Numpy + TensorFlow Review

BIODS 220: Artificial Intelligence in Healthcare

Numpy Review

What is Numpy?

A library that supports large, multi-dimensional arrays and matrices and has a large collection of high-level mathematical functions to operate on these arrays

Outline

- Basics
  - Properties
  - Creating arrays and basic operations
  - Universal math functions
  - Saving and loading images
- Advanced
  - Mathematical operators
  - Indexing, slicing
  - Broadcasting

Basics

```
import numpy as np

a = np.array([[1,2,3],[4,5,6]],dtype=np.float32)

print(a.ndim, a.shape, a.dtype)
```

1. Arrays can have any number of dimensions, including zero (a scalar)
2. Arrays are typed (np.uint8, np.int64, np.float32, np.float64)
3. Arrays are dense (each element of the array exists and has the same type)

Basics

Creating arrays:

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

Basics

Creating arrays:

- **np.ones, np.zeros**
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> np.ones((3,5),dtype=np.float32)
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]], dtype=float32)
```

```
>>> np.zeros((6,2),dtype=np.int8)
array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]], dtype=int8)
```

Basics

Creating arrays:

- np.ones, np.zeros
- **np.arange**
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> np.arange(1334,1338)
array([1334, 1335, 1336, 1337])
```

Basics

Creating arrays:

- np.ones, np.zeros
- np.arange
- **np.concatenate**
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> A = np.ones((2,3))
>>> B = np.zeros((4,3))
>>> np.concatenate([A,B])
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>>
```

Basics

Creating arrays:

- np.ones, np.zeros
- np.arange
- np.concatenate
- **np.astype**
- np.zeros_like, np.ones_like
- np.random.random

```
>>> A
array([[ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5]], dtype=float32)
>>> print(A.astype(np.uint16))
[[4670 4670 4670]
 [4670 4670 4670]
 [4670 4670 4670]
 [4670 4670 4670]
 [4670 4670 4670]]
```

Basics

Creating arrays:

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- **np.zeros_like, np.ones_like**
- np.random.random

```
>>> a = np.ones((2,2,3))
>>> b = np.zeros_like(a)
>>> print(b.shape)
```

Basics

Creating arrays:

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- **np.random.random**

```
>>> np.random.random((10,3))
array([[ 0.61481644,  0.55453657,  0.04320502],
       [ 0.08973085,  0.25959573,  0.27566721],
       [ 0.84375899,  0.2949532 ,  0.29712833],
       [ 0.44564992,  0.37728361,  0.29471536],
       [ 0.71256698,  0.53193976,  0.63061914],
       [ 0.03738061,  0.96497761,  0.01481647],
       [ 0.09924332,  0.73128868,  0.22521644],
       [ 0.94249399,  0.72355378,  0.94034095],
       [ 0.35742243,  0.91085299,  0.15669063],
       [ 0.54259617,  0.85891392,  0.77224443]])
```

Basics

```
a = np.array([1,2,3,4,5,6])

a = a.reshape(3,2)

a = a.reshape(2,-1)

a = a.ravel() # returns a contiguous flattened array
```

1. Total number of elements cannot change.
2. Use -1 to infer axis shape
3. Row-major by default (MATLAB is column-major)

Basics

```
a = np.arange(10).reshape(5,2)

a = a.T

a = a.transpose((1,0))
```

1. a.T transposes the first two axes.
2. np.transpose permutes axes.

Basics

Saving and loading images:

- Using PIL/Pillow (width x height x RGB):

```
from PIL import Image

im = Image.open(*file path*) # opens image

im.save(*file path*) # saves image
```

Basics

Saving and loading images:

- Using OpenCV (height x width x BGR):

```
import cv2

im = cv2.imread(*file path*) # reads in image

cv2.imwrite(*file path*, im) # writes out image
```

Advanced

Mathematical operators

- **Arithmetic operations are element-wise**
- Logical operators return a boolean array
- In place operations modify the array

