Low Dimensional Multi-resolution Analysis for Protein Structures

Teodor Marinov tmarino2@jhu.edu
Nathan Smith nsmith45@jhu.edu
Razieh Nabi
Alex Gain again1@jhu.edu
Nikhil Panu

Abstract

We consider two separate approaches for learning low-dimensional representations for protein structures. The first is based on Geometric Multi-Resolution Analysis (GMRA) Allard et al. [2012] and the second uses deep auto-encoders. We implement both models. The GMRA model is first implemented in python and then extended to make use of pyspark. The auto-encoder is implemented using Tensorflow Abadi et al. [2016]. We evaluate our representations on a "state" prediction task using simple linear classifiers. To view source code for the project, visit: `https://github.com/NathansForYou/BDSLSS_Project`

1 Introduction

TODO:Someone please write a good introduction to our project

Currently, precise physical simulations of proteins exist to better understand their properties w.r.t. conformational changes and features. [ref?] These datasets are large and high-dimensional thus there is a demand for meaningful low dimensional representations and parallelization of algorithms that work in conjuction with the datasets for the sake of improved computational efficiency and accuracy on classification tasks. In this project, we worked with spatial trajectory data from the *Molecular Dynamics Database* (MDDB) **?** project at Johns Hopkins with a focus on learning meaningful low dimensional representations.

Typically, to address the high-dimensionality of the features, packages such as MDtraj **?** are used to perform hand-crafted feature transformations such as selecting a subset of the atoms or residues and computing atoms' pairwise distances. With the goal of automatically learning practical representations, we propose two methods: The first method uses Geometric Multi-Resolution Analysis (GMRA) Allard et al. [2012], an unsupervised learning process that makes use of linear approximations at multiple scales. The second method uses a neural network autoencoder architecture inspired by GMRA and theoretical motivation about the expressitivity of neural networks with ReLU activation.

This paper is split into two parts: The GMRA method and the autoencoder method. We provide background for each method as well as the specifics of their implementations. Additionally, we describe our dataset, evaluation criteria, and results compared to a baseline. Finally, we discuss our results and future work.

2 Protein dataset and tasks

TODO: explain the dataset, state prediction task, state transition task, motivate why we want to automatically learn representations i.e. right now most representations used to solve these tasks are

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

hand-crafted by for example removing some heavy molecules or considering only certain residues. Explain the format of the dataset we are working with as well

Proteins tend to stay in stable states where energy levels are nearly constant. There are a discrete number of stable states, or energy levels, of the proteins, which we can ascertain and assign labels to. During some relatively small time intervals, a process occurs where a protein transitions from one stable state to another. This process is not completely understood, but it is crucial to important applications such as manipulating proteins for pharmeucitical purposes.

Predicting and understanding transitions is a hard task. Thus, as a starting point, necessarily ask a preceding question: Can we understand and predict a protein's state based on its spatial conformation? For this classication task, we train on spatial-temporal information supervised by discrete state labels for each timeframe.

The MDDB dataset contains protein trajectory simulation data of 892 atoms with both temporal and non-temporal information. We only make use of the spatial-temporal data, which consists of the x-y-z coordinates of each atom sampled every 250 picoseconds. Logistically, one simulation is broken into 42 files each containing around 100,000 observations. Our experiments are conducted on the first 10 files for a total of 1,000,000 observations. For state labels, the data is segmented into 1000-frame time windows where a single label at the midpoint of each frame considered representative of the entire window, i.e. the label for the 1500-th timestep determines the label for timesteps 1000 through 2000.

