Automatic Reverse-Mode Differentiation

TAs: Karandeep Johar, Bhuwan Dhingra

Out 11/3/2016
Due 11/17/2016 via Autolab

Prerequisite: Please ensure you have first read Prof Cohen's notes on Automatic Reverse Mode Differentiation[1]. You can also refer to the sample code here[2].

Guidelines for Answers: Please answer to the point. Please state any additional assumptions you make while answering the questions. You need to submit a tar file containing source files and a pdf version of report separately to autolab. Please make sure you write the report legibly for grading.

Rules for Student Collaboration: The purpose of student collaboration in solving assignments is to facilitate learning, not to circumvent it. Studying the material in groups is strongly encouraged. It is allowed to seek help from other students in understanding the material needed to solve a homework problem, provided no written notes are taken or shared during group discussions. The actual solutions must be written and implemented by each student alone, and the student should be ready to reproduce their solution upon request. You may ask clarifying questions on Piazza. However, under no circumstances should you reveal any part of the answer publicly on Piazza or any other public website. Any incidents of plagiarism or collaboration without full disclosure will be handled severely.

Rules for External Help: Some of the homework assignments used in this class may have been used in prior versions of this class, or in classes at other institutions. Avoiding the use of heavily tested assignments detracts from the main purpose of these assignments, which is to reinforce the material and stimulate thinking. Because some of these assignments may have been used before, solutions to them may be available online or from other people.

[1] `http://www.cs.cmu.edu/~wcohen/10-605/notes/autodiff.pdf`
[2] `http://www.cs.cmu.edu/~wcohen/10-605/code/sample-use-of-xman.py`

It is explicitly forbidden to use any such sources or to consult people who have solved these problems before. You must solve the homework assignments completely on your own. We will mostly rely on your wisdom and honor to follow this rule. However, if a violation is detected, it will be dealt with harshly.

- Did you receive any help whatsoever from anyone in solving this assignment? Yes/No
- If you answered yes, give full details:_____
(e.g. "Jane explained to me what is asked in Question 3.4")
- Did you give any help whatsoever to anyone in solving this assignment? Yes/No
- If you answered yes, give full details:_____
(e.g. "I pointed Joe to section 2.3 to help him with Question 2")

1 Overview

In this assignment we will use an automatic differentiation system to implement two neural network architectures for character level entity classification, using `python` and `numpy`. The architectures we will implement are:

- A feedforward network or Multilayer Perceptron (MLP)
- A Long Short Term Memory (LSTM) network followed by a feedforward layer

Character level entity classification refers to determining the type of an entity given the characters which appear in its name as features. For example, given the name "`Antonio_Veciana`" you might guess that it is a `Person`, and given the name "`Anomis_esocampta`" you might guess that it is a `Species`. We will be classifying the following 5 `DBPedia` categories - `Person`, `Place`, `Organisation`, `Work`, `Species`.

2 Neural Architectures

2.1 Mutlilayer Perceptron

MLP is a simple neural network architecture consisting of multiple layers, each of which apply a linear transformation followed by non-linear mapping to their inputs:

$$o_i = f(x^T w_i + b_i)$$

Here $x \in \mathbb{R}^{d_{in}}$ is the layer input and $o_i \in \mathbb{R}$ is the i-th output of the layer. w_i, b_i are layer parameters which will be optimized during learning. The number of outputs at each layer is called the *dimension* of that layer and we denote it by d_{out}. We would like to use vector/matrix multiplications wherever possible to utilize their fast implementation in `numpy`, and combine the above for all i as:

$$o = f(x^T W + b)$$

$W = [w_1, w_2, \ldots, w_{d_{out}}] \in \mathbb{R}^{d_{in} \times d_{out}}$ stacks all the weight vectors horizontally, and $b \in \mathbb{R}^{d_{out}}$ holds all the biases. The non-linearity f is applied elementwise.

