Improving CDR-H3 Modelling in Antibodies

Benjamin James Blundell
Supervised by Andrew C.R. Martin

Department of Biological Sciences
Birkbeck, University of London
Malet Street, London WC1E 7HX
United Kingdom

August 17, 2018

Contents

Abstract

Antibodies - key molecules of the immune system - are increasingly used as therapeutic drugs. The variable domain of the antibody is responsible for binding to an antigen and contains polypeptide loops, each referred to as a *'Complementarity Determining Region (CDR)'*. Five of these loops can be modelled with acceptable accuracy but so far, the sixth (CDR-H3) remains more difficult for all but the shortest loops.

Various methods have been attempted to model this CDR, and while accuracy has improved, more improvement is still needed. One such method is to apply machine learning in an attempt to discern which features of a sequence lead to what final conformation; such an approach was tested in 1995. Since that time, the number of structures available for learning has increased dramatically, as have methods and technologies available for building and training neural networks.

This work revisits the neural network approach to modelling and scoring CDR-H3 in light of these advances. We conclude that several common neural network architectures cannot accurately model CDR-H3 from sequence alone, but show reasonable performance in selecting a good candidate from a large set.

Acknowledgements

I like to acknowledge the following for their support and generosity which made this work possible: Andrew Martin, Katie Eagleton, Lukasz Zalewski, James Phillips & the London Biohackspace community, the Cirrus team at the University of Edinburgh, the Cambridge Service for Data Driven Discovery, the UK Research Software Engineering community, Hannah Dee, Roger Boyle and Nicola Ford.

Acronyms

AMA Antibody Modelling Assessment.

ANN Artificial Neural Network.

$C\alpha$ Carbon Alpha.

CDR Complementarity Determining Region.

CNN Convolutional Neural Network.

GPU Graphics Processing Unit.

LSTM Long Term Short Term.

NN Neural Network.

PDB Protein DataBank.

RMSD Root Mean Squared Deviation.

RNN Recurrent Neural Network.

SGD Stochastic Gradient Descent.

Chapter 1

Introduction

1.1 Introduction

1.2 Antibodies

Antibodies are a major component of the adaptive immune systems in higher vertebrates. In recent times, antibodies have been recognised for their use as drugs. Antibodies have high specificity - they can target a particular antigen to the exclusion of other molecules. They also have high affinity - a strong interaction with the antigen in question. One of the challenges in using antibodies in drug design is to ensure that artificial antibodies are not seen as *foreign* and therefore targeted by the host immune system. Antibodies can be used in various ways to benefit the host: helping to direct drugs to the correct location, activating the immune system to target either host cells or invading cells, and functioning like a traditional drug (for example, binding to an active site in order to block it).

1.2.1 Structure

Antibodies have a characteristic 'Y' shaped structure. These structures can be combined into various different *isotypes* (or classes). In humans and other placental mammals 5 types exist: IgA, IgD, IgE, IgG and IgM. IgA forms a dimer, and IgM a pentamer (the 'Y' shape can still be seen). Each has its own class of heavy chain - $\alpha, \delta, \epsilon, \gamma$ and μ respectively. Some other species have different kinds of antibodies. For example, camelids and sharks have antibodies consisting only of heavy chain.

IgG, a monomer, is the most common type in humans. Like these antibodies, IgG is built up of *'light'* and *'heavy'* chains, featuring the characteristic *'immunoglobin fold'*. Two heavy chains form the *'stem'* and *'forks'* of the Y shaped molecule, with the two light chains attaching to the 'forks'. Light chains are approximately 220 amino acids in length (two domains of 110 amino acids in length), with the heavy chains being approximately 440 amino acids long (four domains of 110 amino acids each). IgG has 4 human subclasses - IgG1 to IgG4, with heavy chains γ_1 to γ_4.

The stem of the antibody is known as the **Fc** (fragment crystallizable) region,

and is responsible for interfacing with the host immune system in various ways. The forks of the 'Y' shape are referred to as the **Fab** (fragment antigen binding) region; the top-half of these regions are the **Fv** (Fragment variable) regions (see figure 1.1)

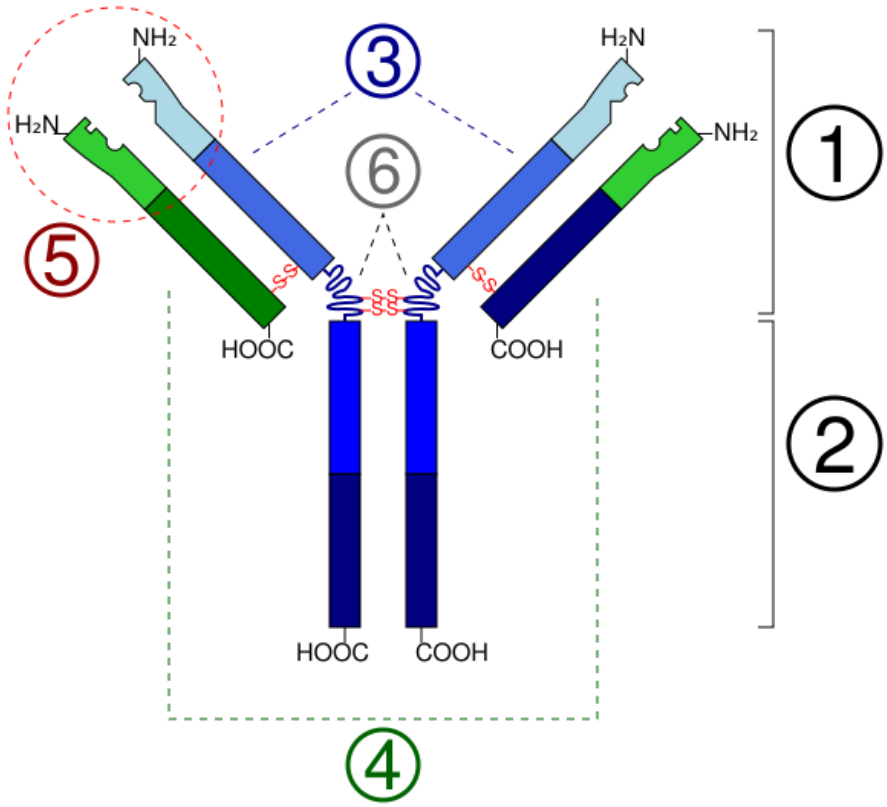


Figure 1.1: Schematic diagram of the basic unit of immunoglobulin. The heavy chain (From the N Terminus - VH, CH1, hinge, CH2 and CH3 regions). The light chain (consisting of VL and CL regions: from N-terminus). The -S-S- labels are disulphide bonds. The notched areas form the antigen binding site. 1) Fab Region 2) Fc Region 3) Heavy chain in blue 4) Light chain in green 5) Fv region. `https://commons.wikimedia.org/wiki/File:Immunoglobulin_basic_unit.svg`.

1.2.2 Complementarity determining regions

The **Fv** region contains the antigen binding site. Within this region are the variable loops connecting to a framework of β strands. These polypeptide loops are responsible for the high specificity and variability of antibodies. They are referred to as *'complementarity determining regions'* (CDRs). The β strands form part of what is often referred to as the *framework region*. This region forms

the scaffold for the CDRs, helping to maintain the overall structure of the **Fv** region.

