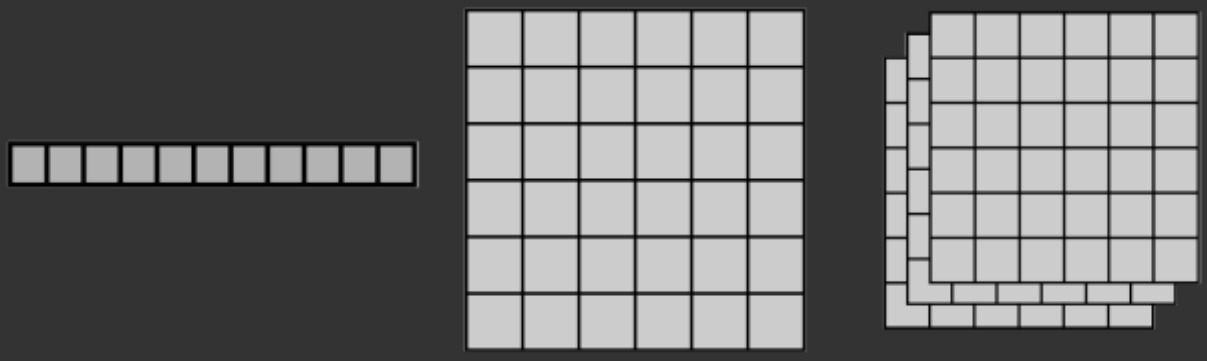
2

These are the topics covered in this module.

Intro to Tensors

What is a Tensor?

- Multidimensional array of data
- General data model that allows for a wide variety of data inputs



4

In mathematics, the term tensor has a much broader and complex meaning then how the term is used in data science and deep learning.

In deep learning, a tensor is a set of values stored in an array with a defined shape.

This model is used in deep learning as it is very flexible and can be used to consistently store and represent a wide variety of data inputs.

Tensors are essentially multidimensional arrays.

NumPy

- Python library
- Allow for creating and working with multi-dimensional arrays in Python
- Very fast and memory efficient
- Stored in memory buffers allocated by C
- Can perform linear algebra/matrix computations

https://numpy.org/

5

In the Python environment, multidimensional arrays are commonly stored and manipulated using NumPy.

In Python, a list of values can be converted to a 1-dimensional (1D) NumPy array. A 1D NumPy array is called a vector. 2-dimensional (2D) arrays are called matrices. A single value, or 0-dimensional (0D) array, is called a scalar.

Similar to Python, in R a 0D array is called a scalar, a 1D array of values is also called a vector, a 2D array is called a matrix, and a 3D array is called an array.

NumPy has proven to be very valuable as it allows for fast and memory efficient handling and manipulation of arrays that are stored in memory buffers allocated by C.

NumPy vs. PyTorch

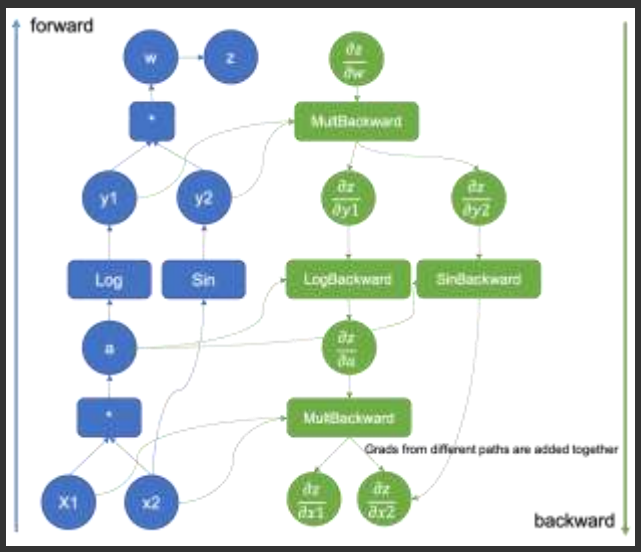
- PyTorch:
  - Supports GPU-based computation
  - Computational Graph
  - *requires_grad*
  - *device_type*
  - Code is very similar to NumPy

https://pytorch.org/

6

PyTorch expands upon NumPy arrays with the tensor class. In contrast to NumPy arrays, PyTorch tensors can be stored and operated on by the CPU or GPU. Also, derivatives can be calculated on these arrays, which allow for performing backpropagation and weight/parameter updates.