```
>>> a
array([1, 2, 3])
>>> b
array([ 4,  4, 10])
>>> a * b
array([ 4,  8, 30])
```

Advanced

Mathematical operators

- Arithmetic operations are element-wise
- **Logical operators return a boolean array**
- In place operations modify the array

```
>>> a
array([[ 0.93445601,  0.42984044,  0.12228461],
       [ 0.06239738,  0.76019703,  0.11123116],
       [ 0.14617578,  0.90159137,  0.89746818]])
>>> a > 0.5
array([[ True, False, False],
       [False,  True, False],
       [False,  True,  True]], dtype=bool)
```

Advanced

Mathematical operators

- Arithmetic operations are element-wise
- Logical operators return a boolean array
- **In place operations modify the array**

```
>>> a
array([[ 4, 15],
       [20, 75]])
>>> b
array([[ 2,  5],
       [ 5, 15]])
>>> a /= b
>>> a
array([[2, 3],
       [4, 5]])
```

Advanced

Indexing

```
x[0,0]     # top-left element

x[0,-1]    # first row, last column

x[0,:]     # first row (many entries)

x[:,0]     # first column (many entries)
```

- Zero-indexing
- Multi-dimensional indices are comma-separated

Advanced

```
I[1:-1,1:-1]      # select all but one-pixel border

I = I[:,:,::-1]   # swap channel order

I[I<10] = 0       # set dark pixels to black

I[[1,3], :]       # select 2nd and 4th row
```

Advanced

```
a.sum()                          # sum all entries
a.sum(axis=0)                    # sum over rows
a.sum(axis=1)                    # sum over columns
a.sum(axis=1, keepdims=True)     # sum over columns + keep dims
```

- Use the axis parameter to control which axis Numpy operates on
- Typically, the axis specified will disappear, keepdims keeps all dimensions
  - E.g. instead of resulting in shape (3,) –> result in shape (3,1)

TensorFlow Review

What is TensorFlow?

- An open-source library for dataflow and differentiable programming across a range of tasks
- Used for machine learning applications such as neural networks

Why TensorFlow?

- Makes it easy to prototype and build machine learning models by providing multiple levels of abstraction
- Handles distributed training for high compute ML training tasks
- Provides a direct path to production to deploy machine learning models for your applications

Installation

- TensorFlow 2.X comes pre-installed on your Deep Learning VMs!

Steps for training models

1. Preprocessing dataset
2. Defining a model architecture
3. Using an optimizer to minimize a loss function w.r.t. model parameters

Outline

- TensorFlow common operations
- TensorFlow Datasets
- TensorFlow Keras API and linear algebra operations for defining models
- Framework for training models

What is a tensor?

- A **tensor** is a generalization of vectors and matrices to potentially higher dimensions
- TensorFlow represents tensors as n-dimensional arrays of base data types
- When writing TensorFlow programs, the main object you manipulate and pass around is a tf.Tensor object
  - A tf.Tensor object consists of:
    - data type (float32, int32, string, etc.)
    - shape (e.g. 3 x 1 vector has shape (3, 1))

Common use operations

- Making tensors (constants and variables) and casting tensors
  - Constants are fixed
    - `const = tf.constant([[3, 2],[5, 2]])`
  - Variables can be assigned to any value and can be optimized (are trainable)
    - `a = tf.Variable([[3, 2],[5, 2]])`
  - Tensors of all zeros or all ones
    - `b = tf.zeros(shape=[5, 4], dtype=tf.int32)`
    - `c = tf.ones(shape=[5, 4], dtype-tf.int32)`
  - Casting tensors to specific data types
    - `c = tf.cast(c, tf.float32)`

Common use operations

- Concatenate two tensors
  - ```
    a = tf.constant([[4, 6],[5, 3]])
    b = tf.constant([[7, 3],[1, 1]])
    c1 = tf.concat(values=[a, b], axis=1) # [[4 6 7 3], [5 3 1 1]]
    c2 = tf.concat(values=[a, b], axis=0) # [[4 6], [5 3], [7 3], [1 1]]
    ```
- Reshape tensor
  - ```
    tf.reshape(tensor = c2, shape=[1, 8]) # [[4, 6, 5, 3, 7, 3, 1, 1]]
    ```
- Can convert tensor to numpy
  - ```
    c_np = c1.numpy()
    ```
- Can convert numpy to tensor
  - ```
    c_tensor = tf.convert_to_tensor(c_np)
    ```

TensorFlow Datasets

- Handles batching and shuffling of data for training in a simple framework
- TensorFlow provides a nice API for loading datasets
  - tf.data.Datasets class for loading datasets consisting of input tensors and label tensors
- Keras built-in datasets
  - Regression and classification datasets built into Keras can be accessed directly using tf.keras.datasets
  - Example (MNIST dataset for handwritten digit classification):

