3D Data Processing in Structural Biology 76562 (Spring 2021)
Assignment 4

(due: June 7)

Last update-27/05

In this exercise we will continue our attempt to model nanobodies using Rosetta.
As you saw in Ex3, Rosetta struggled to model CDR3 accurately, especially when the loop is long. This time we will use neural networks in order to improve our performance from the previous exercise.
Neural networks have become a common tool for modeling proteins, and the recent success of AlphaFold 2 (by Google's DeepMind team) showed the amazing potential of using them.

Resources for further reading:

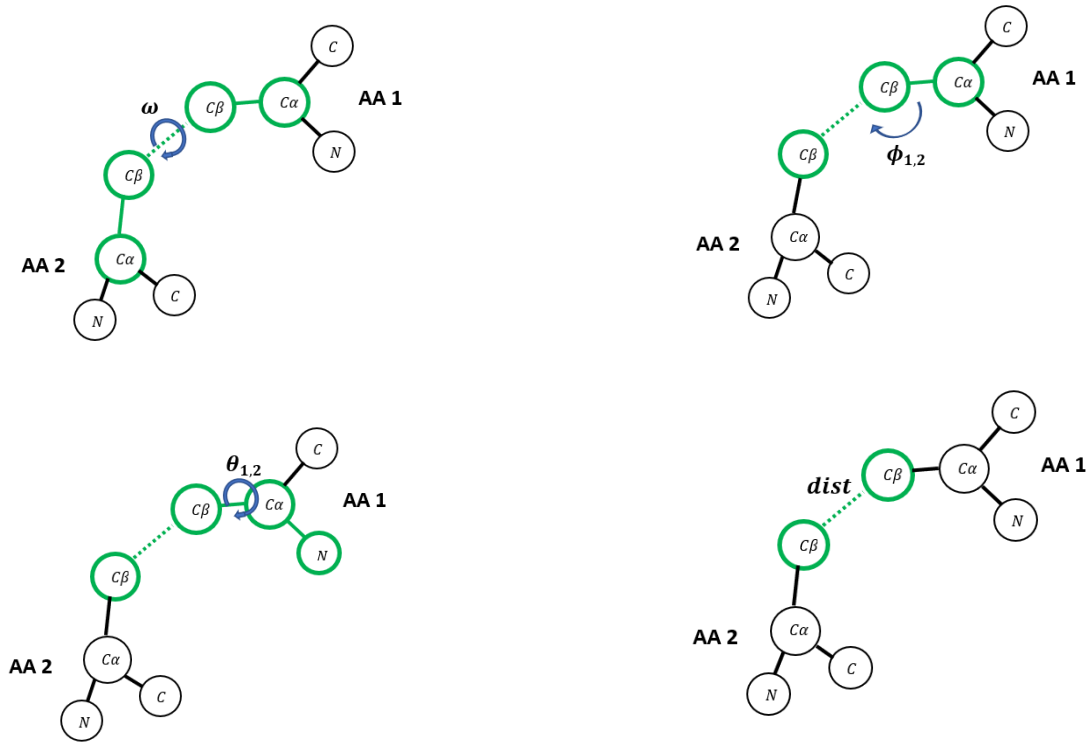
- Keras: the Python deep learning API
- An Intuitive Explanation of Convolutional Neural Networks – the data science blog
- A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way
- CS 230 - Convolutional Neural Networks Cheatsheet
- Constraint File Format

We really recommend using Google Colaboratory when implementing this exercise, especially when training the network. You will be able to use GPU and it will **reduce your training time significantly!**
Google Colaboratory also allows you to save files to your drive (like the input and output data you will generate) and load in any file you want (like when training the network).

Read the short Google Colaboratory guide in the moodle in order to see how to connect to a GPU, save files/trained models, load files and import ipynb files from your drive.

Part 1 - processing the data

In this part you will process all the data into training input and output for the Neural network.
The input of the neural network will be the one-hot encoding of the CDR3 sequence.
The output will be pairwise distances and angles between different atoms in the CDR3 residues.
We will output 3 angles: omega, theta (dihedral angles- between 4 atomes) and phi (planar angle- between 3 atoms). The angles will be represented by their cos and sin values. The reason we are doing this is to overcome the cyclic property of angles, so we can use MSE (mean-squared error) loss on them. We can later on retrieve the angles by using the arctan2 function.
We will also need to divide the distances by a factor of 10, so all the outputs will have the same magnitude.
The distance and angles are defined by:



Note- the distance and Omega values are symmetrical, while theta and phi are not.

