FACULTY OF
BIOSCIENCE ENGINEERING

HYPERDIMENSIONAL COMPUTING FOR PROTEIN LANGUAGE MODELING

...

Michael Fatjanov
Student ID: ...

Supervisor(s): Prof. Dr. Bernard De Baets

A dissertation submitted to Ghent University in partial fulfilment of the requirements for the degree of master in Bioinformatics.
Academic year: 2022-2023

GHENT
UNIVERSITY

Gent, FILL IN THE DATE

The promotor,	The author,
Prof. Dr. Bernard De Baets	Michael Fatjanov

ACKNOWLEDGEMENTS

Thank you, all of you!

CONTENTS

SAMENVATTING

nederlandse samenvatting

SUMMARY

insert english summary here...

1. INTRODUCTION

1.1 Digital biology, protein sequence research and traditional bioninformatics tools

1.2 State-of-the-art, deep learning and protein language modeling

1.3 Hyperdimensional computing

Hyperdimensional computing is a relatively new paradigm of computing developed by Kanerva.Kanerva (2009) that tries to mimick the workings of a (human) brain by computing with vectors of tens of thousands elements long, so in the realm of hyperdimensionality. The human brain consists of about 100 billion neurons (nerve cells) and 1000 trillion synapes that connect these neurons. Each neuron is connected up to 10000 other neurons, creating massive circuits. This is likely fundamental to the workings of the human brain and what seperates our brains from modern von Neumann computer architectures which operate on 8 to 64 bit vectors. This becomes clear when we compare the relative simplicity for a human to learn a language compared to computers which require a large and complecated set of arithmetic operations in the form of deep learning networks together with terabytes of data and thousands of Watts of compute power to try to come close mastering a language whilst a human can recognize other languages relatively easy when they don't even speak it. Likewise languages, we can very easily memorize and compare other intrisically complex and contextual concepts such as images. A computer would have a hard time finding similarity between a set of images and faces because this requires very complex machine learning models. The human brain can do this all with a huge efficiency by uconsuming only roughly 20 W of energy.

To achieve this kind of flexible intelligence, we might have to step away from the restrictive von Neumann architecture and so Kanerva proposes the use of hyperdimensional vectors, a different form of representation for entities by which a computer can compute with. The use of models based on high dimensionality is not entirely new and is being explored since the 1990s. Some of these earlier models include Holographic Reduced RepresentationsPlate (1995), Spatter CodeKanerva (1994) etc. An HDV can represent anything from a scalar number to any kind of concept. This vector is initially made up of totally random elements, but with a simple set of operations which will be explained later, we can use other vectors to combine some concepts into new similar or dissimilar concepts. For example to show the essence of HDC and how it tries to simulate the brain, we can compare the concept of a *table* to the concept of a *brocolli*. We would not immediately conclude that they are in any way similar but as humans we can trace back *table* to *plate* which has some similarities with *food* from which we can easily extract the concept of *brocolli*. These kind of operations are not very obvious for a classical computer but easy for us humans.

The elements in an HDV can be made up from binary bits like in classical computing but also of bipolar or real numbers. The choice of the nature of the elements has also implications on the nature of the different operations and on the results. Highly efficient bit operations could be used on binary vectors but then the amount of information stored in such a vector would be drastically lessened compared to bipolar or real vectors, leading to a lower accuracy.

An initial HDV is made up totally random. This *holistic* or *holographic* representation of a concept smeared out over a vector consisting of thousands of bits gives rise to interesting properties of this paradigm such as its robustness. These kind of systems are very tolerant to noise and failure of bits since we introduce a lot of redundancy in the vector just by stochastics. This is very unlike classical computing where every bit counts and one failure in a bit can lead to disasters. Again, this

BIBLIOGRAPHY

Kanerva, P. (1994). The spatter code for encoding concepts at many levels. pages 226–229.

Kanerva, P. (2009). Hyperdimensional Computing: An Introduction to Computing in Distributed Representation with High-Dimensional Random Vectors. *Cognitive Computation*, 1(2):139–159.

Plate, T. (1995). Holographic reduced representations. *IEEE Transactions on Neural Networks*, 6(3):623–641.

APPENDIX A

MODEL DEVELOPMENT CODING

A.1 Pseudocode of the presented algorithm

Algorithm 1: How to write algorithms

```
Data: this text
Result: how to write algorithm with LaTeX2e
initialization;
while not at end of this document do
    read current;
    if understand then
        go to next section;
        current section becomes this one;
    else
        go back to the beginning of current section;
    end
end
```