```
mnist = tf.keras.datasets.mnist
```

Returns tuple of numpy arrays `(x_train, y_train), (x_test, y_test)`

TensorFlow Datasets

- Generate batches of tensor image data with real-time data augmentation
    - `tf.keras.preprocessing.image.ImageDataGenerator`

Linear algebra operations

- Transpose tensor
  - `a = tf.constant([[4, 6],[5, 3]])`
  - `a = tf.transpose(a)                    # [[4, 5],[6, 3]]`
- Matrix multiplication
  - `a = tf.constant([[4, 6],[5, 3]])`
  - `b = tf.constant([[5], [2]])`
  - `ab = tf.matmul(a, b)                   # [[32],[31]]`
- Identity matrix
  - `tf.eye(num_rows=a.shape[0], num_columns=a.shape[1], dtype=tf.int32)`

Linear algebra operations

- Dot product
  - `a = tf.constant([[2, 1]])`
  - `b = tf.constant([[3], [5]])`
  - `ab = tf.tensordot(a=a, b=b, axes=1)    # [11]`

Model definition

TensorFlow Keras layers (for defining model components easily)

- Fully connected/Dense/Linear layers
  - `tf.keras.layers.Dense(units=32, activation='softmax')`
- Flatten layers
  - `tf.keras.layers.Flatten()`
- 2d convolutional layers
  - `tf.keras.layers.Conv2D(filters=32, kernel_size=(3,3), padding='same', activation='relu')`
- Batch Normalization layers
  - `tf.keras.layers.BatchNormalization()`
- And many more! (https://www.tensorflow.org/api_docs/python/tf/keras/layers)

Examples of model definitions

- A Keras sequential model makes things very simple! (training and testing functions are already built in)
- Pass a list of layers as the input to Sequential

Example (model with 2 linear/dense layers):

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(units=16, activation='relu'),
    tf.keras.layers.Dense(units=2, activation='softmax')
])
```

Examples of model definitions

- There are 3 Keras APIs to define a Keras model: Sequential API, Model Subclassing API, and Functional API
- In this class, we will use the Sequential API and and Model Subclassing API

Examples of model definitions

- Keras Model Subclassing API example:

```
class ResNet(tf.keras.Model):

    def __init__(self):
        super(ResNet, self).__init__()
        self.block_1 = ResNetBlock()
        self.block_2 = ResNetBlock()
        self.global_pool = layers.GlobalAveragePooling2D()
        self.classifier = Dense(num_classes)

    def call(self, inputs):
        x = self.block_1(inputs)
        x = self.block_2(x)
        x = self.global_pool(x)
        return self.classifier(x)

model = ResNet()
```

Training models

Keras sequential model makes things very simple!