Note - the Glycine amino acid lacks a CB atom, so the angles involving Gly residues are ignored and the distances are calculated using the CA atoms.

The function that are provided for you are:

1. <u>find_cdr3(sequence)</u> - receives a sequence of a nanobody (String) and returns a list [start, end] where start is the start index of the CDR3 in the sequence and end is the end index of the CDR3 in the sequence.

2. <u>get_seq_aa(chain)</u> - receives a BioPython chain object and returns a tuple of its aa sequence (String) and a list of all the aa residue objects of that sequence.

3. <u>get_dist(residues)</u> - receives a list of residue objects of size N, and returns a numpy array of size N*N*1 with the pairwise CB distances of the residues.

4. <u>get_theta(residues)</u> - receives a list of residue objects of size N, and returns a numpy array of size NxNx2 with the pairwise theta dihedral angles of the residues (represented by their sin,cos).

5. get_phi(residues) - receives a list of residue objects of size N, and returns a numpy array of size NxNx2 with the pairwise phi planar angles of the residues (represented by their sin,cos).

6. add_padding(matrix) - receives a numpy array of size NxNxM (where N<32) and returns the matrix after padding it with zeros. The returned matrix has a size of 32x32xM.

7. remove_padding(matrix, cdr_length) - receives a numpy array of size 32x32xM that had an original size of cdr_length*cdr_length*M and was padded using add_padding(). It returns the matrix after removing the padding.

8. generate_label(pdb) - this function receives a path to a nanobody PDB file, and returns its CDR3 pairwise distances and pairwise angles (omega, theta, phi) . The function has 4 return values in this order:

   Distance matrix- a numpy array of size 32x32x1
   Omega matrix- a numpy array of size 32x32x2 ([sin, cos]).
   Theta matrix- a numpy array of size 32x32x2 ([sin, cos]).
   Phi matrix- a numpy array of size 32x32x2 ([sin, cos]).

   This function calls the get_dist, get_theta, get_omega and get_phi functions. Because most CDR3 sequences are shorter than 32 aa, The arrays are then padded with zeros, by using the function add_padding(matrix).

9. generate_input(pdb_file) - this function receives a path to a nanobody PDB file, and returns its CDR3 sequence represented in a one-hot encoding matrix. The returned matrix is a numpy array of size 32x21 (each row represents an aa in the sequence).

   Again, because most CDR3 sequences are shorter than 32 aa, it pads the matrix by setting the special 21 column to 1.

Please Implement the following functions in the 'utils.ipynb' file (you have empty templates for the required functions):

a) get_omega(residues) - receives a list of residue objects of size N, and returns a numpy array of size NxNx2 with the pairwise omega dihedral angles of the residues (represented by their sin,cos).

Note - you can implement get_omega() the same way get_theta(), get_phi() and get_dist() are implemented.

Now, use the generate_input() and generate_label() functions in order to generate the input and labels for the network. Use all the nanobodies structures in the Ex4_data directory (2141 structures).

You should generate 5 matrices ;

