Numpy + Tensorflow Review

BIODS220: 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

● Installation

● Basics
  ○ Properties
  ○ Creating arrays and basic operations
  ○ Universal math functions
  ○ Saving and loading images

● Advanced
  ○ Mathematical operators
  ○ Indexing, slicing
  ○ Broadcasting
Installation

- Python version: >= 3.5
- Numpy version: >= 1.16.4
  - To install: `pip install numpy`
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`
Basics

Creating arrays:

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

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

Creating arrays:

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

```python
>>> 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
Basics

Creating arrays:

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

```python
>>> 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

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

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

```python
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):

```python
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):

  ```python
  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

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

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

```python
>>> 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
Advanced

Universal functions

- Element-wise
- Examples:
  - `np.exp`
  - `np.sqrt`
  - `np.sin`
  - `np.cos`
  - `np.isnan`

```python
>>> 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
```

- Slices are views. Writing to a slice overwrites the original array.
- Can also index by a list or boolean array.
Advanced

```python
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
Advanced

Suppose we want to add a color value to an image

- \(a\) is an RGB image of shape \((100, 200, 3)\)
- \(b\) is an array with the color values we want to add with shape \((3)\)

\(a + b\) will pad \(b\) with two extra dimensions so it has an effective shape of \(1 \times 1 \times 3\), thereby broadcasting over the first and second dimensions.

- Dimensions between two arrays are compatible if they are equal or if one of them is 1
- Broadcasting starts with trailing dimensions and works its way forward
Tensorflow Review
What is Tensorflow?

Tensorflow is an open-source library for dataflow and differentiable programming across a range of tasks.

It is used for machine learning applications such as neural networks.
Why Tensorflow?

Tensorflow makes it easy to prototype and build machine learning models by providing multiple levels of abstraction.

Tensorflow handles distributed training for high compute ML training tasks

Tensorflow provides a direct path to production to deploy machine learning models for your applications
Installation

- Python version: >= 3.5
- Tensorflow version: >= 2.0.0
- To install:
  - **CPU Version**: `pip install tensorflow`
  - **GPU Version**: `pip install tensorflow-gpu`
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
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.Dataset 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):
    ```python
    mnist = tf.keras.datasets.mnist
    Returns tuple of numpy arrays (x_train, y_train), (x_test, y_test)
    ```
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!
Examples of model definitions

Keras sequential model makes things very simple! (training and testing functions are already built in)

Example (model with 2 linear/dense layers):

```python
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

Custom model class using Keras APIs (have to define training and testing functions)

Example (model with 2 linear/dense layers):

class KerasModel(tf.keras.Model):
    def __init__(self):
        self.dense_layer0 = tf.keras.layers.Dense(units=16, activation='relu')
        self.dense_layer1 = tf.keras.layers.Dense(units=2, activation='softmax')

    def __call__(self, x):
        return self.dense_layer1(self.dense_layer0(x))
Training models

Keras sequential model makes things very simple!

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

```python
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
  - GradientTape records all operations executed in its scope
    - Forward pass + loss computation
  - Tape is used to compute gradients through automatic differentiation
  - Optimizer applies gradients to update specified model parameters

Will go into more details on each of these next!
Training models

Calculating gradients

def fn(x):
    return 0.5 * tf.pow(x, 2)

x = tf.Variable(0.5)

with tf.GradientTape() as tape:
    y = fn(x)
    grad = tape.gradient(y, x)  # 0.5 <- result of performing dy/dx
Training models

Given a model, `model`, and a loss function, `loss_fn` (both from `tf.keras`),

```python
with tf.GradientTape() as tape:
    out = model(...)
    loss = loss_fn(...)
    grads = tape.gradient(loss, model.trainable_variables)
...
```
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()`
  - Compute gradients from loss (shown in previous slide)
    - `grads = tape.gradient(loss, model.trainable_variables)`
  - Use optimizer to apply gradients to model variables
    - `optimizer.apply_gradients(zip(grads, model.trainable_variables))`
Training models

- Computing metrics during training
  - Metrics like loss, accuracy, mean squared error, etc. can be computed and tracked using tf.keras.metrics

- Example:

```python
train_loss = tf.keras.metrics.Mean('train_loss', dtype=tf.float32)
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy('train_accuracy')

train_loss(ground_truth, prediction)  # accumulates metric statistics
train_accuracy(ground_truth, prediction)  # accumulates metric statistics

train_loss.result()  # computes and returns the metric value tensor
train_loss.reset_states()  # resets all of the metric state variables
```
Example framework with model train/test functions

```python
class CustomModel(Model):
    def __init__(self, loss_fn, optimizer, train_loss, train_metric, test_loss, test_metric):
        super(CustomModel, self).__init__()
        self.flatten_layer = Flatten(input_shape=(28, 28))
        self.dense_layer = Dense(10, activation='softmax')

        self.loss_fn = loss_fn
        self.optimizer = optimizer
        self.train_loss = train_loss
        self.train_metric = train_metric
        self.test_loss = test_loss
        self.test_metric = test_metric

    def model_arch(self, x):
        return self.dense_layer(self.flatten_layer(x))

    @tf.function
    def train_step(self, images, labels):
        with tf.GradientTape() as tape:
            predictions = self.model_arch(images)
            loss = self.loss_fn(labels, predictions)
            gradients = tape.gradient(loss, self.trainable_variables)
        self.optimizer.apply_gradients(zip(gradients, self.trainable_variables))
        self.train_loss(loss)
        self.train_metric(labels, predictions)

    @tf.function
    def test_step(self, images, labels):
        predictions = self.model_arch(images)
        t_loss = self.loss_fn(labels, predictions)
        self.test_loss(t_loss)
        self.test_metric(labels, predictions)

    def fit(self, train, test, epochs):
        for epoch in range(epochs):
            for images, labels in train:
                self.train_step(images, labels)
            for test_images, test_labels in test:
                self.test_step(test_images, test_labels)

            template = 'Epoch {}, Loss: {}, Accuracy: {}, Test Loss: {}, Test Accuracy: {}'
            print(template.format(epoch + 1, self.train_loss.result(),
                                   self.train_metric.result() * 100, self.test_loss.result(),
                                   self.test_metric.result() * 100))
            self.train_loss.reset_states()
            self.train_metric.reset_states()
            self.test_loss.reset_states()
            self.test_metric.reset_states()
```
Example framework with model train/test functions

```python
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)
```
Training models

Functions are compiled into a graph for faster execution, running on GPU or TPU, etc.

- Constructs a callable that executes a tensorflow graph (tf.Graph)
  - Allows Tensorflow runtime to apply optimizations/exploit parallelism in the defined computation
- A normal keras layer
  
  \[
  \text{tf.nn.relu}(x)
  \]
- With function annotation
  
  ```
  @tf.function
  def relu_layer(x):
      return tf.nn.relu(x)
  ```
Useful tips for debugging

- With eager execution, execution happens line by line
  - Simple checks (shape of tensors, etc.) can be done in-line

- Tensorboard
  - Tool for measurements and visualizations (very useful!)
    - Ensuring loss decreases over time
    - Tracking training and validation accuracy over time
    - Visualizing model graph
    - Projecting embeddings to lower dimensional space
    - Much more!
Useful tips for debugging

Tensorboard

- **Keras API** `model.fit`
  - Specify callbacks (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)])` 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)` logs training loss per epoch
Tensorflow 2.0 Docs:
https://www.tensorflow.org/api_docs/python/