3 Geometric Multi-resolution Analysis

Often times one likes to assume that data in some high dimensional Euclidean space $\mathbb{R}^D$ comes from a distribution supported on some compact low-dimensional manifold isometrically embedded in $\mathbb{R}^d$ where $d << D$. When the data turns out to come from a linear subspace one can use simple dimensionality reduction techniques like Principal Component Analysis. If the data, however, lives in a non-linear space now one needs to consider non-linear transformations like Kernel PCA. Unfortunately KPCA is not known to scale well with data - computational time for n data points is $\mathcal{O}(n^3)$ and has no guarantees that it will recover the correct subspace. GMRA on the other hand provides a computationally efficient way (the order of computations only depends on parameters associated with the geometry of the underlying manifold and linearly on D) for finding good low-dimensional representations. Further it gives theoretical guarantees on how good the representations are. One more benefit of GMRA is that it provides representations at different "scales" for the underlying process which can be beneficial if we suspect that the process behaves differently at different levels. For our purposes we know that proteins change states not very often and the transition between states is quite short - so we only need fine-grained representations of our data near moments where transitions occur and the rest for the rest of the data we would be satisfied with coarse representations.

3.1 Background

We follow the notation in Allard et al. [2012]. Our setting is in a metric, measure space $(\mathcal{M}, \rho, \mu)$ with metric ρ and probability measure μ. GMRA consists of three main parts - a multi-scale partition of the data into *dyadic* cells, a linear approximation at each dyadic cell and a wavelet-type difference operators which encode the difference of representations between scales. The following assumptions are made for our data. Since we work with coordinates of atoms of the proteins we assume that each of the coordinates are some smooth enough function of time, energy and other unknown parameters so that the data originally belongs to a $C^{1+\alpha}$ ($\alpha \in (0, 1]$) compact Riemannian manifold in $\mathbb{R}^d$. We also assume that there is some independent noise added to our observations so that our data lives in a small tube of radius r around the underlying manifold. We note that r should be small enough so that the tube does not intersect itself. We also assume that the function is periodic so that we only need to observe data in a fixed time interval to be able to do predictions.

3.1.1 Multi-scale dyadic partition of our data

The first step of the GMRA procedure is to construct a multi-scale partition of the data $\{C_{k,j}\}_{k \in \mathcal{K}_j, j \in \mathbb{Z}}$ (here j indexes the scales and k indexes the partition at each scale) with the following properties

- for every $j \in \mathbb{Z}$, $\mu\left(\mathcal{M} - \bigcup_{k \in \mathcal{K}_j} C_{k,j}\right) = 0$ i.e. at each scale j the dyadic cells $C_{k,j}$ cover our data.
- for $j' \geq j$ and $k' \in \mathcal{K}_{j'}$, either $C_{k',j'} \subseteq C_{k,j}$ or $\mu\left(C_{k',j'} \bigcap C_{k,j}\right) = 0$ i.e. at finer scales each dyadic cell is either contained in a dyadic cell from a coarser scale or is disjoint from a dyadic cells in coarser scales. Note that this property also implies that each scale forms a disjoint partition of the data
- for $j < j'$ and $k' \in \mathcal{K}_{j'}$ $\exists! k \in \mathcal{K}_j$ s.t. $C_{k',j'} \subseteq C_{k,j}$ i.e. each dyadic cell at a finer scale has a "parent" dyadic cell at each coarser scale
- each $C_{k,j}$ contains $c_{k,j}$ s.t. there exist constants c_1 and c_2 for which $\mathcal{B}(c_{k,j}, c_1 2^{-j}) \subseteq \boldsymbol{C}_{k,j} \subseteq \mathcal{B}(c_{k,j}, c_2 2^{-j})$ i.e. the dyadic cells at scale j are almost like balls of radius 2^{-j} in $(\mathcal{M}, \rho, \mu)$

Such partitions exist for metric, measure spaces $(\mathcal{M}, \rho, \mu)$ with the following property - for any $x \in \mathcal{M}, r \in \mathbb{R}$ there exists a constant c independent of x and r such that $\mu\left(\mathcal{B}(x, 2r)\right) \leq c\mu\left(\mathcal{B}(x, r)\right)$ Guy [1991]. In practice a data-structure satisfying these properties is a cover-tree Beygelzimer et al. [2006].