Computational Graph



- Keep track of mathematical operations
- Allow for calculation of gradient with respect to parameters
- Necessary for updating parameters

https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/

7

PyTorch can keep track of the series of mathematical operations that are applied to data to obtain a tensor. This record of the mathematical operations applied is called a computational graph. This computational graph allows for the calculations of gradients, which is necessary for optimization algorithms, such as mini-batch gradient descent, to be applied to perform weight/parameter updates.

With the computational graph stored, PyTorch is able to use backpropagation to determine the gradient of the loss with respect to a trainable parameter. The optimization algorithm can then use this information to update the parameters.

In summary, PyTorch provides functionality that is not provided by NumPy and is required to implement deep learning. It allows for operations to be performed on either a CPU or GPU, data to be moved between devices, and backpropagation and optimization to be performed by maintaining the computational graph.

Data Types

```
t13 = torch.ones(size=(2,4,4),
dtype=torch.float64)
print(t13.dtype)

t14 = torch.ones(size=(2,4,4),
dtype=torch.float16)
print(t14.dtype)

t15 = torch.ones(size=(2,4,4),
dtype=torch.uint8)
print(t15.dtype)

t16 = torch.ones(size=(2,4,4),
dtype=torch.int8)
print(t16.dtype)
```

- Integer (8- , 16- (short), 32-, 64-bit (long))
- Unsigned Integer (8- , 16- (short), 32-, 64-bit (long))
- Float (16-, (half-precision), 32- (float), 64-bit (double)
- Boolean

https://pytorch.org/docs/stable/tensors.html

8

PyTorch tensors can store values with different data types. All values in the same tensor will have the same data type.

Signed or unsigned integer values can be stored as 8-, 16-, 32-, or 64-bit. Signed data can differentiate positive and negative values while unsigned cannot. A higher bit-depth will take up more space, so it is generally best to use the lowest depth possible. 16-bit is called short while 64-bit is called long. A higher bit depth means that more unique values can be differentiated.

Float numbers, which can include decimal values, are stored as 16-, 32-, or 64-bit. 32-bit is termed float, 64-bit is termed double, and 16-bit is termed half-precision.

It is also possible to generate and manipulate tensors of Boolean values: True and False.

Tensor Characteristics

```
print(type(t1))
print(t1.shape)
print(t1.dtype)
print(len(t1))
print(t1.ndim)
print(torch.numel(t1))
```

- Size = number of data points in the array
- Dimensions = number of dimensions or axes
- Shape = length of each dimension

```
<class 'torch.Tensor'>
torch.Size([3, 3])
torch.int64
3
2
9
```

9

A tensor can be described based on its size, dimensions, shape, and the data type of the values stored.

The size of an array is the number of values stored across all dimensions. Dimension refers to the number of unique dimensions (0D, 2D, 3D, 4D, 5D, etc.). It is uncommon to use tensors larger than 5D in deep learning. The shape relates to the length of each dimension.

The example torch tensor has a data type of 64-bit signed integer (long), two dimensions, three positions within each dimension, and a total of 9 values.

Tensor Dimensions

- 0D = single value or data point (scalar)
- 1D = set of values or data points (vector)
- 2D = data points spread across two dimensions (matrix)
- 3D = data points spread across three dimensions
- 4D = data points spread across four dimensions
- 5D = ...
- 6D = ...

```
t5 = torch.ones(size=(2,4,4))
print(t5)
```

```
tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
```

10

This slide provides some examples of tensor dimensions.

A 0D tensor is called a scalar and consists of a single value or data point. A 1D tensor is a list of values and is known as a vector. A 2D tensor is a matrix, where the data values are spread across 2 dimensions (or, rows and columns).

Tensors with 3 or more dimensions do not have special names.

The example on the slide is a 3D tensor.

Working with Tensors

Tensor Indexing

- Use bracket notation
- [First Dimension, Second Dimension, Third Dimension]
- [:,3:, 2:4]
- Indexing starts at 0
- First index included
- Last index not included
- : = select all values in index
- : Value = select all values before index
- Value : = select value and all values following it

```
tOut = tIn[0:4,0:4]
```

12

Subsets of values from an array or tensor can be extracted using bracket notation. Please read through the indexing and subsetting rules on this slide. Remember that Python and PyTorch indexing starts at 0 as opposed to 1.