To further speed-up the computation we can process a minibatch of inputs together. Let $X \in \mathbb{R}^{N \times d_{in}}$ be a matrix holding N examples row-wise. We can compute the layer outputs for all of these together:

$$O = f(XW + B) \tag{1}$$

$B = \mathbf{1} \otimes b^T \in \mathbb{R}^{N \times d_{out}}$ is a "broadcasted" version of the bias of appropriate dimensions. For this assigment we will use `numpy` for all matrix operations, which takes care of broadcasting automatically (see here[3] for details), hence we can use the vector b directly. The nonlinearity we will use is the `Rectified Linear Unit (ReLU)`:

$$f(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$$

For vectors the nonlinearity is applied element-wise, and we can again use numpy broadcasting for this. In multi-layer networks, output of layer k is passed as input to layer $k+1$:

$$O^{(k+1)} = f(O^{(k)} W^{(k)} + b^{(k)}) \tag{2}$$

[3] `https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html`

We can set output size of the last layer of the MLP to produce a vector the same size as the number of labels C in our dataset. The operations described thus far map inputs to positive reals, but for classification tasks we are interested in obtaining a *distribution* over class labels. This is usually done by passing the output of the last layer through a `softmax` operation:

$$p_j = \frac{e^{o_j}}{\sum_{j'=1}^{C} e^{o_{j'}}}$$

Note that p defines a valid distribution, and elements of o which have a high relative value will have a high probability in p. In case o_js are very large or very negative there might be numerical issues in computing the above. A more numerically stable version of softmax uses the following:

$$p_j = \frac{e^{o_j - a}}{\sum_{j'=1}^{C} e^{o_{j'} - a}} \tag{3}$$

This is true for any a, we will use $a = \max_j o_j$. Lastly, we need to define a loss function which measures how far the output distribution p_i for input i is from its target distribution t_i. We will use the cross-entropy loss for this:

$$l_i = -\sum_{j=1}^{C} t_i^{(j)} \log p_i^{(j)} \tag{4}$$

For single-label classification, t_i is an C-dimensional one-hot vector encoding the correct label for this example. The above equations compute p and l for a single o, but in your code you should use `numpy` operations to compute a minibatch of distributions P and losses L from a minibatches of O. The objective function we will optimize is the average of losses across a minibatch:

$$\text{loss} = \frac{1}{N} \sum_{i=1}^{N} l_i \tag{5}$$

Now we can take gradients of loss wrt to the parameters of the network and perform Stochastic Gradient Descent (SGD).

To summarize, the first architecture you will implement for this assignment consists of an MLP with one hidden layer, followed by a softmax layer

(3) and cross-entropy loss (4). Given an input minibatch X and their associated targets T, the output P and loss are computed as:

$$\begin{aligned} O^{(1)} &= \text{relu}(XW^{(1)} + b^{(1)}) \\ O^{(2)} &= \text{relu}(O^{(1)}W^{(2)} + b^{(2)}) \\ P &= \text{softmax}(O^{(2)}) \\ \text{loss} &= \text{mean}(\text{cross-entropy}(T, P)) \end{aligned}$$

2.2 Long Short Term Memory

The MLP is a powerful model – with enough hidden units it can approximate any function, but it is not the most appropriate model when the input is a sequence. For sequences, the input size of the MLP and consequently size of $W^{(1)}$, will increase linearly with the size of the length of the sequence and might get prohibitively large. Instead, we would like to have a model which can loop over the input sequence, and starting from an initial state iteratively updates its output based on the input at that time step. LSTMs are one example of such a model [4].

Lets say we have a sequence of inputs $x_1, x_2, \ldots, x_M \in \mathbb{R}^{d_{in}}$, an initial *cell state* $c_0 = \mathbf{0} \in \mathbb{R}^{d_{out}}$ and an initial *output* $h_0 = \mathbf{0} \in \mathbb{R}^{d_{out}}$. At time t the LSTM does the following updates:

$$\begin{aligned} i_t &= \sigma(x_t^T W_i + h_{t-1}^T U_i + b_i) \\ f_t &= \sigma(x_t^T W_f + h_{t-1}^T U_f + b_f) \\ o_t &= \sigma(x_t^T W_o + h_{t-1}^T U_o + b_o) \\ \tilde{c}_t &= tanh(x_t^T W_c + h_{t-1}^T U_c + b_c) \\ \\ c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\ h_t &= o_t \odot tanh(c_t) \end{aligned}$$

Here $W_* \in \mathbb{R}^{d_{in} \times d_{out}}$, $U_* \in \mathbb{R}^{d_{out} \times d_{out}}$, $b_* \in \mathbb{R}^{d_{out}}$ are parameters, $\odot$ is an element-wise product and σ is the sigmoid function (applied element-wise):

$$\sigma(x)_i = \frac{1}{1 + e^{-x_i}} \tag{6}$$

[4] An introduction to LSTMs can be found at `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Note: The above equations are shown for single inputs x for clarity. In your implementation you must use minibatches X of N examples at a time, the same way as we did for the MLP. This amounts to replacing x_t^T with matrices X_t of size $N \times d_{in}$, and h_t^T, c_t^T with matrices H_t, C_t of size $N \times d_{out}$.