Take-off regions are mentioned by various authors [47][60][68]. Whilst no formal definition appears to exist, they provide a useful set of constraints for modelling the CDRs. Somewhat analogous to the term *anchor points*, the phrase *take-off* point implies a direction for the subsequent loop to follow.

There are 6 such CDRs at the end of each antibody fork - three on the light chain and three on the heavy chain - numbered 1 to 3 with the prefix H or L. For example, the second heavy loop is referred to as CDR-H2, whereas the first light CDR is CDR-L1. The conformation of these loops is responsible for which epitope a particular antibody will attach to. Figure 1.2 shows the loops as they appear in the model *1hzh* taken from the Protein DataBank (PDB).

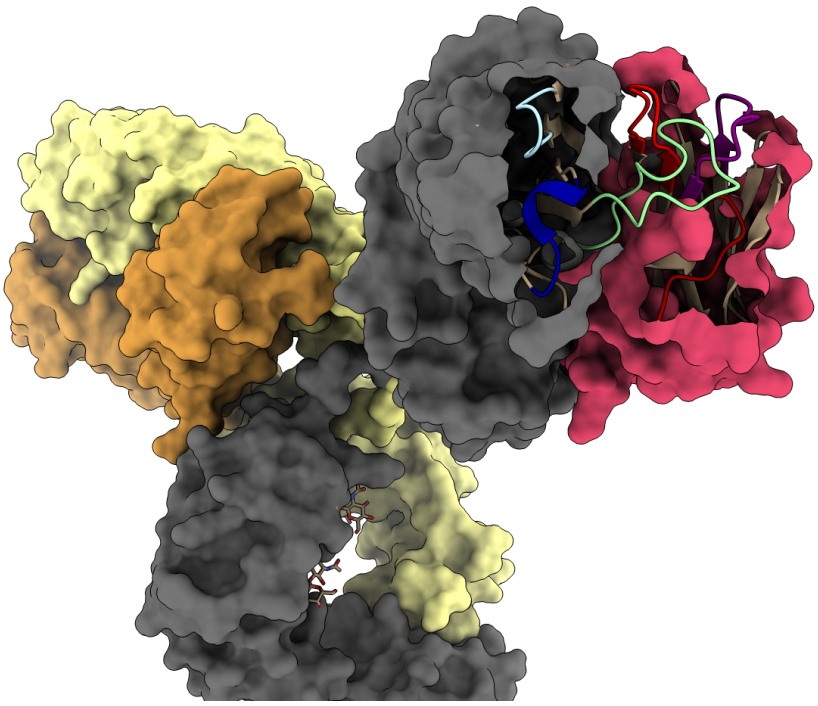


Figure 1.2: The model 1hzh. The CDR loops are shown as ribbons of different colours, with CDR-H3 shown in the foreground, in green. The grey and yellow surfaces are the heavy chains. The orange and red surfaces, the light chains.

CDR definitions

Which amino acids form the CDRs exactly? Different definitions exist.

The Kabat scheme [80] looks for short stretches in the sequence that show high variability. The Chothia scheme [11] takes a closer look at the actual structure of the hyper variable area, considering the hyper-variable loops to be these stretches of amino acids outside the *"framework β sheet"*.

Oxford Molecular's AbM program uses a definition based on a compromise between Kabat and Chothia. It differs in its definitions of H1 (taking the lower

Loop	Kabat	AbM	Chothia	Contact
L1	L24 - L34	L24 - L34	L24 - L34	L30 - L36
L2	L50 - L56	L50 - L56	L50 - L56	L46 - L55
L3	L89 - L97	L89 - L97	L89 - L97	L89 - L96
H1 (Kabat Numbering)	H31 - H35B	H26 - H35B	H26 - H32..34	H30 - H35B
H1 (Chothia Numbering)	H31 - H35	H26 - H35	H26 - H32	H30 - H35
H2	H50 - H65	H50 - H58	H52 - H56	H47 - H58
H3	H95 - H102	H95 - H102	H95 - H102	H93 - H101

Table 1.1: The definitions of CDRs according to the various schemes, taken from `http://www.bioinf.org.uk/abs/index.html`

bound from Chothia and the upper bound from Kabat) and H2 (the lower bound of Kabat and an upper bound between the two schemes).

The Contact scheme [49] [41] takes a different approach. It is based on the residues that are observed to make contact with an antigen, as observed from analysing crystal structures.

CDR numbering schemes

Numbering reflects which residues are considered to be *indels* (an insertion or deletion). Consistent numbering is incredibly useful as it allows one to refer to certain residues across different structures.

Unfortunately, several methods of numbering the CDR regions exist. One of the earliest and most widely adopted is the Kabat scheme [34]. Residues are numbered as normal, with additional indels given an alphabetical suffix.

The Kabat scheme has some problems. Firstly, the positions at which insertions occur in CDR-L1 and CDR-H1 do not match the structural insert positions. Further analysis of the immunoglobin structures showed that residues outside of these defined by Kabat showed variation [3]. The Chothia scheme corrects for this.

The Chothia numbering scheme builds on the Kabat scheme but places the insertions for CDR-L1 and CDR-H1 in positions that are consistent structurally [3]. This refinement does not extend to these areas outside of the CDR - the *framework regions*.

Abhinandan and Martin [1], *"have analyzed the numbered annotations in the widely used Kabat database and found that approximately 10% of entries contain errors or inconsistencies."* The Kabat numbering scheme was found to be applied incorrectly in certain cases. One example: Kabat places an insertion in CDR-H2 at position 82 in HFR3, whereas Abhinandan and Martin found 74 sequences within the Kabat database that should have had this insertion placed in the CDR-H2, at residue 52.

The new scheme proposed by Abhinandan and Martin [1] (called the Martin or Chothia-enhanced) is based on the Chothia scheme but provides *"corrections to the positions of the insertions and deletions in the framework regions"*. A program called *AbNum* was used to perform this re-numbering and analysis

		Canonical Structure			
Residue	Angle	1	2	3	4
26	ϕ	-71	-80	-87	-5
	ψ	-16	-20	-22	-15
27	ϕ	-158	-132	-134	-136
	ψ	169	166	166	157
28	ϕ	-59	-72	-66	-66
	ψ	133	128	139	140
29	ϕ	-113	-112	-95	-103
	ψ	151	6	23	9
30	ϕ	-76	56	-77	-86
	ψ	-38	-120	119	130
30a	ϕ		-125	-75	-105
	ψ		29	156	107
30b	ϕ			-54	-56
	ψ			-34	-34
30c	ϕ			-72	-59
	ψ			-4	-54
30d	ϕ				1
	ψ				8
30e	ϕ			97	47
	ψ			-18	45
30f	ϕ			-81	-132
	ψ			156	153
31	ϕ			-117	-102
	ψ			107	128
32	ϕ	-157	-88	-89	-95
	ψ	160	73	77	74

Table 1.2: An example from the canonical rules of Chothia [3] for the Vκ Light Chain CDR-L1 structures. Angles given are averages in degrees.

and can be found on the web[1].

Complementarity determining region classification

Most of the CDRs fall into one of a limited set of conformations and can therefore be predicted from their sequence of amino acids and modelled. The exception to this rule is CDR-H3, which does not readily adopt a particular conformation.

An example canonical rule from the Chothia set [3] for CDR-L1 is given in table 1.2. For each residue and canonical structure class, an average phi Φ and psi Ψ angle is given. If the loop identified has backbone torsions that fit these angles, it belongs in that class.