Tensor Slicing

- Extract values based on indices

```
tOut = tIn[0:4,0:4]
```

13

Tensor slicing is the process of extracting values based on indices.

In the provided example, values at index 0 through 3 in both the 1st and 2nd dimensions will be extracted.

Array Reshaping

- Change number of dimensions and/or length of each dimension
- Values must completely fill the new shape

```
t20 = torch.reshape(t17, (2, 4, -1))
print(t20)

t22 = t17.view((2,16))
print(t22)

t34 = torch.permute(t33, (1,2,0))
t34
```

```
t21 = torch.flatten(t17)
print(t21)

t30 = t29.squeeze()
print(t30.shape)

t31 = t30.unsqueeze(dim=2)
print(t31.shape)
```

14

Tensors can be reshaped using the reshape() function. The number of values in the original array must completely fill the new shape.

The view() method allow for the original values in a tensor to be represented using a new shape without copying the values in memory.

The permute() function allows for re-ordering the bands or dimensions of the tensor.

The flatten() function will convert any tensor shape to a 1D tensor.

Lastly, squeeze() is used to remove dimensions with a length of 1 while unsqueeze() is used to add a dimension with a length of 1 at a defined position.

Tensor Operations

- Operations on a single tensor
- Operations between tensors
- Operations between tensors and constants
- Operations that yield Boolean result

```
t43 = t35 > 180
```

```
tensor([[[False, False, False, False, False, False],
         [False,  True, False, False, False, False],
         [False, False,  True, False, False, False],
         [False, False, False, False, False,  True],
         [False,  True,  True, False,  True, False],
         [False, False, False,  True,  True, False]],
```

15

PyTorch supports a variety of mathematical operations that can be performed on a single tensor, between multiple tensors, or between a single tensor and a constant.

Logical operations will return Boolean tensors where True indicates that the condition was met at that position and False means that it was not.

Tensor Reduction

- Aggregate to single statistics
- Aggregate across dimensions

```
t47 = torch.argmax(t45, dim=1)

t48 = torch.amax(t45, dim=0)

t50 = torch.mean(torch.tensor(t45,
dtype=torch.float32))
```

16

There are also a variety of functions for reducing a tensor. For example, the argmax() function returns the index that contains the largest value relative to a specified dimension while amax() returns the largest value. You will see argmax() applied in this class to obtain a hard classification from a set of class probabilities.

Statistical measures, such as the mean, median, or standard deviation, can be used to reduce the tensor to a single value.

Tensor Examples

Batch Dimension

- First dimension is often the sample or mini-batch dimension
- With this dimension included, images are 4D tensors
- Weight/parameter updates by mini-batch

(Samples, Color Depth, Height, Width)

Example

(32, 3, 512, 512)

- 32 images in mini-batch
- 3 channels per image
- 512 pixels tall per image
- 512 pixels wide per image

18

When using tensors to feed training data into a deep learning model, it is common to submit a subset of the data as opposed to a single sample or the entire dataset. These are known as batches or mini-batches. Weight/parameter updates will be implemented after each mini-batch when using mini-batch gradient descent or its derivatives (e.g., Adam) as opposed to after each full training epoch.

To support this, the first dimension of tensors is commonly reserved for the number of samples. This is known as the batch or mini-batch dimension.

The provided example represents images as input to a convolutional neural network. The first dimension is the mini-batch dimension (32 images are submitted). The second dimension represents the channels dimension. So, each image has three channels: red, green, and blue. The last two dimensions represent the rows and columns of pixels (each image has 512 rows and 512 columns of pixels).

Spreadsheet or Data Table

- 2D Tensor or matrix

(Sample, Variable)

- Excel Spreadsheet
- R or Pandas data frame

19

We will now relate a variety of common data types to their representation as a tensor.

A spreadsheet or data table would be a 2D tensor or matrix where each row represents a sample, and each column represents a piece of data, measurement, or variable for each sample. For example, each row could be a student while each column represents data for each student.

This is similar to an Excel Spreadsheet or R or Pandas data frame.