Now we are ready to implement the second architecture for this assignment. We will replace the first layer of MLP in the previous section with an LSTM layer. Let $\text{LSTM}(X_1, X_2, \ldots, X_M)$ be a function which loops over the sequence $X_1, \ldots, X_M$, performs the updates described above, and returns the final output H_M. Then, given inputs $X_1, \ldots, X_M$, the output P and loss are computed as follows:

$$\begin{aligned} H_M &= \text{LSTM}(X_1, X_2, \ldots, X_M) \\ O^{(2)} &= \text{relu}(H_M W^{(2)} + b^{(2)}) \\ P &= \text{softmax}(O^{(2)}) \\ \text{loss} &= \text{mean}(\text{cross-entropy}(T, P)) \end{aligned}$$

2.3 Learning

Parameters of the above networks can be trained using minibatch SGD. Once the loss function is defined we can take its derivative wrt to any parameter w_{ij} and update it as follows:

$$w_{ij}^{(k)} \leftarrow w_{ij}^{(k-1)} - \lambda \frac{d\text{loss}}{dw_{ij}} \tag{7}$$

λ is the learning rate.

3 Autodiff Implementation

In the starter code the following files are given -

- `xman.py` – classes for expression manager, registers and operations
- `utils.py` – classes for data preprocessing and forming minibatches (more on this below)
- `functions.py` – function definitions and their gradients are declared here

- `mlp.py` – you need to implement the mlp here
- `lstm.py` – you need to implement the lstm here
- `autograd.py` – class for forward and backward propagation over a Wengert list

Here we briefly outline the steps needed to implement a model using the package provided with the handout.

3.1 Declare operations, their functions and gradients

First we must declare all the primitive operations our function uses in the `XManFunctions` class. For example, to declare `relu`, add the following lines to `f` class in `functions.py`:

```
class f(XManFunctions):
    @staticmethod
    def relu(a):
        return XManFunctions.registerDefinedByOperator('relu',a)
```

`add`, `mul` and `subtract` are decalred by default in `XManFunctions`.

Next we need to define functions for both the forward and backward pass for each of the operators in our definition. The following example shows the definitions for `add`:

```
import numpy as np
# forward pass
EVAL_FUNS = {
    'add':      lambda x1,x2: x1+x2,
}

def _derivAdd(delta,x1):
if delta.shape!=x1.shape:
    # broadcast, sum along axis=0
    return delta.sum(axis=0)
else: return delta

# backward pass
BP_FUNS = {
```

```
    'add':      [lambda delta,out,x1,x2: _derivAdd(delta,x1),
                      lambda delta,out,x1,x2: _derivAdd(delta,x2)],
    }
```

Lets take a closer look at what is happening here. `EVAL_FUNS` is a dictionary whose keys are the names of the operators as declared in the previous section and values are the actual functions themselves (usually defined using lambda calculus). `BP_FUNS` is another dictionary with the same set of keys as `EVAL_FUNS`, but whose value for a key is a list of functions each computing its gradient wrt one of its inputs.

In the above example `BP_FUNS['add'][0]` computes the derivative wrt `x1`, and `BP_FUNS['add'][1]` computes the derivative wrt `x2`. As input each of these functions receives:

- `delta` - partial derivative of the output of this operation
- `out` - output of this operation in the forward pass. This can be sometimes useful for computing the derivative. For example, for the sigmoid nonlinearity $\sigma'(x) = \sigma(x)(1 - \sigma(x))$.
- `x1`,`x2`,... - all inputs to the operation

Some things to be careful about:

1. Remember that our functions need to run in minibatch mode, where they receive matrices as input and produce matrices as output.
2. Ensure that the shape of a gradient matches the shape of the input wrt it is computed. As an example consider matrix multiplication $M = A \cdot B$ where A is $n \times m$ B is $m \times k$, then both M and the backpropagated derivative of the loss w.r.t. M ($\frac{d\text{loss}}{dM}$) will be $n \times k$. To get the partial derivates of the loss w.r.t. A and B:

$$\frac{d\text{loss}}{dA} = \frac{d\text{loss}}{dM} \cdot B^T$$
$$\frac{d\text{loss}}{dB} = A^T \cdot \frac{d\text{loss}}{dM}$$

   In our code $\frac{d\text{loss}}{dM}$ will be stored in `delta[M]`.