Since the Chothia rules were published, there have been several improvements made. MacCallum *et al* [41] define four distinct classes of protein sur-

[1]made available at `http://www.bioinf.org.uk/abs/abnum/`.

faces, applied to antibodies: concave and moderately concave, ridged and planar. North *et al* [54] consider a more recent database of structures and use a conformational clustering technique to derive their classifications. Their dataset is 15 fold larger than the set used by Chothia. North estimates 85% of non-CDR-H3 CDR structures can be predicted based on the source of the gene and its sequence. Chothia defines 12 canonical classes [11], Al-Lazikani *et al* recognised 25 canonical classes [3] whereas North *et al* recognise 72 [54].

CDR-H3

One of the reasons for CDR-H3's variability in structure is the total number of amino acids varies considerably. Wu *et al* [80] show the length distribution of CDR-H3 between 2 and 19 amino acids in Humans. The abYsis database and antibody resource [71] shows the majority of the lengths are between 9 and 14 amino acids long but lengths can be as short as 1 or as long as 28; longer CDRs exist in other species. Figure 1.3 illustrates the CDR-H3 length distribution in three different species.

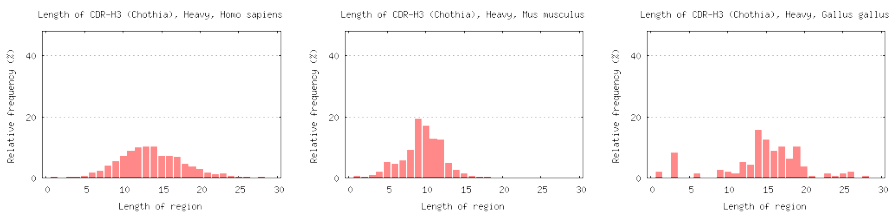


Figure 1.3: Length Distribution for CDR-H3 (Chothia definition) in humans (left), mouse (middle) and chicken (right) (images taken from `http://abysis.org/`).

CDR-H3 shows the greatest structural diversity (figure 1.4) of the CDRs. It also appears in the middle of the binding site, and makes the most contacts with the antigen on average [41].

The variability in the number of amino acids comprising CDR-H3 makes such classifications more difficult. Shirai *et al* [66] attempt to classify part of the CDR-H3 loop using a set of complex rules. They note that the base of the loop, particularly the first residue and the last three residues (the anchor points of the loop) conform to either a *'bulged'* or *'extended'* shape (bulged is also referred to as *'kinked'* in several places — see figure 1.5). Morea *et al* [52] place the torso regions as being the 4 residues at the N terminus, and 6 residues at the C terminus.

However, since the number of experimental structures has increased, the rules give by Shirai *et al* and Morea *et al* do not appear to be accurate. According to North *et al* [54] a bulged or non-bulged conformation is not dependent on the presence or absence of *'key residues'* and the torso regions show lower variability.

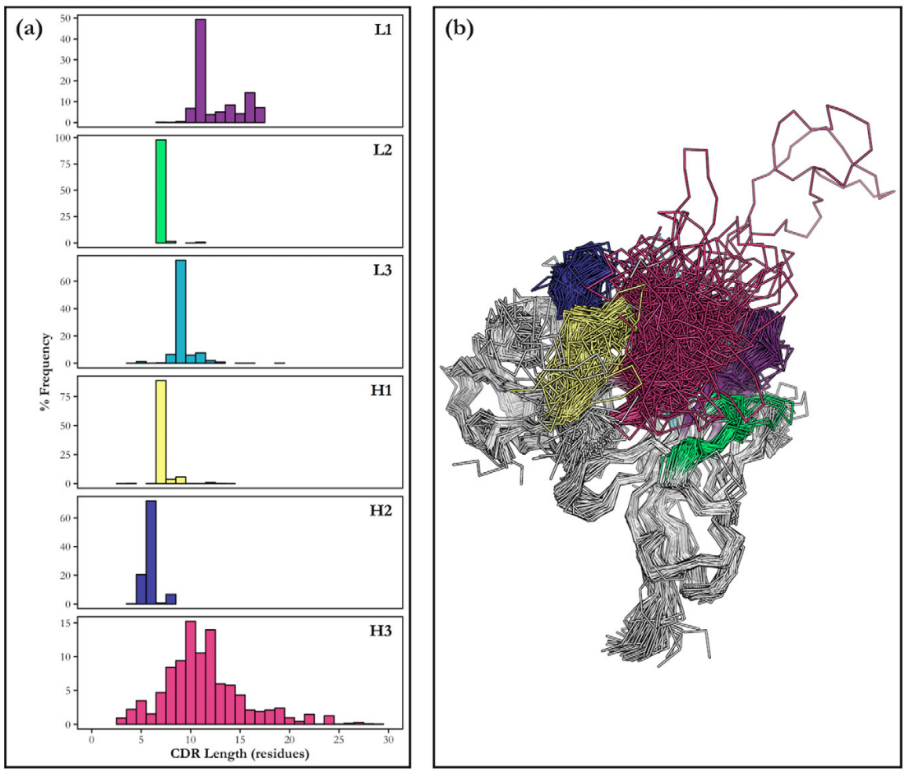


Figure 1.4: (a) The different distributions of the lengths of the each CDR. CDR-H3 is in pink at the bottom of the figure, and shows a greater variance than the other loops. (b) The differences in the structure of the CDRs. CDR-H3 is dark pink. This figure is derived from the SAbDab database. Taken from the paper by Marks & Deane [46].

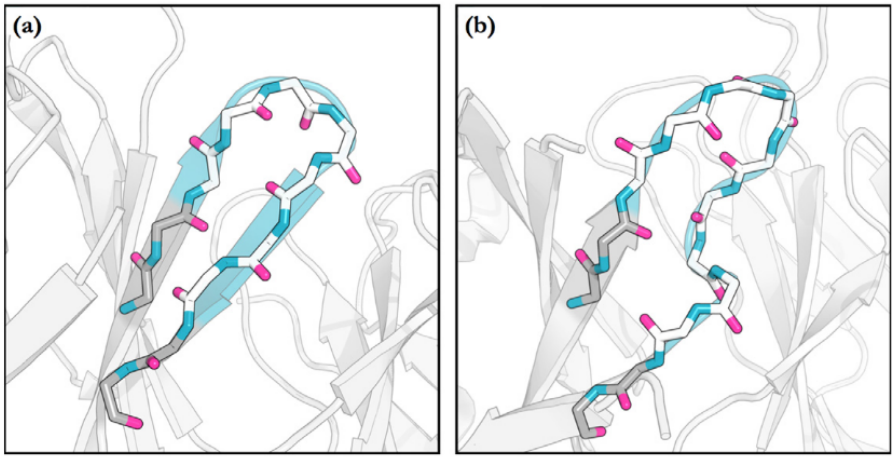


Figure 1.5: (a) An example of an *'extended'* CDR-H3 loop and (b) a *'bulged'* CDR-H3 loop. Taken from the paper by Marks & Deane [46]

Regep *et al* [61] have suggested that CDR-H3 is quite different to other polypeptide loops; 75% of CDR-H3 loops do not have a *'sub-Angstrom structural neighbour'* outside of other antibodies. The authors go on to note that despite a computational study suggesting that CDR-H3 is somewhat flexible due to the lack of stabilising bonds, the CDR-H3 loop is less flexible than other loops in the PDB. On average, CDR-H3 loops contain 10 times as many unique conformations as other loops; grouping CDR-H3 into categories based on structure is much more difficult.