Dimension Order

- Tensorflow uses channels-last convention

  (Samples, Height, Width, Channels)

  (32, 512, 512, 3)

- PyTorch uses channels-first convention

  (Samples, Channels, Height, Width)

  (32, 3, 512, 512)

20

A batch of images is a 4D tensor of samples, channels, height, and width dimensions.

There are two different conventions used to represent an image as a tensor: channels-last and channels-first. We will use the channels-first convention, which is expected when using PyTorch.

RGB

- Data stored in three channels (RGB)
- Images are commonly 8-bit (0 – 255 gray levels)
- Will need to be converted to float (0 – 1) for deep learning

21

RGB (Red, Green, Blue) images contain three image channels represented as a tensor dimension.

It is common to represent brightness values in the 8-bit space (0-255). However, deep learning operations in PyTorch generally expect the data to be float, as opposed to 8-bit integer.

Such data conversion is a common pre-processing step.

Grayscale

- Can be represented as 3D tensor

  (Samples, Height, Width)

  (32, 512, 512)

- Can be represented as 4D tensor

  (Samples, Channels, Height, Width)

  (32, 1, 512, 512)

22

Grayscale image batches can be represented as a 3D tensor or a 4D tensor. If a 4D representation is used, the channel dimension will have a length of 1, which is equivalent to a 3D tensor.

This can be tricky. If an operation requires a 4D input, you will need to reshape the 3D grayscale data to 4D. This can be accomplished using unsqueeze().

Time Series Data

- Time series could be 3D tensor

  (Samples, Time, Variable)

- Video or time-lapse images could be 5D tensor

  (Samples, Frame/Time, Channel, Height, Width)

23

A time series is commonly modelled as a 3D tensor, which includes a mini-batch dimension, time dimension, and variable dimension.

Video or time-lapse photography data are commonly 5D: mini-batch, frame number or time, channels, height, and width.

Data Pipelines

DataSets

- PyTorch expects tensors of defined shape and data type
- Predictor variables + Labels
- DataSet class defines pipeline to convert a raw data sample to a tensor

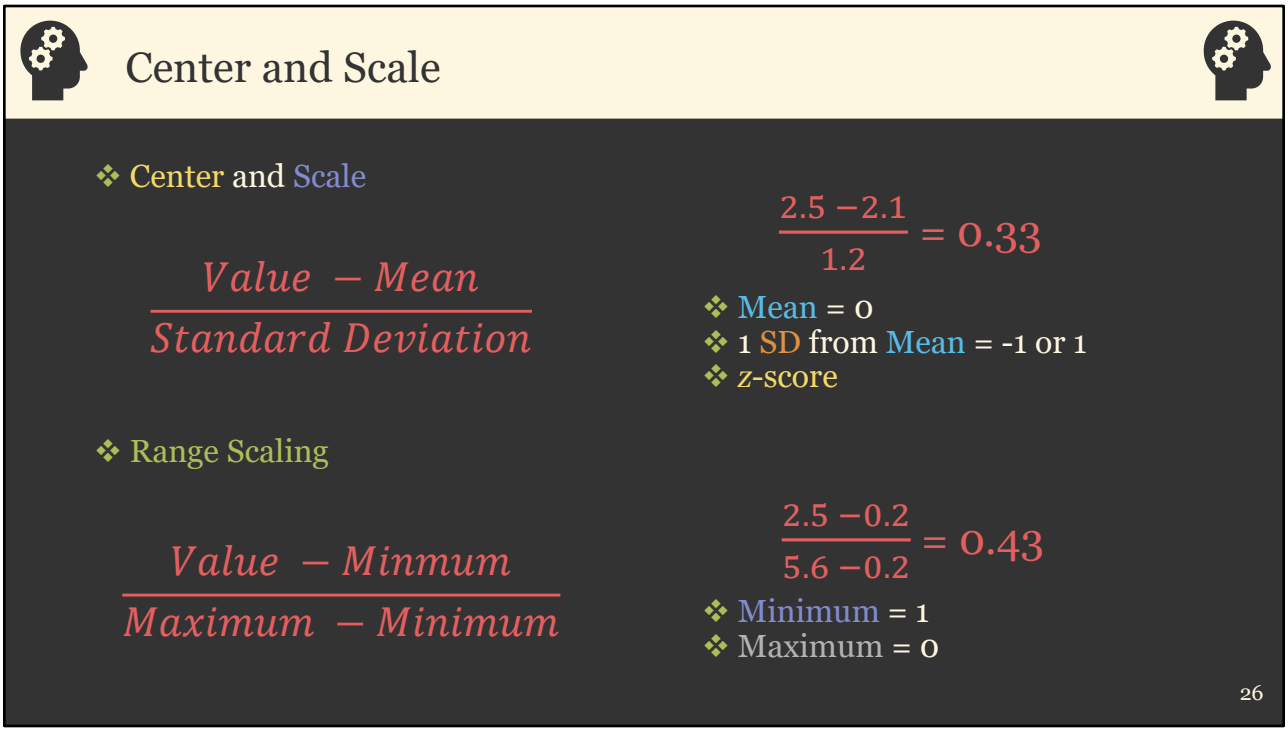
```
class EuroSat(Dataset):

    def __init__(self, df, mnImg, sdImg):
        self.df = df
        self.mnImg = mnImg
        self.sdImg = sdImg

    def __getitem__(self, idx):
        image_name = self.df.iloc[idx, 1]
        label = self.df.iloc[idx, 3]
        label = np.array(label)
        source = rio.open(image_name)
        image = source.read()
        source.close()
        image = image[[1,2,3,4,5,6,7,8,11,12], :, :]
        image = np.subtract(image, self.mnImg)
        image = np.divide(image, self.sdImg)
        image = image.astype('float32')
        image = torch.from_numpy(image)
        label = torch.from_numpy(label)
        label = label.long()
        return image, label

    def __len__(self):
        return len(self.df)
```