3.1.2 Low-dimensional affine approximations

After computing the multi-scale partition the next step is to compute the affine approximations to each of the dyadic cells $C_{k,j}$. This is done by computing the eigenvalue decomposition of the auto-covariance operator of $C_{k,j}$. To be more explicit let $c_{j,k} = \mathbb{E}_\mu\left[x | x \in C_{k,j}\right]$ and let $\mathbb{E}_\mu\left[(x - c_{k,j})(x - c_{k,j})^\top | x \in C_{k,j}\right] \approx \Phi_{k,j}\Sigma_{k,j}\Phi_{k,j}^\top$ be the rank-d truncated eigenvalue decomposition of the auto-covariance operator. Let $x \in C_{k,j}$ then the affine projection operator is defined by $P_{k,j}(x) = \Phi_{k,j}\Phi_{k,j}^\top(x - c_{k,j}) + c_{k,j}$. For our low-dimensional representations we just use $\Phi_{k,j}^\top(x - c_{k,j}) \in \mathbb{R}^d$. Notice that the dominant term in computational complexity is going to come from the SVD of the auto-covariance operator which will require about $\mathcal{O}(Dd^2)$ time.

3.1.3 Encoding differences between scales - Geometric Wavelets

The final step is to compute the difference operators between scales. Let $x \in C_{k,j}$ and $x \in C_{k',j+1}$ then $Q_j(x) := P_{k',j+1}(x) - P_{k,j}(x)$. Let $\Psi_{k',j+1} = \left(I - \Phi_{k,j}\Phi_{k,j}^\top\right)\Phi_{k',j+1}$ then by equation 2.18 in Allard et al. [2012] we have $Q_j(x) = \Psi_{k',j+1}\Psi_{k',j+1}^\top\left(P_{k',j+1}(x) - c_{k',j+1}\right) + \left(I - \Phi_{k,j}\Phi_{k,j}^\top\right)(c_{k',j+1} - c_{k,j}) - \Phi_{k,j}\Phi_{k,j}^\top\sum_{l=j+1}^{J-1} Q_{l+1}(x)$. Notice that $Q_j(x)$ does not depend on x but merely on $C_{k,j}$ and $C_{k',j+1}$ and that all of the operations to compute $Q_j(x)$ require only linear time in D and quadratic time in d.

3.2 Implementation

For our implementation we use python with numpy Van Der Walt et al. [2011],sklearn Pedregosa et al. [2011] and pyspark a library for python based on spark Zaharia et al. [2010]. The implementation can be split into two parts - first we have a purely python implementation without spark, then we extend our python implementation to make use of spark. Before presenting in more detail the algorithm we introduce some notation. Let $Uin\mathbb{R}^{m \times n}$ then $U[0:d] \in \mathbb{R}^{m \times d}$ is the matrix consisting of the first d columns of U. Also $\mathbf{SVD}$ is a routine which computes the singular value decomposition of a given matrix, $\mathbf{mean}$ computes the empirical mean of the provided list of points, $\mathbf{split}$ splits the provided point-cloud into two partitions satisfying the properties listed in 3.1.1 and $\mathbf{append}$ appends given items to the end of a list. We note that in practice the expectation of a dyadic cell is replaced by the empirical mean of the points belonging to that cell and similarly the auto-covariance operator is replaced by its empirical estimate.

The general routine for constructing the GMRA can be found as pseudo-code in 1. The routine consists of a recursively constructing the dyadic cells $C_{k,j}, C_{k+1,j}$ from its parents $C_{k',j-1}$ effectively creating a binary-tree partition of our data. After each partition the empirical means and auto-covariance operators are constructed and then used to compute the low-dimensional representations of the points in the respective dyadic cell. We also provide a routing for projecting test points onto

Algorithm 1 Compute GMRA for data X

Input: Point-cloud X, manifold dimension d, finest scale level r
Output: Array of low-dimensional representations L, array of orthogonal matrices Φ, centers of dyadic cells $centers$