- `model.fit(...)` takes care of training the whole model end to end
- Example:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(units=16, activation='relu'),
    tf.keras.layers.Dense(units=2, activation='softmax')
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
model.fit(x, y, epochs=10)
```

TensorFlow backpropagation

- Process of optimizing model parameters through gradient updates during training
- Backpropagation is handled implicitly in Tensorflow
- TensorFlow generates a **computation graph** that consists of tensors and the operations between them
- Keras makes this very high-level and hides it under the hood

Training models

- Can use optimizers to minimize a loss function by applying gradients
    - Define optimizer (example uses Adam optimizer but there are other alternatives):
        - `optimizer = tf.keras.optimizers.Adam()`
    - Define loss function (examples uses sparse categorical cross entropy but there are other alternatives)
        - `loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()`

Example framework with model train/test functions

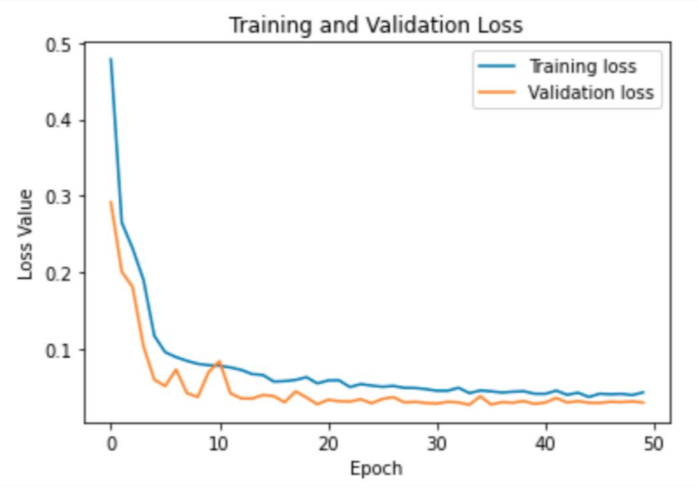
```
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()

optimizer = tf.keras.optimizers.Adam()

train_loss = tf.keras.metrics.Mean(name='train_loss')
train_metric = tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')

test_loss = tf.keras.metrics.Mean(name='test_loss')
test_metric = tf.keras.metrics.SparseCategoricalAccuracy(name='test_accuracy')

model = CustomModel(loss_fn = loss_fn,
                    optimizer = optimizer,
                    train_loss = train_loss,
                    train_metric = train_metric,
                    test_loss = test_loss,
                    test_metric = test_metric)

model.fit(train = train_ds,
          test = test_ds,
          epochs = 5)
```

Visualizing Results

Save the history of the Keras `fit` function:

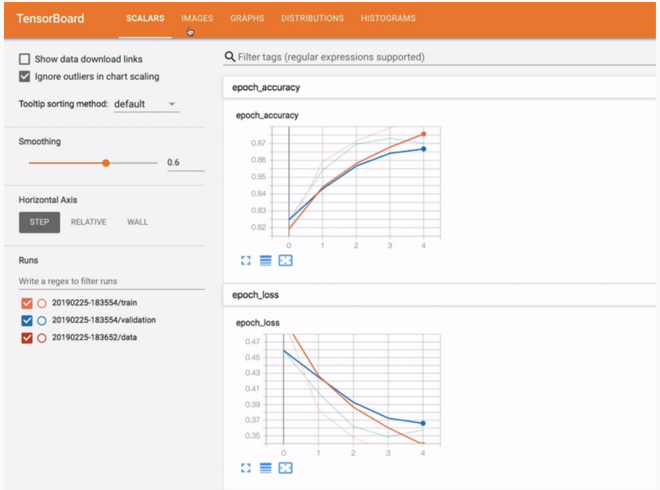
```
history = model.fit(...)
```

Use matplotlib to plot the history



Visualizing Results

- Can also use TensorBoard! (Could be useful for final projects)
    - Real-time tracking of loss curves and train/val metrics over time
- `tf.keras.callbacks.TensorBoard` can be used for logging to TensorBoard
    - e.g. `model.fit(..., callbacks=[tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)]`
    - This logs metrics, loss, etc. to tensorboard
- Other methods
    - Set up summary writers using tf.summary
    - e.g. `tf.summary.scalar('loss', train_loss.result(), step=epoch)`
    - This logs training loss per epoch



Evaluate models

Very simple!

```
model.evaluate(test_dataset)
```

TensorFlow Keras Docs:

https://www.tensorflow.org/api_docs/python/tf/keras