25

In order to perform deep learning on a wide variety of data formats, you will need to be able to convert these data to tensors with the correct shapes and data types. Fortunately, PyTorch provides utilities for converting data to tensors. Additional functionality is provided by TorchVision and TorchAudio.

One key component of this implementation is the DataSet class. You can create a custom DataSet by subclassing the DataSet class. This subclass must be able to deliver a single sample as a tensor with the correct shape and data types. To train the model, each sample will need to include the predictor variable(s) and the dependent variable. For example, the predictor variables could be the image bands while the labels could be numeric codes associated with each class.

In the PyTorch modules, you will learn to build and use custom DataSets.



It is common to normalize data values in some way prior to feeding the data through the network.

There are multiple ways to rescale data. The center and scale method consists of subtracting the mean from each data point (centering) then dividing by the standard deviation (scaling). Once this process is undertaken, the data will be centered at or have a mean of zero, and data points that are 1 standard deviation from the mean will have a value of -1 or 1. This scaling is commonly referred to as a z-score.

Another option is range scaling in which the minimum is subtracted from each value and the result is divided by the range (maximum minus minimum). Using this method, the maximum value will scale to 1 and the lowest value will scale to 0.

You may rescale the data using statistics derived from the training samples. When applying transfer learning, you may use statistics derived from the dataset that the original weights/parameters were learned from, such as ImageNet.

Transforms and Augmentations

- Convert to tensor
- Normalize
- Convert data types
- Reshape
- Apply augmentations to minimize overfitting

27

There are several common operations that are applied to prepare data. These include converting the raw data to PyTorch tensors, performing normalization, converting between data types (e.g., 32-bit float to 64-bit double), reshaping such that the number and order of the dimensions are correct, and/or applying data augmentations. Applying data augmentations or transformations is one means to potentially combat overfitting, especially when the training dataset is not large.

You will see examples of data transformations and augmentations throughout the PyTorch examples.

DataLoaders

- PyTorch class
- Deliver mini-batch of data to the learning or inference workflow
- Weight/parameter updates generally happen after each mini-batch is processed

```
trainDL = torch.utils.data.DataLoader(trainDS, batch_size=32,
shuffle=True, num_workers=0, pin_memory=False, drop_last=True)
```

28

The DataLoader class is used to provide the samples to the algorithm as batches or mini-batches. Thus, this class is key to training models using mini-batch gradient descent or one of its derivatives. Given an instance of a DataSet subclass, the DataLoader must be able to deliver mini-batches of samples with the correct shapes and data types. You can also apply shuffling for enhanced randomization or define a sampler to subsample samples from the larger dataset.

This is the end of this lecture module.

Please return to the West Virginia View Webpage for additional content.

Thanks! Hope you found this useful.