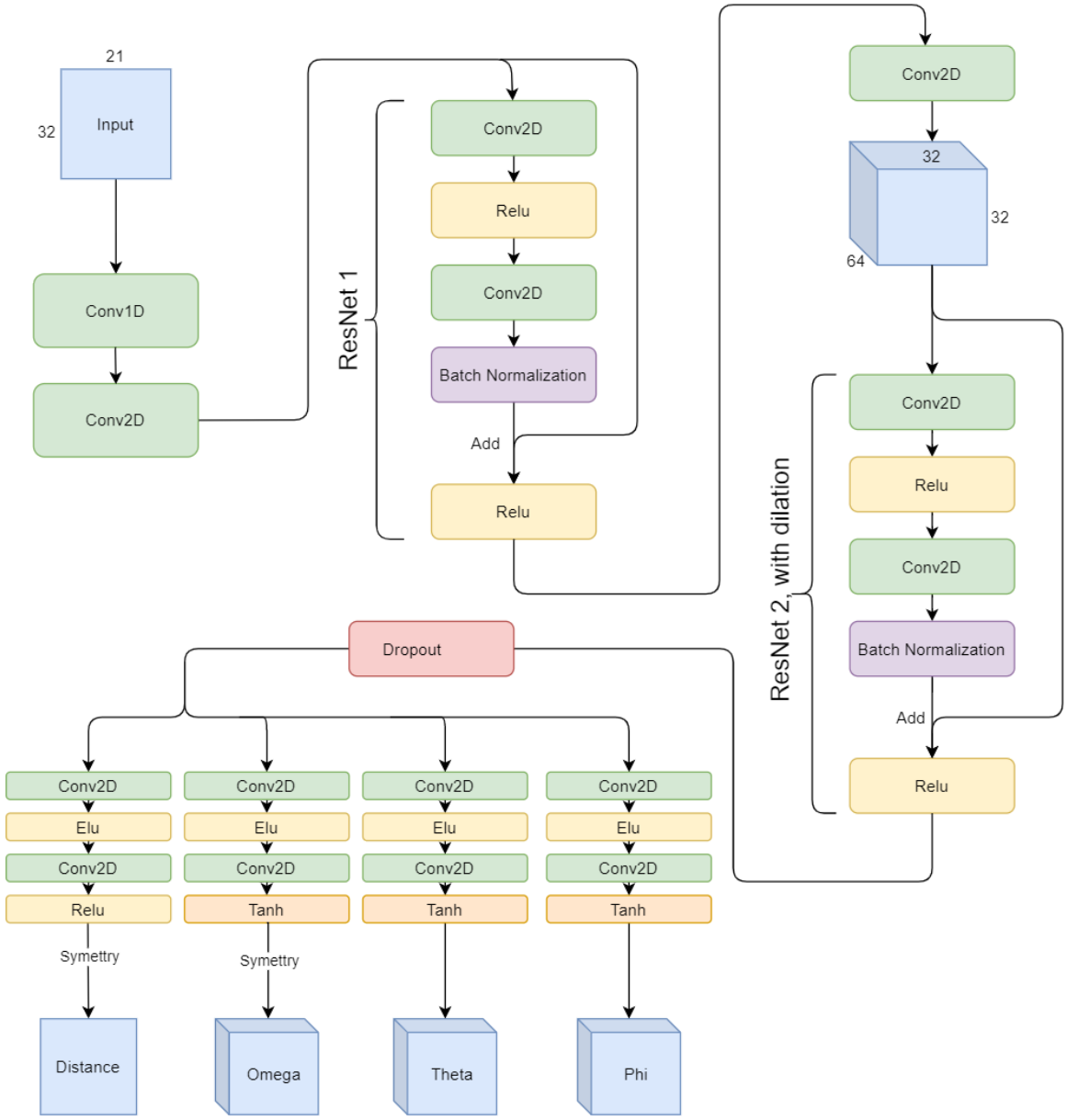
- one for each label, each with a size of 2141x32x32x2 / 264x32x32x1
- one for the one-hot encoding matrix with a size of 2141x32x21

You can save them using the pickle library, we will use this data in the next part.

(this part is implemented for you as well, **you might need to change the 'data_path' and 'save_path' variables according to your folder path in the drive**)

Part 2 - building the neural network

In this section we will build the neural net



(don't panic- most of the networks is implemented for you)

The network consists of the following layers:

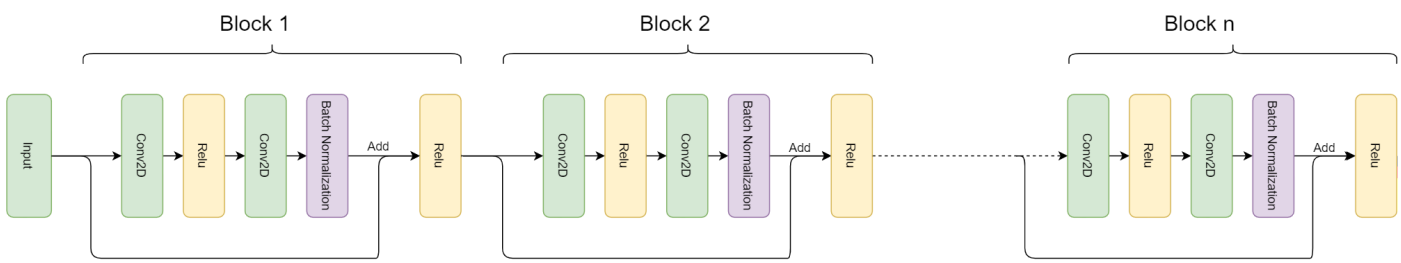
1. 1D convolution - with 32 kernels and a kernel size of 11. This layer ‘decodes’ our one-hot encoding representation.

2. 2D convolution - with 32 kernels and a kernel size of (11,11). This layer outputs a 32x32x32 matrix, so we can put it through the first ResNet.

3. First ResNet - this layer consists of 3 ResNet blocks, each consists of:

   Input layer -> 2D convolutional layer with 32 kernels, kernel size of (11,11) -> Relu activation layer -> 2D convolutional layer with 32 kernels, kernel size of (11,11) -> Batch normalization layer -> adding the input layer to the last layer -> Relu activation layer.

   Then the last Relu layer becomes the input layer for the next ResNet block.



4. 2D convolution - with 64 kernels and a **kernel size of your choice** (same size as the second ResNet). This layer outputs a 32x32x64 matrix, so we can put it through the second ResNet.

5. Second ResNet - this layer consists of n*m dilated ResNet blocks (n - number of repetitions to make, m- length of the dilation list).
   For example, let's say we chose to use n=3 and dilation=[1,2,4] (m=3). So we will have 9 ResNet blocks with [1,2,4,1,2,4,1,2,4] dilations in this order.
   each block consists of:

   Input layer -> dilated 2D convolutional layer with 64 kernels -> Relu activation layer -> dilated 2D convolutional layer with 64 kernels -> Batch normalization layer -> adding the input layer to the last layer -> Relu activation layer.

   Again, the last Relu layer becomes the input layer for the next ResNet block.
   You need to **choose the kernel size, dilations, repetitions (n)**.

6. Dropout layer- this layer is used for regularization. In each step of training it randomly sets some of the input weights to zero (**with a percentage of your choice**).
7. Output layers - the network then splits into 4 output layers:

    - Distance : this layer consists of:
      2D convolutional layer with 4 kernels -> Elu activation layer -> 2D convolutional layer with 1 kernel -> Relu activation layer (to enforce positive output) -> combining with its transpose (to enforce symmetrical output).

    - Phi : this layer consists of:
      2D convolutional layer with 4 kernels -> Elu activation layer -> 2D convolutional layer with 2 kernels -> Tanh activation layer (to enforce -1<=output<=1).

    - Omega: this layer consists of:
      2D convolutional layer with 4 kernels -> Elu activation layer -> 2D convolutional layer with 2 kernels -> Tanh activation layer (to enforce -1<=output<=1) -> combining with its transpose (to enforce symmetrical output).

    - Theta: same as Phi.

    All of them have a kernel size of (5,5).

All the following sections should be implemented in '**net.ipynb**':

a) Your job is to implement the **Second ResNet**, **Phi layer and Omega layer** in the functions **resnet_2(), phi_layer() and omega_layer()**.
   Use the functions we implemented for you (resnet_1(), distance_layer(), theta_layer()) as inspiration.
   Use padding="same" for all convolutional layers. For **resnet_2()** set the dilation using the dilation_rate parameter.

   You also need to choose some of the parameters for the network, all are specified at the top of the python file.

b) After you built the network, Compile the model with Adam optimizer and a learning rate of 0.001.

c) Fit the model with the data you generated in the previous part with batch size of 32 **and number of epochs of your choice**.
   Make sure to set the y value to [distance, omega, theta, phi] in this order.

d) Save your final model after training.

Note - you can split your data into training, validation and testing sets as you like. You can use train_test_split() from sklearn.

Tip- You are provided with the function 'plot_val_train_loss()' that receives a history object (the output of fit function) and plots the training loss and validation loss as a function of the epoch. Use it to see when you are overfitting.