Finally, the authors state that more than 1000, 4-residue fragments adopt conformations not seen in any other structures, from the 64,830 4-residue fragments they extracted from their dataset. More than 30% of the CDR-H3 loops the authors tested contained at least one such structure. These structures are found in the tip of the CDR-H3 loop and show *'a high propensity for Tyrosine and Glycine in unfavourable conformations'*. The tip is defined as the residue that contains the Carbon Alpha ($C\alpha$) furthest away from the anchor points of the loop. This suggests that modelling methods involving restricting torsions to the Ramachandran map may introduce errors. The uniqueness of the CDR-H3 loops may place some restrictions on template based modelling; the loop to be modelled may have a unique structure not present in the template database.

1.2.3 Datasets

The number of structures available for analysis has steadily increased. According to the PDB, the rate that entries are added has also increased year-on-year (with a small number of exceptions). As of 2017, the number of entries in the PDB exceeds 125,000.

The Summary of Antibody Crystal Structures (*SACS*) is a "self-maintaining web-site containing summary information on antibody structures in the PDB"[4]. A summary of the structures found by *SACS* to be antibodies can be downloaded as an XML file[2].

The structural antibody database(SabDab) [14] is an online database, specifically targeting antibody structures. As of 18th July, 2017, it contains 2779 structures [3]. 2401 of these contain at least one paired light and heavy chain. This results in 5257 F_v regions and a search for complete CDR-H3 structures using the Chothia definition returns 2629 results.

AbDb[4]is a collection of structures processed from the PDB. The complete set contains 1184 non-redundant antibodies.

abYsis [72][5] combines data from various sources, including Kabat and the PDB, along with various discovery tools and utilities. As of 19th July 2017, there are 131241 non-identical sequences; a search for structures derived from the PDB returns 4315 results.

The *International Immunogenetics Information System* is an online resource[6] containing several databases relating to antibody research. The IMGT/3Dstructure-DB contains 4766 entries, extracted from the PDB [33].

[2]Available online at `http://www.bioinf.org.uk/abs/sacs/`

[3]according to `http://opig.stats.ox.ac.uk/webapps/sabdab-sabpred/Welcome.php`

[4]`http://www.bioinf.org.uk/abs/abdb/`

[5]`http://www.abysis.org/`

[6]`http://www.imgt.org/`

1.3 Modelling

Given a sequence, generating the structure of a protein can be challenging. However it is of considerable importance to know this structure as it defines biological function. Structure is therefore, more conserved than sequence.

At the time of writing, the number of sequences that exist inside UniProtKB/SwissProt is of the order of 550,000. These are manually reviewed and annotated. However UniProtKB/TrEMBL, which is automatically annotated, holds 87 million sequences. The PDB contains around 132,000 experimental entries; roughly 0.2% of UniProtKB/TrEMBL and 24% of UniProtKB/SwissProt. It is expected these percentages will go down as sequencing becomes cheaper. Until methods such as X-Ray crystallography and NMR are as cheap and widespread as sequencing, modelling the structure from the sequence alone will be important.

1.3.1 Current approaches to tertiary structure prediction

Several approaches exist for generating three-dimensional, or tertiary structure, from a sequence of amino acids. There are broadly two categories: *Ab initio* and *knowledge-based*. More specifically:

1. *Ab initio* approaches that make use of:
   (a) Energy minimisation and conformational search.
   (b) Molecular dynamics simulations.
2. Knowledge based approaches including
   (a) Comparative modelling.
   (b) Threading (as a precursor to the above).
3. Combined approaches.
   - using a knowledge based approach combined with a form of machine learning and / or *Ab Initio* method.

Molecular dynamics (MD) simulations model individual atoms as spheres following Newton's laws of motion. Originally developed within the field of theoretical physics, it has since found use in materials science and biology. One of the more famous examples is the Folding@Home project[7]. Such simulations require an enormous amount of computing power, even for relatively small molecules. Anton is a special purpose machine designed to perform MD simulation on biological macro-molecules such as proteins. In their 2008 paper, Shaw *et al,* [65] claim that their 512 node Anton machine should be able to simulate a 1 millisecond, 23,558 atom protein-cancer-drug system, in one month .

Energy minimisation (EM) approaches seek to find the energy minimum of a force-field — often the same force field equations used in MD. An energy level is calculated for a particular conformation of atoms and a search algorithm is used to find which conformation yields the lowest free energy — *exploring the free energy landscape*. Techniques such as gradient descent and simulated annealing

[7]More information is available at `http://folding.stanford.edu/about/`

are often used to find a route through the incredibly large number of valid conformations. The major problem is avoiding local minima and finding which *direction* to head in. One recent example by Heo *et al*[26] calculates the energy of of a number of *decoy loops*,selecting the one with the lowest free energy to provide an initial starting point.

Comparative modelling falls into the category of *knowledge based* approaches. An unknown structure may be modelled if a known structure can be shown to have some relation to it. One such method, called *homology modelling* takes into account evolution; if another structure exists that has a common ancestor to the one being modelled, it might well serve the same function and therefore have similar structure. Often, homologs are identified by similarity of sequence.

Comparative modelling doesn't have to rely on homologs. Through *fold recognition* techniques, a suitable template can be found where no evidence of evolution exists.

Most approaches split the model into conserved and non-conserved regions, using the template (most likely a homolog) for the structurally conserved regions (SCR). Once the structural variable regions (SVR) are identified, these are modelled either by hand, using knowledge based approaches (i.e. selecting a likely candidate and grafting it onto the template), *ab initio* approaches (or some combination of the three).

Programs such as COMPOSER [55] and Swiss-Model [7] are examples of comparative modellers that follow such an approach. However, MODELLER [19] takes a slightly different approach. Rather than attempt to divide the model into conserved and non-conserved regions, the entire structure is considered and refined at once. The quality of the results, in both cases, are reliant on how similar the template really is to the model and the quality of the alignment between template and target sequence.

Despite the enormous number of protein sequences, there are only a small number of protein folds[85][8]. As new structures are submitted to the PDB, more and more proteins share pre-existing folds, supporting this idea. This has lead to the approach known as *Fold Recognition*.

One way of describing fold recognition is that it reverses the problem other methods are trying to solve. Rather than taking a sequence and modelling the structure, we consider *how well* a particular sequence fits one of the known structures we have in our library of structures. *How well* is determined by various scores — a set of statistical potentials is one of the most common scores. Rather than calculate the full set of free-energy field equations, with possible approximations (such as using a simple harmonic bond instead of the Scrödinger Equation), we can use scores based on physical properties and attributes of model in question; one such program, GenTHREADER [32] uses a set of functions and the resulting scores (such as pair-wise energy, solvation, etc) as inputs into a neural network. This network derives a score, showing how well a particular sequence fits the fold.

A sequence is described as being *threaded* through a series of library folds — a technique called *threading*. This has been quite successful in modelling proteins. Indeed, relationships between proteins with very different sequences have been

[8]CATH lists 1391 *topologies* at the time of writing - `http://www.cathdb.info/wiki/doku/?id=release_notes`

uncovered using this method [55] — detecting relationships between proteins that have $< 30\%$ sequence identity Certain classes of proteins have proven to be more difficult to match using threading (particularly these with many β structures). The size of the protein and the number of secondary structures it contains has a bearing on the overall accuracy [55].