$cells \leftarrow [X], \Phi \leftarrow [], L \leftarrow []$
$emp\_mean \leftarrow \mathbf{mean}\,(X))$
$centers \leftarrow [emp\_mean]$
$U\Sigma U^\top \leftarrow \mathbf{SVD}\,(X - emp\_mean)$
$\Phi \leftarrow \Phi.\mathbf{append}\,(U[0:d])$
$L \leftarrow L.\mathbf{append}\left(U[0:d]^\top\,(X - emp\_mean)\right)$
for $i = 0$ to 2^r **do**
  $X_{k,j}, X_{k+1,j} \leftarrow \mathbf{partition}\,(cells[i])$
  $cells \leftarrow cells.\mathbf{append}\,(X_{k,j}, X_{k+1,j})$
  $emp\_mean \leftarrow \mathbf{mean}\,(X_{k,j})$
  $centers \leftarrow centers.\mathbf{append}\,(emp\_mean)$
  $U_{k,j}\Sigma_{k,j}U_{k,j}^\top \leftarrow \mathbf{SVD}\,(X_{k,j} - emp\_mean)$
  $\Phi \leftarrow \Phi.\mathbf{append}\,(U_{k,j}[0:d])$
  $L \leftarrow L.\mathbf{append}\left(U_{k,j}[0:d]^\top\,(X_{k,j} - emp\_mean)\right)$
  $emp\_mean \leftarrow \mathbf{mean}\,(X_{k+1,j})$
  $centers \leftarrow centers.\mathbf{append}\,(emp\_mean)$
  $U_{k+1,j}\Sigma_{k+1,j}U_{k+1,j}^\top \leftarrow \mathbf{SVD}\,(X_{k+1,j} - emp\_mean)$
  $\Phi \leftarrow \Phi.\mathbf{append}\,(U_{k+1,j}[0:d])$
  $L \leftarrow L.\mathbf{append}\left(U_{k+1,j}[0:d]^\top\,(X_{k+1,j} - emp\_mean)\right)$
end for

the already computed GMRA. Pseudo-code can be found in 2. The routine takes a point x and at each scale assigns x to the dyadic cell for which the distance between x and the empirical mean of the dyadic cell is smallest. Because of the geometry associated with the dyadic cell partitions we only need to do a depth-first search of the binary tree.

Algorithm 2 Project test point x

Input: Test point x, Φ ,$centers$
Output: Array of low-dimensional representations $proj$ of x

$j \leftarrow 0$
while $2^{j+1} < \mathbf{len}\,(\Phi)$ **do**
  $idx \leftarrow \arg\min\left(\mathbf{dist}\,\left(x, centers[2^j]\right), \mathbf{dist}\,\left(x, centers[2^j + 1]\right)\right)$
  $j \leftarrow idx$
  $proj \leftarrow proj.\mathbf{append}(\Phi[idx]^\top(x - centers[idx]))$
end while

3.2.1 Numpy implementation specifics

In practice we replace the cover-tree partition by a partition based on k-means clustering. In the the pseudo-code 1 the routine **partition** just uses k-means clustering with 2 centers to partition the data. We note that all of the properties 3.1.1 except the last still hold. Also the **SVD** routine is switched between incremental SVD and standard SVD depending on how many points a dyadic cell contains. We also do a sort of a pruning of the tree structure by removing all dyadic cells and their associated structures if the angle between the affine approximation of a dyadic cell and its parent is close to 0. One should be careful when doing such a pruning as it is not entirely clear that there won't be non-trivial affine approximations of dyadic cells at finer scales with removed parents. By non-trivial here we mean that the angle between the affine approximation of a child and the affine approximation of a parent is larger than 0.