A.2 Sensitivity base class code

```python
import os
import numpy as np

from parameter import *
import matplotlib.pyplot as plt
from matplotlib.ticker import FixedLocator, MaxNLocator

class SensitivityAnalysis(object):
    """
    Base class for the Sensitivity Analysis

    Parameters
    ----------
    ParsIn : list
        ModPar class instances in list or list of (min,max,'name')-tuples

    Attributes
    -----------
    ParsIn : list
        a list of (min,max,'name') values,
        [(min,max,'name'),(min,max,'name'),...(min,max,'name')]
    parmap : dict
        tracks the sequence of the parameters
    Pars : list of ModPar instances
        Used when working with the pyFUSE package
    ndim :  int
        number of uncertain input factors
    namelist : list
        list of the uncertain input factors used

    """

    def __init__(self,ParsIn):
        '''
        Check if all uniform distribution => TODO ! if all -> sobol sampling
        is possible, else, only uniform and normal distribution are supported
        for using the sobol sampling... Here is still work to do!!
        '''

        if  isinstance(ParsIn, dict): #bridge with pyFUSE!
            dictlist = []
            for key, value in ParsIn.iteritems():
                dictlist.append(value)
            ParsIn = dictlist
            print ParsIn

        #control for other
        self.ParsIn = ParsIn
        self.parmap={} #dictionary linking ID and name, since dict instance has no intrinsic sequence
        for i in range(len(ParsIn)):
            if isinstance(ParsIn[i], ModPar): #or isinstance(ParsIn[i], pyFUSE.parameter.ModPar):
                cname = ParsIn[i].name
                self.Pars = ParsIn
                self.ParsIn[i] = (ParsIn[i].min, ParsIn[i].max, cname)
                self.parmap[i] = cname

            elif isinstance(ParsIn[i],tuple):
                if ParsIn[i][0] > ParsIn[i][1]:
                    raise Exception('Min value larger than max value')
                if not isinstance(ParsIn[i][0],float) and isinstance(ParsIn[i][1],float):
                    raise Exception('Min and Max value need to be float')
                if not isinstance(ParsIn[i][2],str):
                    raise Exception('Name of par needs to be string')
                self.parmap[i] = ParsIn[i][2]
                #create modpar instance of the tuple
                self.Pars=[]
                for par in ParsIn:
                    self.Pars.append(ModPar(par[2],par[0],par[1],(par[0]+par[1])/2.,'randomUniform'))
            else:
                raise Exception('The input type for sampling not correct,\
                choose ModPar instance or list of (min,max)-tuples')

        self.ndim=len(ParsIn)

        self.namelist = []
        for i in range(self.ndim):
            self.namelist.append(self.parmap[i])

    def WritePre(self,filename = 'inputparameterfile', *args, **kwargs):
        '''
        Parameterinputfile for external model, parameters in the columns files
        and every line the input parameters

        Parameters
        -----------
        filename : str
            name of the textfile to save
        *args, **kwargs : args
            arguments passed to the numpy savetext-function
        '''

        np.savetxt(filename,self.parset2run,*args,**kwargs)
        print 'file saved in directory %s'%os.getcwd()

    def ReadRuns(self,filename, *args, **kwargs):
        '''
        Read model outputs (TODO: do sobol for multiple outputs,
           iterating the
        post)
        Format is: every output of the ithe MC on ith line

        output2evaluate can also be made on a other way

        Parameters
        -----------
        filename : str
            name of the textfile to load
        *args, **kwargs : args
            arguments passed to the numpy loadtext-function

        '''
        self.output2evaluate = np.loadtxt(filename, *args, **kwargs)
```

A.3 Model input file for PyFUSE model

```
######################################################
##    Model Parameter input file
##    The parameter is defined by his distribution,
##    boundaries and extra info needed by distribution
##    provide on each line one parameter with
##    following information:
##    name : string
##        Name of the parameter
##    minval : float
##        Minimum value of the parameter distribution
##    maxval :  float
##        Maximum value of the parameter distribution
##    optguess : float
##        Optimal guess of the parameter, must be
##        between min and max value
##    pardistribution : string
##        choose a distributionfrom: randomUniform,
##        randomTriangular, randomTrapezoidal,
##        randomNormal, randomLogNormal
##    *kargs  :
##        Extra arguments necessary for the
##        chosen distribution
##  Lines with ## marks are neglected
######################################################
## NAME MIN MAX OPTGUESS DISTRIBUTION ARGS*
S1max 50. 5000.000 400. randomTriangular 1000.
```

```
S2max 100. 10000.000 1000. randomNormal 500. 25.
fitens 0.01 1.0 0.99 randomLogNormal  0.5 0.2
firchr 0.050 0.950 0.5 randomTrapezoidal 0.4 0.6
fibase 0.050 0.950 0.5 randomUniform
r1 0.050 0.950 0.5 randomUniform
ku 0.01 1000. 0.044 randomUniform
c 0.99 20.0 1. randomUniform
alfa 1.000 250. 150. randomUniform
psi 1.000 5.0 2.5 randomUniform
kappa 0.050 0.950 0.5 randomUniform
ki 0.001 1000. 0.00833 randomUniform
ks 0.001 10000. 0.5 randomUniform
n 1.000 10. 3. randomUniform
v 0.00001 0.250 0.004 randomUniform
vA 0.001 0.250  0.0015 randomUniform
vB 0.001 0.250 0.0015 randomUniform
Acmax 0.050 0.950 0.5 randomUniform
b 0.001 3.0 0.2 randomUniform
loglambda 5.000 10.0 7.5 randomUniform
chi 2.000 5.0 3.5 randomUniform
mut 0.010 5.0 0.6 randomUniform
be 0.99 4. 3.1 randomUniform
alfah 0.01 0.99 0.5 randomUniform
tg 0.0 0.7 0.3 randomUniform
tif 0.0 0.7 0.26 randomUniform
tof 0.0 0.7 0.12 randomUniform
ko 0.01 0.99 0.15 randomUniform
timeo 2. 48 24 randomUniform
timei 2. 250. 20 randomUniform
timeb 200. 10000. 2100. randomUniform
```