Protein modelling techniques compete in *The Critical Assessment of Protein Structure Prediction (CASP)*[9] — a world-wide experiment assessing the state of protein structure prediction, taking place every two years since 1994 [53]. CASP is roughly divided into *Template Based Modelling (TBM)* (i.e. knowledge based) and *Template Free Modelling (FM)* (i.e. *ab initio*) categories. Results are available on the CASP website, with critical assessments published in scientific journals the following year.

At the time of writing, assessment for CASP11 (2014) is available, with raw results available for CASP12 (2016). Some of the methods in CASP12 TBM techniques managed to place 70% of a protein's residues to within 1Å accuracy, for the $C\alpha$ atoms. Looking briefly at the data, it is clear that TBM greatly out-performs FM[10]

1.3.2 Loop Modelling

Loops are so-called as the polypeptide segment does not form into any recognisable secondary structure (such as a β sheet or α helix). Loops are often found on the outside of proteins — therefore often in contact with other molecules of interest. Loops are difficult to model — the sequence of amino-acids they are comprised of is quite variable. Some loops are also flexible. Errors in loop modelling are considered to be the dominate problem in comparative modelling.

Insertions and deletions are modelled as loops. Once an area is identified as a loop, the *anchor points* are further identified — usually the two residues either side of the loop. From this point, there are several modelling approaches one can take. Much like protein modelling in general, there are three approaches: *Ab initio*, knowledge based and a combination of both.

Most approaches start by generating a large number of rough solutions. These *'decoys'* are removed via various criteria such as impossible solutions (these with overlapping atoms), poor side-chain packing, torsion angles in unlikely regions of the Ramachandran plot and other metrics. The final selection usually involves scoring a small subset of the possible solutions. Using a set of force field equations (such as AMBER or CHARMM) to rank the solutions by the lowest energy, is one approach. Using the aforementioned statistical potentials as a score is another approach to scoring.

The aforementioned COMPOSER looks for loop structures in homologous proteins that have a similar sequence of amino acids. The user can choose from these loops and effectively *graft* the loop onto the model. Minor adjustments to the ϕ and ψ angles are made, often using simulated annealing, so that the loop fits the anchor points (consisting of 3 residues).

FREAD is an example of a knowledge based approach. Possible solutions for a loop are selected from a restricted database using a score derived from how well the template loop fits the anchor points and the similarity between the

[9] `http://predictioncenter.org/`

[10] `http://www.predictioncenter.org/casp12/results.cgi`

target and template sequences. The disadvantage with FREAD is if a suitable template is not found, FREAD will not return a result. Other knowledge based approaches include SuperLooper [27] and LoopWeaver [30].

Ab initio approaches are used in programs such as MODELLER. Again, minimising the free-energy by searching conformational space using force-fields is the method used here. Fiser *et al* [19] state that *"Loops of 8 residues have a 90% chance to be modelled with useful accuracy"*. Useful accuracy here is a Root Mean Squared Deviation (RMSD) of less than 2Å between the $C\alpha$ atoms that form the back-bone of the loop, for loops up to 8 residues in length. Loops of up to 12 residues can be modelled with reasonable accuracy. MODELLER starts by joining the anchor points with a straight line of residues, then scaling and adjusting this conformation using energy minimisation. Constraints are placed on this energy minimisation based on a set of templates from homologous loops.

CODA combines both the knowledge based method FREAD, and the *ab initio* program PETRA [12]. The authors report an improvement over using either method individually.

The Protein Local Optimization Program (PLOP)[11] is another example of using *ab initio*, force-field based approaches [31]. RMSD accuracy decreases as the loop lengths increase with an average RMSD of 2.71Å for loops 11 residues long. LEAP [40] is another example where an initial set of decoys, created *ab-initio*, are further refined by modelling side-chains and scoring energies for every atom. The authors report an average RMSD of 2.1Å over 325, 12 residue length loops.

One interesting approach to modelling loops uses inverse kinematics; a technique commonly used to orientate robotic arms, or animate computer game characters. Rosetta refers to this method as Kinematic closure (KIC) and uses this in conjunction with Cyclic Coordinate Descent (CCD) [9]. Mandell *et al* [43] have refined this approach in the latest version of Rosetta. The authors of both papers report results on loops of upto 12 residues in length.

1.3.3 Antibody Modelling

The estimation of the total number of possible CDR conformations is huge. On average, a variable domain is 110 amino acids long. A human can produce at least 10^9 different antibodies [58] (more recently, this value has risen to 10^{13} [46]). Chapter 1.2.3 stated that the number of structures currently available is of the order 10^3. Fast and accurate modelling is therefore desired to close this gap.

Algorithms and services exist for modelling antibodies specifically. Specific characteristics of antibodies can be taken advantage of, in order to improve modelling. For example, despite the name, the F_v / variable region has a higher level of conservation than most proteins. than The constant region, as the name suggests, does not change. The exceptions are the CDR Loops, specifically CDR-H3.

One aspect of antibody modelling is how the variable domains of the light and heavy chains are orientated with respect to each other — the V_H-V_L packing angle. Abhinandan *et al* [2] point out that residues in the framework regions, relatively distant from the antigen binding site can have a significant effect on

[11] `http://jacobsonlab.org/plop_manual/plop_overview.htm`

affinity. Their analysis of the angle shows a mean of $-45.6°$ but varies from $-60.8°$ to $-31.0°$.

Perdersen *et al* [58] make the point that correct position of the two domains, the V_H-V_L packing, can have a dramatic effect on the positioning of the CDRs, particularly the take-off points. Weitzner *et al* [78] assessed *RosettaAntibody*, concluding that V_H- V_L orientation is a key factor in the rankings of their generated models.

It has been observed that 5 of the 6 CDRs adopt only a small number of conformations (as seen in figure 1.4); they fall into a canonical class. CDR-H3 is the exception[11]. The implications for modelling are that knowledge based approaches for these loops are quite effective. A small number of loops from other CDRs occasionally do fall outside of the canonical classes [79]; these must be modelled explicitly.

Current approaches

Martin *et al* [48] report RMSD scores of less than 2Å for all loop atoms not in CDR-L3 or CDR-H3, for a model HyHel-5. Similar results are reported for another model, Gloop2. The CDR-H3 loops are 7 and 5 residues long respectively. The algorithm presented relies on a database of backbone conformations for loops longer than 5 residues. For these loops longer than 7 residues, the centre section of the loop is removed and modelled with *CONformation GENerator* (CONGEN) — a program that samples possible conformations for these with the lowest free energy [8]. Side chains are modelled once the backbone is complete. The searches are constrained by Cα-Cα distances within known loops, with a second filter comparing the torsion angles. The final result is chosen from the five models with the lowest free energy.

The algorithm for the above is called *Combined Antibody Modeling Algorithm* (CAMAL) [58] — an early example of combining both knowledge-based and *ab initio* approaches (CAMAL is realised in the program AbM). The rules for canonical CDRs are consulted but the authors note that whilst the majority of loops follow these rules, there are exceptions (such as a *peptide flip* in CDR L1). If no canonical rules exists, the CDR is constructed using either just examples from a database search, a conformational search for just the central section in combination with database examples or a conformational search alone, depending on the number of database examples found.