Q1 : list all the parameters you chose, how many trainable parameters does your network have? (use Model.summary()).

Q2 : use your trained model in order to predict the distances and angles of nanobody **6dlb** from the previous exercise (use ref.pdb). Use the function '**plot_distance_heatmap()**' provided for you in order to plot the true distance matrix compared to the one you predicted. Add it to your submission and describe it.

Bonus:

We will test your models on a test set that you didn't have during training. The team with the best performance on our test set will receive 10 bonus points for the exercise. The second and third team will receive 3, 5 points respectively.

(You are allowed to change all the parameters specified in the top of net.ipynb)

Part 3 - writing a constraints file

After we predict the distances and angles for a new nanobody using our neural network, we need to feed them to the Rosetta modeling tool. We will convert our predictions into constraints on the modeled structure.
Rosetta has an option to receive a constraints file, the constraint terms will be added to the score function (energy function). The better the model meets the constraints, the lower the energy function will be. Rosetta does this by applying a 'penalty' function on the constraint we provided and adding it to the energy function. For distances we will use the harmonic function:

$$f(x) = \left(\frac{x - DistanceConst}{std}\right)^2$$

And for the angles we will use the circular harmonic function:

$$f(x) = \left(\frac{NearestAngleRadians(x, AngleConst) - AngleConst}{std}\right)^2$$

You can read all the needed information on Rosetta constraints here.

You are provided with a python program '**const.ipynb**' (and a **const.py** version that can run on your local computer if you prefer) that receives a PDB file, a trained neural network path and a name (path) for the output constraints file to generate. The program predicts the distances and angles using the provided network and it then converts them into Rosetta constraints format with the above functions. It saves the constraints into the output file provided**.**

a) All you have to do in this part is determine the standard-deviation of each constraint. Fill your values into the four variables at the top of the '**const.py'/const.ipynb** file (DIST_STD, OMEGA_STD, THETA_STD, PHI_STD).

   Tip- if you used a validation set in the training part, you might want to use your validation loss in order to choose the STD for each constraint.
   We want a bigger STD for values we predict less accurately.

b) Use the program in order to generate a constraint file for PDB **6dlb**, this is the same nanobody you modeled in the previous exercise. Use **'model-0.relaxed.pdb' (this is the model after the homology modeling and before the H3 loop modeling from Ex3)** and your trained model as parameters.

Part 4 - modeling using rosetta with constraints

Now we can finally model **6dlb** using the constraints we generated.

a) Run the following command in your terminal (after you follow the same instructions as in exercise 3 - setting the environment variables and making the H3_neural_modeling folder):

   antibody_H3.linuxgccrelease @abH3.flags -s model-0.relaxed.pdb -nstruct 1 -out:file:scorefile H3_neural_modeling_scores.fasc -out:path:pdb H3_neural_modeling -constraints:cst_file constraints_file -constraints:cst_weight 1.0 > h3_neural_modeling-0.log

Use here '***model-0.relaxed.pdb***' (this is the model after the homology modeling and before the H3 loop modeling from Ex3) and the constraints file you generated in the previous part as the ***constraints_file.***

Note- if you are running it on the 'hm' cluster using the example script file in the moodel, change the maximum running time parameter to 6:0:0 (--time=6:0:0)

Q3: what is the total score of your model? What is the value of the angle (phi) and dihedral (omega, theta) constraints in the energy function? Add the score file in your submission.

Q4: align the model you created to the reference structure (ref.pdb) in pymol/chimeraX. Evaluate the model compared to the model you created in exercise 3.
Did you manage to model CDR3 better this time?

Q5: What is the RMSD of your model?

Tip: To find H1,H2,H3 you can use the following link in order to renumber the models according to the Rosetta numbering. Then, you can color them using the positions you found in exercise 3.

Submission

Yout submission should include a directory named Ex4.tar with the following files:

1. **answers.pdf -** answers for questions 1-5.
2. **trained_model -** your trained model.
3. **utils.py -** with the requested functions implemented (py file).
4. **net.py** - with the requested functions implemented (py file).
5. **H3_neural_modeling_scores.fasc -** your score results for Rosetta H3 modeling.
6. **H3_neural_modeling -** the directory containing the model you created.