3. Be careful about `numpy` broadcasting. An example of this is shown in `_derivAdd`. In numpy if we add a $N \times d$ matrix and a size d vector, the vector is broadcasted to the size of the matrix. In this case the gradient coming back will be size $N \times d$, and needs to be summed along the first dimension for the vector (but not for the matrix!). This happens for example when we write $XW + b$.

4. `autograd.py` (described below) has an optimization which combines the backward pass for `crossEnt` and `softMax` operations. The gradients of this combined operation are easier to compute than the individual ones. So you need to implement `BP_FUNS` only for the combined operation `crossEnt-softMax`, but implement them separately in `EVAL_FUNS`.

3.2 Describe the Model

Once the primitive operations are defined, we can go ahead and define the model. First we need to declare registers to hold inputs and parameters, suppose there is one input `x`, a target `y` and two parameters `W` and `b`:

```
x = f.input(name='x', default=np.random.rand(1,10))
y = f.input(name='y', default=np.random.rand(1,10))
W = f.param(name='W', default=a*np.random.uniform(10,10))
b = f.param(name='b', default=0.1*np.random.uniform(10,))
```

You **must** specify the `name` and `default` fields for each register, including `input` registers! The `name` is used during the forward and backward passes to bind values to the correct register (more on this later). The `default` value is used as initialization for parameters and also for performing gradient checks. We will use the `inputDict` method described in the next section to collect values for all registers and perform gradient checking using that. For this purpose, you can assign any random default value to the `input` registers, as long as it is the right shape.

Note on initialization of parameters: As discussed in class, it is important to initialize parameters such that intermediate values in the network do not lie in the saturated regions of the non-linearity. One good heuristic is to sample the weights for W of size $d_{in} \times d_{out}$ from a uniform distribution

$\mathcal{U}[-a, a]$ whose scale a is given by:

$$a = \sqrt{\frac{6}{d_{in} + d_{out}}} \tag{8}$$

This is called Glorot initialization[5]. Bias terms can be initialized at a scale of 0.1.

Now write the model in terms of primitive operations:

```
xm = XMan()
xm.o1 = f.relu( f.mul(x,W) + b )
...
xm.loss = ...
my_xman = xm.setup()
```

We can construct the Wengert list for any register in `my_xman` by calling `operationSequence()`.

```
# wengert list for the 'loss' register
wengert_list = my_xman.operationSequence(my_xman.loss)
```

3.3 Forward / Backward Pass

Another useful method in the `XMan` class is `inputDict()`, which returns a dictionary containing default values associated with its registers:

```
valueDict = my_xman.inputDict()
# valueDict contains register names as keys
# and defaults as values
```

At the beginning of training we will call `inputDict` to collect initial values of all parameters (`valueDict` also contains default values for the data, but we will overwrite this). Then we will iterate through minibatches of the data (more on this below), write the inputs to `valueDict` and perform updates.

One update consists of propagating data and parameters through the Wengert list to get the outputs, and backpropagating gradients to get the updates for all parameters. The `Autograd` class in `autograd.py` provides helper functions to do this.

[5] `http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf`

```
# valueDict contains current value of params
# xc and yc are inputs for current minibatch
valueDict[‘x’] = xc
valueDict[‘y’] = yc
ad = Autograd(my_xman)
valueDict = ad.eval(wengert_list, valueDict) # forward pass
gradients = ad.bprop(wengert_list, valueDict,
                     loss=np.float_(1.))    # backward pass
```

The last argument specifies gradient of the loss which will be backpropagated. `gradients` is a dictionary whose keys are register names and values are gradients. These can be used to apply updates to the parameters as follows:

```
for rname in grads:
    if my_xman.isParam(rname):
        valueDict[rname] = # do this yourself
```

3.4 Gradient Checking

How do we check if our implementation is correct? One way to do that is gradient checking. Recall from calculus that for any function $J(\theta)$:

$$\frac{dJ(\theta)}{d\theta} = \lim_{\epsilon \to 0} \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon} \tag{9}$$

Thus for any specific value of a parameter θ we can numerically approximate the right hand side above by evaluating loss of the network at $\theta + \epsilon$ and $\theta - \epsilon$. For small ϵ the numerical gradient should match the one computed using autodiff. You can use $\epsilon = 10^{-4}$ and check if gradients match up to 3 decimal places.