The reported results for one case study give RMSD scores between 2.43Å and 0.8Å. In this case study, the CDR-H3 RMSD is 0.81Å for an 8 residue loop.

WAM: an improved algorithm for modelling antibodies on the WEB [79]. WAM follows the approach of *AbM* in the main, with some additions. Like AbM, it searches for templates, builds the apex of the loop with CONGEN and compares the computed torsion angles with known angles as part of a screening step. WAM has the following additions

1. Build the canonical loops, minimising the free energy by changing placements of anchor points. Uses the CONGEN program.

2. The results are *energy-screened* using a program called Eureka.

For CDR-H3 specifically, WAM considers whether or not the loop is predicted to be *'kinked'*, changing its database search accordingly. WAM also

considers the side chains when modelling CDR-H3. The accuracy is reported to be between 1.3 and 2.7 Å RMSD for loops of lengths 10 to 12[79]. The authors note the lack of examples of longer loops at the time WAM was developed.

The Prediction of Immunoglobulin Structure (PIGS) models CDRs as follows [44][45]:

1. The target sequence is aligned to known antibodies via the framework regions. Alignment is performed with a Hidden Markov Model. The user can optionally select which framework to use and improve the alignment manually.

2. The closest templates (measured as RMSD distance between $C\alpha$ backbone atoms) for each loop are selected and aligned with the target. PIGS uses the BLOSUM62 [25] score for determining which of the templates are considered for forming the framework for the eventual loop.

3. Canonical loops are identified via sequence and grafted onto the model. These loops that are not canonical are modelled by selecting a template with the same length and highest sequence similarity.

4. The two predicted domains (light and heavy) are packed against each other, taking into account certain residues that are known to be conserved at the interface.

5. Side chains are modelled by either copying from the original templates or predicted with the program SCWRL 4.0[12].

CDR-H3 loops in PIGS follow a special protocol. PIGS considers the presence or absence of the β bulge in its prediction of the CDR-H3 structure closest to the framework region. Templates for CDR-H3 loops are selected from these with the best BLOSUM score. If the loops have different lengths, insertions are made between residues 92 and 104, as this has been found to be the location at which insertions are made when *'antibodies of known structure and different CDR-H3 lengths are superimposed'* [44].

ABGEN [42] models all the CDRs in the same way — templates are selected from a database, based on sequence similarity and length. Residue mismatches are resolved by replacing side-chains, with clashes being removed by iteratively changing the torsion angles. ABGEN achieves accuracies of 1.9Å RMSD for loops up to 10 residues, rising to 3.0Å RMSD for longer loops.

RosettaAntibody, the antibody specific version of the Rosetta program, uses the following methods to model CDR-H3 loops [78]:

1. Use BLAST to find templates. Reject these with a poor MolProbity score. These Chothia-numbered, Kabat-defined structures form a set of constraints for subsequent *de novo* modelling.

2. Perform *de novo* modelling with *Next Generation Kinematic Closure* [68], whilst refining the VL-VH packing (using the Rosetta docking algorithm.

3. Incorporate the *'kink'* or *'bulge'* prediction. Refine the loops by first setting bond lengths and angles to so-called standard values, then gradually

[12] `http://dunbrack.fccc.edu/scwrl4/`

increase the repulsive term in the Lennard-Jones potential, re-apply the all-atom constraints and re-pack the side-chains.

Kinematic Closure (KIC) is a method that uses inverse kinematics from the field of robotics [43]. Having one anchor point in space and one target, the peptide backbone is modelled in the same way as a robotic arm, with fixed lengths and restricted degrees of rotation around the joints. The authors report an improvement in their 12 residue, 25 loop set of 0.8Å RMSD; the original method in Rosetta averaging 2.0Å RMSD.

Stein *et al* [68] improve on how the KIC algorithm traverses the *'rugged energy landscape'*; they test different sampling strategies, concluding that *intensification* and *annealing* combined yield large accuracy gains. This *Next Generation KIC* has been adopted by the latest version of Rosetta.

Messih *et al* [51] propose a new knowledge based approach to modelling CDR-H3 using a machine learning approach called *Random Forest*. This method chooses a template from a set of existing CDR-H3 loops. The chosen template is fed to the MODELLER program to model the CDR-H3 loop, whilst the remaining sections of the antibody are built with PIGS. The authors report significant improvement over RosettaAntibody with '75% of cases achieving similar or better accuracy'.

The structure-based antibody prediction server (SAbPred) can automatically model the F_v regions using a template selected from SAbDab [15]. CDRs are modelled using ConFREAD. If this is not possible, SAbPred reverts to using MODELLER. ConFREAD, a specific version of FREAD also uses information about the contact points that CDR loops make with their antigens. This increases the accuracy in modelling CDR-H3 from 2.25Å on average, to 1.23Å. The disadvantage with both methods is coverage. ConFREAD is not guaranteed to produce a prediction; the authors report the *'coverage'* is 70%. The latest version of SAbPred is known as *ABodyBuilder*, which combines SAbDab with FREAD and several other programs to create an online submission system for modelling antibodies[13].

Kotai's Anti-body Builder [81] models CDR-H3 by predicting whether or not the loop is *'bulged'*. It then looks for loop fragments, filtering by sequence similarity, secondary structure similarity, clash score and anchor residue similarity. The top 20 structures are refined using molecular dynamics simulations. Interestingly, the authors note that *'slight flexibility in the non-H3 CDRs'* was necessary to gain improvement in modelling CDR-H3.

Comparison of methods

The Antibody Modelling Assessment II (Antibody Modelling Assessment (AMA)-II) [75] presents a comprehensive review of various antibody modelling techniques. It considers several methods including Kotai Anti-body Builder, PIGS, the commercial program Prime (which uses PLOP) and RosettaAntibody. Generally, all methods managed an average of 1.2Å RMSD (between backbone carbonyl atoms) for non-CDR-H3 loops. The mean RMSD for CDR-H3 was around 3Å.

More specifically, the lengths of CDR-H3 ranged from 10 to 16 residues. Modelling methods were tested both with and without a reference framework for

[13] Available at `http://opig.stats.ox.ac.uk/webapps/sabdab-sabpred/Modelling.php`

CDR-H3. Some improvements were evident but *"even with this additional information, targets that proved difficult in the first round remained difficult"* [75]. The two models with the worst results across the various methods had the longest CDR-H3 loops.

The authors of SAbPred compared their method with the other approaches in the AMA-II, reporting an overall F_v region accuracy of 1.19Å, comparable to the methods tested in the AMA.

Despite the improvement in accuracy, revealed by the AMA, CDR-H3 accuracy varies between roughly 1.5 and 3Å average RMSD. Shorter loops are modelled more accurately. Marks & Deane [46] state that often, accuracies worse that 3.0Å occur. Thy go on to conclude that few of the methods used to predict CDR-H3 are purely knowledge based; CDR-H3 loops differ too much for prediction based on previous observations to be effective. Using more restrictive features, such as these in ConFREAD increases accuracy but at the cost of coverage. *Ab initio* approaches alone however, are more often more costly in terms of runtime and do not take advantage of the useful structural information already available. Research in this area appears to be moving towards hybrid approaches, such as Kotai Antibody Builder, CODA, and latest RosettaAntibody algorithm.

1.3.4 Aims

The aims of this project are as follows:

- Evaluate various neural network architectures against the accuracy resulting models.
- Experiment with different encodings of sequence and structure in these networks.
- Attempt to provide a confidence score for any predictions.
- Assess whether a neural network could provide a confidence score for models produced by other means.

