Theano cheatsheet

(for detailed information on Theano see: http://deeplearning.net/software/theano)

Imports:

- `import theano` #main module
- `from theano import tensor as T` #variables & vector/matrix/tensor operations

Symbolic variables

- `X = T.matrix()`
- scalar, vector, matrix, tensor, etc.
- Type can be specified by using i/f/d for integer/float32/double: `X = T.ivector()`

Operations with symbolic variables

- Basic elementwise operations: +, -, *, /, **, etc.
- Dot product: `T.dot(A, B)` or `A.dot(B)`
- Aggregations: `T.min()`, `T.max()`, `T.mean()`, `T.sum()`, etc.
    - axis: only aggregate along the specified dimension
        - Maximum per column of matrix A: `T.max(A, axis=0)`
        - Sum of rows of matrix A: `T.sum(A, axis=1)`
- Reshaping
    - `A.reshape((10, 1))`, `A.reshape((10))`, `A.reshape((2,5))`, etc.
    - `A.dimshuffle((1,0))`, `A.dimshuffle(('x', 0))`, etc.
        - Use already existing dimension indeces and use 'x' for a new broadcastable dimension (e.g. ('x', 0) creates a matrix row (2D) from a 1D vector))
    - `A.T` #transpose
- Subtensor operators
    - Indexing as in numpy (Might be slow on GPU! Indexing 2D matrices by row is fast.)
    - Slicing similar to numpy
    - `T.set_subtensor(subtensor, value)`
    - `T.inc_subtensor(subtensor, inc)`

Shared variables

- Variables with persistent value
- Ideal for model parameters
- `w = theano.shared(np.random.rand(100).astype('float32'))`
    - Initialized with numpy arrays (determines shape, size, type and starting values)
        - In the example above: a vector of 100 float32 values
    - Optional name parameter
- Can be used as a variable in symbolic expressions
- Any operation with shared variables results in a symbolic experession/variable
- `W.get_value()`, `W.set_value(new_value)`
    - Getting/setting the value of a symbolic variable
    - Use only when necessary
    - borrow parameter

        - False: always copy the given array
        - True: reference the given array when possible

Computational graph

- Operations with variablers produce other symbolic variables
- Computational graph
    - Node: a symbolic expression resulting in a variable
    - Directed edges
        - In: from variables needed for computing the node
        - Out: to variables using this variable for computations

Functions

- Parts of the computation graph compiled into a function
    - C = A + B
    - `add = theano.function([A,B], C)`
- theano.function
    - First parameter: list of inputs
        - Always a list, even if there are no inputs
        - All variables we need to provide value for to do the computations
        - Shared variables must not be in the list (they already have value)
    - Second parameter (out): output or list of outputs if there are multiple
        - Optional
    - update: dictionary of updates of the shared variables
        - Optional
- Calling the function: `res=add(1,1)`

Gradient

- `T.grad(X, wrt=W)`
    - Computes the gradient of the scalar X with respect to paramaters W
    - W must be a part of the computations leading to X

Updating shared varibles

- Update dictionary: OrderedDict that contains shared variable & symbolic variable pairs of the same shape, size and type
    - `from collections import OrderedDict`
    - `updates = OrderedDict()`
    - `updates[W] = 2*W`
- Use the dictionary in the update parameter of a function
    - `mult_by_2 = theano.function([], updates=updates)`
    - The shared variable updates every time the function is called based on the rule defined by the update dictionary

Random number generation

- `from theano.sandbox.rng_mrg import MRG_RandomStreams as RandomStreams`
- `srng = RandomStreams() #random stream`
- `srng.binomial((10,5), p=0.5, dtype=theano.config.floatX)` #generate 10x5 binomial random variables

Example: simple FNN with mini-batch stochastic gradient

```
import theano
from theano import tensor as T
import numpy as np
from collections import OrderedDict
#from theano.sandbox.rng_mrg import MRG_RandomStreams as RandomStreams


class Learner:

    def __init__(self, batch_size, learning_rate, n_hidden=0):
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.n_hidden = n_hidden

    def softmax(self, X):
        e_x = T.exp(X - X.max(axis=1).dimshuffle(0, 'x'))
        return e_x / e_x.sum(axis=1).dimshuffle(0, 'x')

    def relu(self, X):
        return T.maximum(X, 0)

    def init_model(self, n_features, n_classes):
        self.W1 = theano.shared(0.01 * np.random.rand(n_features,
self.n_hidden).astype(theano.config.floatX) - 0.005, borrow=True)
        self.B1 = theano.shared(np.zeros(self.n_hidden, dtype=theano.config.floatX))
        self.W2 = theano.shared(0.01 * np.random.rand(self.n_hidden,
n_classes).astype(theano.config.floatX) - 0.005, borrow=True)
        self.B2 = theano.shared(np.zeros(n_classes, dtype=theano.config.floatX))
        self.params = [self.W1, self.B1, self.W2, self.B2]

    def model(self, X):
        h = self.relu(T.dot(X, self.W1) + self.B1)
        y = self.softmax(T.dot(h, self.W2) + self.B2)
        return y

    def cross_entropy(self, Yhat, Y):
        return T.mean(-T.log(T.diag(Yhat.T[Y])))

    def update_model(self, cost):
        updates = OrderedDict()
        for V in self.params:
            G = T.grad(cost, wrt=V)
            A = theano.shared(V.get_value()*0., borrow=False)
            A2 = A + G**2
            V2 = V - self.learning_rate * G / T.sqrt(A2+1e-6)
            updates[A] = A2
            updates[V] = V2
        return updates

    def fit(self, data, n_steps=100000, print_n=1000, check_n=-1, test=None):
        self.init_model(data.shape[1]-1, len(np.unique(data[:,0].reshape(-1))))
        X = T.matrix()
        Y = T.ivector()
        Yhat = self.model(X)
        cost = self.cross_entropy(Yhat, Y)
        updates = self.update_model(cost)
        train_function = theano.function([X, Y], cost, updates=updates)
        c = []
        for i in range(n_steps):
            idxs = np.random.choice(len(data), size=self.batch_size)
            x = data[idxs]
            y = x[:,0].reshape(-1).astype('int32')
            x = x[:,1:].astype(theano.config.floatX)
            c.append(train_function(x, y))
            if ((i+1) % print_n) == 0:
                print(i+1, np.mean(c))
                c = []
            if check_n > 0 and ((i+1) % check_n) == 0:
                print('Test classification accuracy after {} steps: {}'.format(i+1,
self.classification_accuracy(test)))

    def predict(self, data):
```

```
    if not hasattr(self, 'predict_function'):
        X = T.matrix()
        Yhat = self.model(X)
        self.predict_function = theano.function([X], Yhat)
    yhat = self.predict_function(data)
    return yhat

def classification_accuracy(self, test):
    labels = test[:,0]
    test = test[:,1:]
    yhat = np.argmax(self.predict(test), axis=1)
    return (labels==yhat).sum()/len(labels)
```