3.2.2 Pyspark implementation specifics

The pyspark implementation differs from the numpy implementation mainly in two ways. First if a dyadic cell is too large we create an rdd from it where each row of the rdd is a separate point in the dyadic cell represented as numpy array. We then have implemented functions which compute the mean of an rdd row-wise, mean-center the rdd and compute its truncated SVD to the first d singular values. The function which computes the SVD will return the first d right singular vectors in the form of a numpy matrix. Note that this numpy matrix is only $D \times d$. Next we compute the low-dimensional representations of the centered rdd by multiplying each row of the rdd by the orthogonal matrix we received from the SVD step.
If the number of a points in each dyadic cell at a fixed scale j is not too large we create an rdd with rows each of the dyadic cells at scale j. We then *map* our routine which computes the empirical mean, affine projection operator and low-dimensional representation of each dyadic cell to the rdd.
In general we decided to split up our spark implementation in these two ways for the following reasons. One if we have memory constraints and a dyadic cell doesn't fit into memory we can allow spark to take care of memory management by representing the dyadic cell as a separate rdd. Two if the dyadic cells at a fixed scale j are small enough so that each cell fits into memory of the *workers* we can parallelize the process of computing the projection operators and the affine approximations of each dyadic cell at scale j since all cells at a fixed scale are disjoint the operations needed to compute the required representations are independent from each other. Sadly since we are no experts in using or setting up spark this implementation runs much slower compared to the pure python implementation and thus for our experiments the representations were computed by the pyre python implementation.

3.3 Evaluation

TODO: explain what experiments were ran, what is the baseline, what classifiers were used, the observed results, etc.

4 Auto-encoder

Neural networks may be used in a wide variety of learning tasks in which current domain-knowledge or algorithmic limitations may limit researchers from defining their own learning methods. Autoencoders, a particular type of unsupervised neural network used for the learning of efficient data encodings, proved especially applicable to our challenge of protein dimensionality reduction. A basic autoencoder functions as a feed-forward, non-recurrent neural network in which there is an input layer, and output layer, and hidden layers between them working to iteratively learn computations. In the case of autoencoders, the input and output layers have an equal number of nodes, and the intention is for the output layer to accurately reconstruct its inputs.

All autoencoders consist of two core sections: the encoder, in which the input layer is fed into a sequence of hidden layers that ultimately decrease in dimensionality, and the decoder, in which hidden layers ultimately lead to the output layer of equal dimension **?**. The encoder transforms the n-dimensional input layer into a new m-dimensional feature space, where a decrease in dimensionality may be interpreted as a compressed representation of the input space. In the case of our protein dataset, an autoencoder may be used to take an initial input layer of dimension 3n, where n is the number of atoms in the dataset, and decrease it significantly in the encoding phase, with an accurate decoding reconstruction. Ideally, a central hidden layer of two or three dimensions could be produced to visually represent a protein. Proteins could then be classified by their states, and a graphical comparison of these states could be made by segmenting a two or three dimensional representation.

4.1 Theoretical motivation

One way to view what GMRA does is as a model which computes a piecewise linear approximation to an unknown smooth function (the function we associated our manifold with in the GMRA section). From Arora et al. [2016] Theorem 2.1 we know that every piecewise linear function can be represented by a deep neural network with ReLU activations. By part of the proof of this theorem we actually know that if we want to represent a piecewise linear function $f : \mathbb{R}^d \to \mathbb{R}$ with p pieces by e ReLU DNN of depth $k + 1$ we need the total number of neurons in the network to be at least $\frac{1}{2}kp^{\frac{1}{k}} - 1$

(lemma A.6). From these results we start gaining a rough idea of how our auto-encoder should look like to be able to perform as well as or even better than GMRA. Since as we already stated GMRA learns a piecewise linear function and ReLU DNNs are able to represent every linear function we decide to use a Deep ReLU auto-encoder with reconstruction error being the l^2 norm.

4.2 Architecture and Implementation

To implement an autoencoder neural network, Google's Tensorflow was set up on the DAMSL computing network. Tensorflow is a machine learning system built to operate in heterogeneous environments, using dataflow graphs to represent computation, shared state, and operations that mutate that state. Tensorflow supports Python as a front-end language, allowing for fast prototyping and testing of network designs. To train the network, a subset of 20,000 frames from one of the provided protein files was used, with another 5,000 frames then being used as a test set. Numerous different network architectures were experimented with, varying the hidden layer depth, hidden layer node counts, learning rate, activation function, loss function, number of epochs, and optimizer of the autoencoder.