Chapter 2

Materials and Methods

2.1 Machine learning

Machine learning was defined by Samuel [63] as *"giving computers the ability to learn without being explicitly programmed"*. Machine learning is a large topic; many methods and algorithms exist. We focus on these areas relevant to CDR-H3 modelling: these that have been applied to modelling tasks in the past and new developments in these techniques.

Many machine learning techniques, such as *Hidden Markov Models* and *Random Forests* have been used in various areas of bioinformatics and modelling. In this work, we focus on *neural networks*.

2.1.1 Neural networks

Artificial Neural networks (Artificial Neural Network (ANN) or just Neural Network (NN)) are one form of machine learning, inspired by biological neural networks. Such a network consists of artificial neurons with connections between them, roughly analogous to axons and synapses in the human brain. Such neurons receive a number of inputs, adjusted by weights. These inputs are combined using a particular function and a signal is emitted. These signals go on to form inputs to other neurons in different layers, or form the output of the network.

One of the earliest models is the Perceptron [62]. It is a form of linear classifier. It combines a set of inputs that are weighted, using a particular function, outputting a 1 or 0. Modern neural networks are built from a similar component but introduce a bias term and a non-linear transfer function. Perceptrons arranged in multiple layers are mathematically equivalent to a single layer; the transfer functions can be summed into a single function. In order to take advantage of multiple layers in a neural network, non-linear transfer functions must be used.

Neural networks have become quite popular at the time of writing. Major companies such as Nvidia, Google and Facebook make extensive use of *deep neural networks*, so called as rather than adding more neurons per layer, more layers are added. This has proved popular in computer vision tasks, largely because the hierarchical nature of the network reflects the problem in hand. For example, in recognising a simple shape in an image, one first learns to identify edges, then combinations of edges, then higher level shapes.

Very large datasets are now available for academic purposes, such as the MNIST database of handwritten digits[1] or ukWaC[2]. Such datasets are big enough to provide enough examples for an ANN to learn from.

Training a neural network requires a number of steps. Firstly, one needs to create a loss function. Taking all the free values of the network as variables to the function, one can calculate the *cross entropy loss*. As we know the underlying distribution from our training data, the output of our network can be compared to the correct values. This is referred to as the loss — minimising this value is the goal of training.

To minimise the loss we need to take the derivative of the loss function. A naive loss function would take all the training data, multiply it by all the free variables and sum all of them together to find the loss. However, working out the derivative of a loss function takes roughly three times the computation as finding the loss itself. Given that a naive loss function in a neural network can have millions of free parameters or more, computing the actual loss and correct gradient is prohibitive.

Rather than consider the entire training set, a random subset is chosen. This is called *Stochastic Gradient Descent* (Stochastic Gradient Descent (SGD)) and while the estimation it provides of the loss is poor, it scales relatively well with large datasets and large models. Various techniques can be used to mitigate SGD's limitations, such as taking smaller, many more steps.

Many of the mathematical operations in neural networks rely on linear algebra. 3D computer graphics are similar. Thanks to the computer games industry, dedicated graphics processor units (Graphics Processing Unit (GPU)) have constantly grown more powerful. Their optimised matrix functions make them almost perfect hardware for building and training neural networks (very recently, dedicated chips have been built by IBM for this purpose but are not available at present).

2.1.2 Neural network architectures

There are various kinds of neural network architectures. Each one is suitable for a particular kind of problem. Several different kinds have been used (or could be used) in bioinformatics problems, such as:

- Feed-Forward
- Recurrent (Recurrent Neural Network (RNN))
- Time-Delay (TDNN)
- Convolutional (Convolutional Neural Network (CNN))
- Long Short Term Memory (Long Term Short Term (LSTM))

Feed-forward or recurrent neural networks have either no dependencies on their previous outputs, or several respectively. Feed-forward are the most basic networks with no feed-back loops; data flows from one end and leaves at the other. Feed-forward networks often have three layers: an *input* layer, a *hidden*

[1] at `http://yann.lecun.com/exdb/mnist/`

[2] `http://wacky.sslmit.unibo.it/doku.php?id=corpora`

layer (so called because the user does not interact with it directly) and an *output* layer. Some networks have more than one hidden layer — so called *Deep Neural Networks* (DNN).

2.1.3 Time Delay Neural Networks

Time-delay neural networks have found use in processing natural language [77] and in modelling CDR-H3 [60] (discussed in section 2.1.9). Their advantage over simpler networks is their ability to incorporate the order of data into their learning. This is achieved by having neuronal inputs that are sensitive to both a feature and a time. Figure 2.1 illustrates this idea. Features, z, have a time step, t, associated; in this case successive inputs look backwards by one time-step from the previous input. Each input has a weight, just like a normal neuron, but the input itself is only associated with that particular point in time.

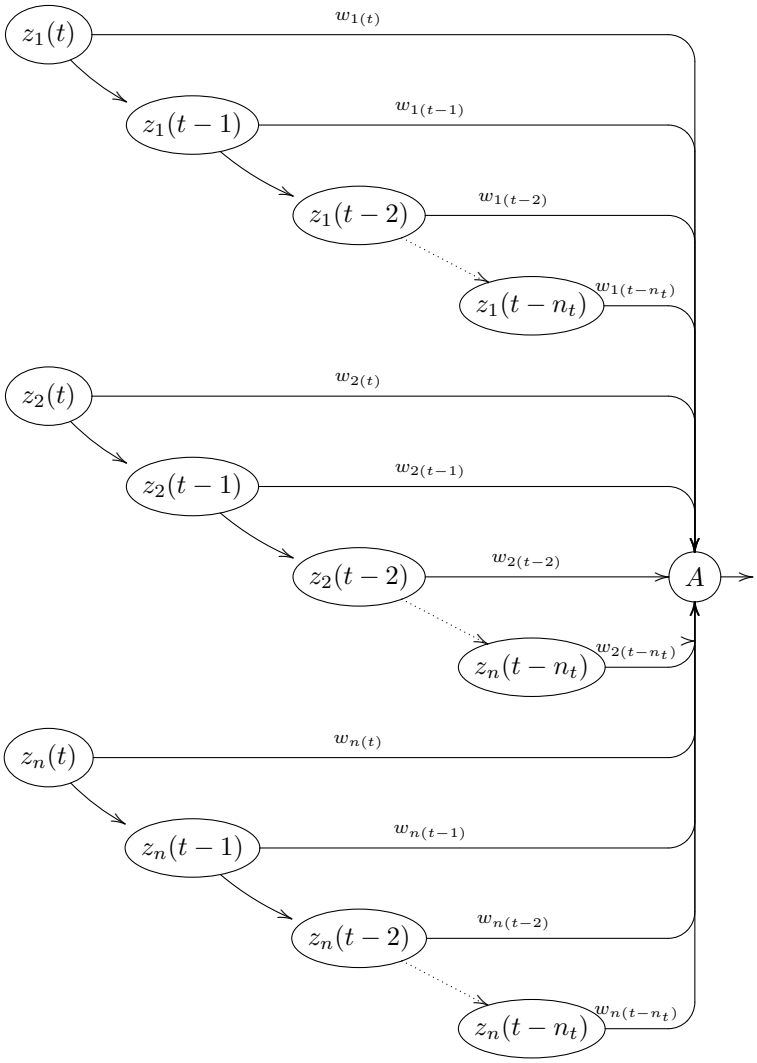


Figure 2.1: Diagram of a single Time-Delay Neuron. The inputs to the neuron are on the left, marked z. Each input occurs again for a certain number of time-steps (e.g $z_2(t-1)$). While the input value is repeated, it has a unique weight for each time-step (in this case $w_{2(t-1)}$).