4 Data

For this assignment you need to predict the category label of a DBPedia entity based on its title. The data format is **different** than the earlier assignments. The data contains two columns separated by tab, with the title in the first column and label in second.

```
Lloyd_Stinson   Person
Lobogenesis_centrota    Species
Loch_of_Craiglush   Place
```

We will encode entities for input to the networks by converting characters to a one-hot representation. Suppose we have a dictionary mapping each character in the data to an index `chardict = {'a':1,'b':2...}` and the total number of characters in the dataset is V, then we will represent `'a'` as a V-dimensional vector $[1, 0, 0, \ldots, 0]$, and `'b'` as another V-dimensional vector $[0, 1, 0, \ldots, 0]$. A string of characters will be encoded to a matrix whose each row is a V-dimensional vector.

For both MLP and LSTM we will fix the maximum length of an entity to M, longer entities will be truncated to this length, and shorter ones will be padded with white-space. We have provided you with code that preprocesses the data and divides it into minibatches in utils.py. You can,

```
from utils import *
# load data and preprocess
dp = DataPreprocessor()
data = dp.preprocess(<training_file>, <validation_file>,
       <testing_file>)
# minibatches
mb_train = MinibatchLoader(data.training, batch_size, max_len,
           len(data.chardict), len(data.labeldict))
mb_valid = MinibatchLoader(data.validation, len(data.validation),
           max_len, len(data.chardict), len(data.labeldict),
           shuffle=False)
mb_test = MinibatchLoader(data.test, len(data.test), max_len,
           len(data.chardict), len(data.labeldict), shuffle=False)
```

`max_len` is the maximum length M which we set. `shuffle=True/False` tells the batch loader whether to shuffle the data after every epoch. For validation and test sets we set the batch size same as the size of the dataset. You can then iterate over the data using,

```
for (idxs,e,l) in mb_train:
    # idxs - ids of examples in minibatch
    # e - entities in one-hot format
    # l - corresponding output labels also in one-hot format
```

After every epoch (full sweep through `mb_train`) the data is shuffled for the next epoch in `mb_train`. `idxs` has shape N, e has shape $N \times M \times V$ and l has shape $N \times C$ where N is the batch size. Make sure that this makes sense to you.

For input to the MLP we will concatenate all the one-hot encodings into one row vector, so you will need to flatten `e` to a $N \times MV$ size matrix whose each row consists of the encoding of all characters in the entity one after the other. You can use `numpy.reshape` for this.

For input to the LSTM, we will create a sequence of inputs $X_1, \ldots, X_M$ from `e`. Each of these would be a $N \times V$ matrix holding the batch inputs from time-step 1 through M. Since we are picking the *final* state of the LSTM for classification, **we feed inputs to the network in reverse order so that the useful characters appear at end.**

You are given two datasets for this assignment - `tiny` for debugging, and `smaller` for reporting results. The `Data` and `MiniBatchLoader` classes create dictionaries for all characters and labels in the dataset and use that to encode the inputs and labels into a one-hot vector format.

A validation file (*.valid files) is provided to prevent overfitting. You should evaluate the loss function on the validation dataset after every epoch and store the parameters of the best model in a separate dictionary. Then after training is completed, use these best parameters to make predictions on the test set. Rememeber you should not do backpropagation on the validation dataset. For reporting the results you use test (*.test) files.

You can get the data from

`https://drive.google.com/open?id=0B58rr945j04BcGctanY1UHVFZVE`

For the evaluation on Autolab we will run your code on a separate train, validation and testing dataset `autolab` with a specific set of params as command line arguments. This is smaller than the `smaller` dataset provided to you, so you should be careful about overfitting.

5 Deliverables

You need to write code for building, training and evaluating an MLP and LSTM in mlp.py and lstm.py respectively.

You need to write your function definitions and their derivatives in functions.py. Make sure that you call the final predicted labels as `outputs` and

the loss function as `loss` in the `XMan` object. You should also remember to initialize the default parameters and inputs. Otherwise you'd get zero score on our gradient checking code.
You should run your code using the following defaults:

```
python mlp.py --max_len 10 --num_hid 50
              --batch_size 64 --dataset autolab
              --epochs 15 --init_lr 0.05
              --output_file output
python lstm.py --max_len 10 --num_hid 50
               --batch_size 64 --dataset autolab
               --epochs 15 --init_lr 0.5
               --output_file output
```

In autolab we will call the main functions in `mlp.py` and `lstm.py` and pass a dictionary of hyperparameter values. We will check your code for gradient correctness, mean Loss on the output, memory and speed compared to the benchmark code written by us. You will receive extra credit if your loss on the test set is better than ours, but be careful about making your architecture more complex – you may lose marks on speed and memory checks.