The encoder was tested with between 1 and 4 hidden layers, with the same range tested on the decoder. As covered in the motivation section, ReLU, an activation function that has proven more effective than the general logistic sigmoid function in some areas, was selected as the activation function for the network, however sigmoids were also tested for comparison. The input layer featured the (x,y,z) coordinates of each of the 892 atoms in the dataset, with a total of 2676 nodes in the input and output layers. Many variations of hidden layer node counts were tested, including having all layers feature 2676 nodes until the final encoding step, in which the node count would be significantly reduced to test dimensionality reduction of the protein data. Other variations in decreasing node count per hidden layer were tested for comparison.

A variety of loss functions were also tested for the autoencoder. Initially the quadratic cost, or mean squared error, represented as $\frac{1}{j}\sum_j (true - pred)^2$, was tested. Between 5 and 50 epochs were tested, with the total test raging from approximately five minutes to one hour. The RMSProp optimizer, which utilizes the magnitude of recent gradients to normalize gradients, was initially used. RMSProp has several advantages: it is a very robust optimizer, and can deal with stochastic objectives nicely, making it applicable to mini batch learning. TensorFlow's standard gradient descent optimizer was also tested for comparison. Final evaluation of decoding output was assessed by computing a simple mean squared error between test input and decoder output. Further information on several of the attempted autoencoder implementations may be found on the project repository at `https://github.com/NathansForYou/BDSLSS_Project`.

4.3 Results

Ultimately, despite numerous experiments with the autoencoder's structure, the decoded output never closely resembled the test input. When using ReLU activation functions, it was found that the decoding output would quickly converge towards all atom coordinate values being 0, despite the squared error of this approximation remaining quite high. Potential sources of this error include failure of the autoencoder's loss function to deter network weights from converging towards zero, although given the significant error from that approximation we suspect another issue may be involved. While this autoencoder proved unsuccessful, given the motivations discussed above, we believe a neural approach to dimensionality reduction may prove successful if properly implemented. Very few sources were available with similar experiments and discussions of how an autoencoder may be optimally structured for point cloud dimensionality reduction, however further research on the subject and collaboration with researches with a greater background in machine learning may lead to a successful autoencoder.

In addition to the potential of autoencoders for atemporal protein dimensionality reduction, neural networks show strong potential for estimating protein state transitions in temporal data. As discussed in "Bidirectional Dynamics for Protein Secondary Structure Prediction" **?**, architectures based on hidden Markov models and recurrent neural networks demonstrate a significant step in predicting secondary state transitions in protein structures. Recurrent networks have been heavily used in natural

language processing and machine translation for conditional word prediction, and the problem of protein state transitions may be seen as an analogous task in a different domain.

5 Discussion and future work

TODO:

6 Contributions to project

Teodor:

- background work on GMRA
- implementation of python and pyspark versions of GMRA (Razieh provided the pyspark SVD implementation and Nathan set up pyspark on mddb2)
- theoretical justification for why a ReLU auto-encoder should learn similar representations to GMRA

Alex:

- GMRA evaluation
  - Hyperparameter tests
  - Parallelizing workflow
- Data formatting and general debugging for GMRA evaluation
- General discussion of methods and future work

Nathan:

- Initial meetings with Teodor to decide on GMRA and autoencoder implementations.
- Built and incrementally reworked/evaluated autoencoder for protein point cloud
- Set up Spark and Tensorflow on mddb2, did initial testing with mdtraj and basic GMRA functions

TODO:

References

Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467, 2016.

William K Allard, Guangliang Chen, and Mauro Maggioni. Multi-scale geometric methods for data sets ii: Geometric multi-resolution analysis. Applied and Computational Harmonic Analysis, 32(3):435–462, 2012.

Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. Understanding deep neural networks with rectified linear units. arXiv preprint arXiv:1611.01491, 2016.

Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In Proceedings of the 23rd international conference on Machine learning, pages 97–104. ACM, 2006.

David Guy. Wavelets and singular integrals on curves and surfaces. 1991.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. Journal of Machine Learning Research, 12(Oct):2825–2830, 2011.

Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. Computing in Science & Engineering, 13(2):22–30, 2011.

Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. HotCloud, 10:10–10, 2010.