Your final output probabilities should be stored in the file specified by –output-file in numpy format using np.save() in the same row order as the input test file. Tar the following files for submission.

- mlp.py (This file would contain the MLP class and the main file with the default params)
- lstm.py (This file would contain the LSTM class and the main class with the default params)
- functions.py (This file would contain the function definitions and their gradients)
- autograd.py
- Any other helper files

Tar the files directly using the command below. Do NOT put the above files in a folder and then tar the folder. You do not need to upload the saved temporary files.

```
tar -cvf hw5.tar lstm.py mlp.py functions.py autograd.py
```

Also, please do not forget to submit your report hw5.pdf via the HW5: Report link in Autolab.

6 Questions

1. Plot the average training time per epoch for both MLP and LSTM for batch size $N = \{16, 32, 64\}$, and default values of remaining hyperparameters. What trends do you observe for the two architectures? Are they similar? Why or why not?

2. Plot the average training time per epoch for both MLP and LSTM for maximum input length $M = \{10, 15, 20\}$, and default values for remaining hyperparameters. What trends do you observe for the two architectures? Are they similar? Why or why not?

3. Plot the total number of forward and backward steps in one update of the LSTM for maximum input length $M = \{10, 15, 20\}$, and default values for remaining parameters. Now repeat the plot for batch size $N = \{16, 32, 64\}$. One forward step is defined as computing the output of a primitive operation, and one backward step is defined as computing the derivative of a primitive function wrt *one* of its inputs. Briefly explain the trends that you observe.

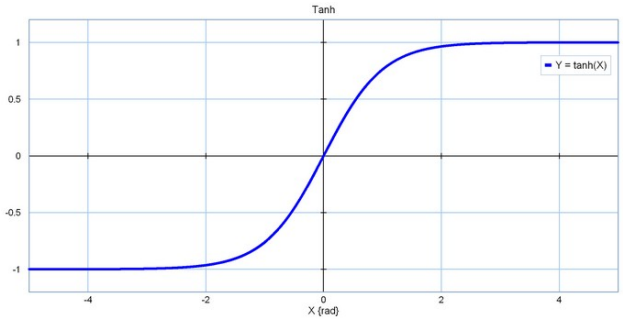
   Hint: You can count the forward and backward steps by placing counters in `Autograd.eval` and `Autograd.bprop` respectively.

4. Given feature vectors for tokens in a query $q_1, q_2, \ldots q_M$ and feature vectors for tokens in a document $d_1, d_2, \ldots d_N$, where $q_i, d_j \in \mathbb{R}^n \quad \forall i, j$, the *soft-attention* distribution of ith query token over the document is given by:

$$\alpha_j^{(i)} = \frac{\exp q_i^T d_j}{\sum_{j'} \exp q_i^T d_{j'}} \qquad j = 1, \ldots N \tag{10}$$

   Given a $M \times n$ matrix Q and an $N \times n$ matrix D holding all the query and document feature vectors respectively, write the steps needed to compute all the attention distributions for $i = 1, \ldots M$, using only matrix operations. You can use broadcasting, and assume efficient implementaions of `row-sum` and `exp` are given.

5. Consider a single-layer MLP without bias which computes $f = \tanh(\sum_{i=1}^{1000} x_i w_i)$ where $x_i, w_i \geq 0 \quad i = 1, \ldots 1000$. Recall that $\tanh(x)$ looks like:



We initialize the weights $w_i \sim \mathcal{U}[0, a]$. Choose the scale a for the following inputs:

- One-hot – $x_i = 1$ for $i = i_0$ and $x_i = 0$ for $i \neq i_0$.
- Small – $x_i \in [0, 1]$ for all i.
- Big – $x_i \in [1000, 1001]$ for all i.

7 Marking breakdown

- Code Correctness (gradient check)[6] - 30 points
- Code Accuracy (loss on held-out test set) - 20 points
- Code Speed and Memory usage - 20 points
- Report Questions - 30 points

[6] **Reminder** - You must name the loss register as "`loss`" for our gradient checking to work