2.1.4 Convolutional Neural Networks

Convolutional Neural Networks (CNN or ConvNet) have one or more layers that apply a convolving filter which reduces the number of input connections to that layer, whilst increasing the layer's *depth*. Applied originally to image processing tasks [38], CNNs have found use in a variety of fields, including natural language processing [35] and recommender systems [76].

CNNs are inspired by the organisation of neurons in the human visual sys-

tem, where sets of neurons respond only to certain areas of the visual field, with some amount of overlap. At each level in the CNN, different hierarchical features are learned. For example, the first level of a CNN may have neurons that are sensitive to straight lines. The second may be sensitive to a particular shape such as triangle while the last layer would build on these features and be sensitive to road warning signs.

Perhaps the most famous example of a CNN in widespread use is the digit classifier by LeCun *et al* [38]. LeNet-5 as it is known, uses 7 layers and a series of convolutions.

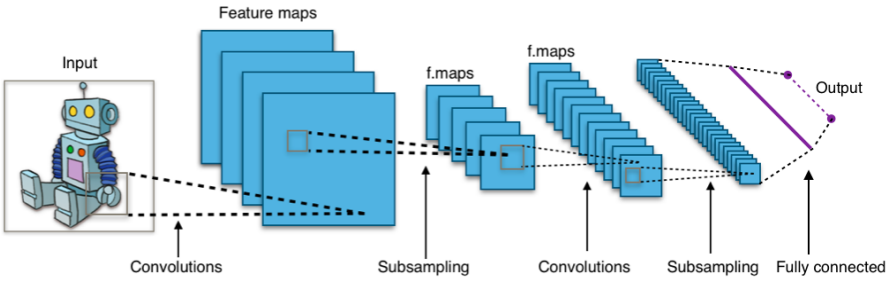


Figure 2.2: A typical CNN architecture. `https://commons.wikimedia.org/wiki/File:Typical_cnn.png`.

Figure 2.2 shows a typical CNN architecture. CNNs introduce the concept of a *feature map*. If one considers a typical computer image, each pixel is represented by 3 values: red, green and blue. These channels can be mapped to a particular filter (also known as a *kernel*) that is sensitive only to that channel (or colour). The final result of applying this kernel to the entire image is a feature map. As the filter is larger than one pixel (typically) the resulting feature map will be smaller than the original image. However, there will be multiple feature maps for each layer. The result is CNNs tend to get *smaller but deeper* at each successive layer.

Each convolutional layer has a number of *kernels* which are convolved over an input image, creating a *feature map*. The number of kernels convolved over the image is equal to the number of feature maps which is equal to the depth. A typical example might be the first layer of a *ConvNet* that looks at an RGB image. Three kernels are convolved over the image, each one looking at a single colour channel. If such an image was 300 pixels in both height and width, a 3 by 3 pixel kernel could be moved over the image 100 times in each direction, creating a new layer of neurons 100 x 100 x 3.

The key difference is that each neuron in the *convolutional layer* is fully connected to the *depth* information but only locally connected to the *spatial* dimensions (the width and height). For example, a single neuron might be attached to an input volume which has a size of 16 x 16 x 20. If the kernel in use is 3 x 3 in size, each neuron would have $3 * 3 * 20 = 180$ connections, rather than 5120 connections, as would be the case if the neuron was *fully connected* to every possible input value. The weights for each neuron are shared spatially, but not across the depth.

Other functions are often added to CNNs in order to increase accuracy. *Rectified linear units* (ReLU) add a small amount of non-linearity to aid in learning.

Max-pooling takes a set of inputs (typically from a feature map) and reduces the output by simply returning the largest value. This feature was implemented to control over-fitting in the neural network. Another method commonly used is *drop-out*, where neurons are randomly chosen to be ignored during training. Such features can now be found in many other NN architectures.

The depth at each layer is chosen by the designer of the network; another hyper-parameter. Values such as the *stride* (the distance the kernel is *shifted* at each step), the *padding* (extra zeros surrounding the input) and the kernel dimensions are also chosen arbitrarily but have a mutual constraint that the kernel must exactly fit the input volume.

$$h_{ij}^{k} = tanh((W^{k} * x)_{ij} + b_{k}). \tag{2.1}$$

Equation2.1 shows how a particular value (i, k) in feature map h is computed, from a set of weights W^k. The neurons, their weights and biases (b_k) are not fully connected across the convolutional layer. The same weights are also used across each stride of the kernel.

Relationship to Time-delay neural networks

Time-delay neural networks (TDNN) pre-date convolutional networks [77].

$$o_{k,p} = f_{ok}(\sum_{j=1}^{J+1} w_{kj} f_{yj}(\sum_{i=1}^{I} \sum_{t=0}^{n_t} v_{j,i(t)} z_{i,p}(t) + z_{I+1} v_j, I+1)) \tag{2.2}$$

Equation2.2 defines the output of a complete TDNN. Focusing on a single unit (the inner set of summations) the output is defined as a sum of the inputs over a set time period, with each time delay having a particular weight ($v_{j,i(t)} z_{i,p}(t)$). This is analogous to the convolutional kernel. Consider an image with dimensions [1,300], and a kernel with size [1,5]. As the kernel strides across the image, new pixels are presented and older pixels leave. The width of the kernel (5) is analogous to the number of time delays in the TDNN. The output of the unit at any single step is dependent not only on the current pixel, but on these surrounding it. With careful choice of parameters, a convolutional network is a reasonable approximation of a TDNN.

2.1.5 Recurrent Neural Networks

Recurrent Neural Networks (RNN) take their outputs and *feed-back* to their own inputs, potentially creating a circular dependency. This gives them the advantage of processing arbitrary length inputs. However, training such networks is more complex as the error gradients tend to either vanish or explode exponentially as the time between important events increases — optimising such networks can be difficult [22]. RNNs are often *unrolled* for a certain number of time steps to aid in design and implementation. Figure 2.3 illustrates this architecture.

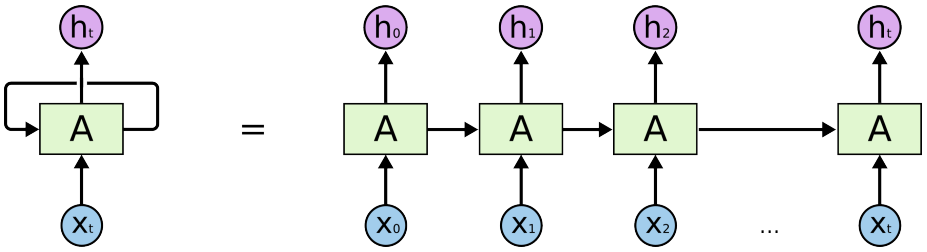


Figure 2.3: Un-rolling an RNN reveals the inputs from new data, the inputs from earlier time-steps and the corresponding outputs. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

2.1.6 Long-term Short-term neural networks

Long-term, short-term neural networks (LSTMs) are an extension of recurrent neural networks that address some of the problems of RNNs. As the name suggest, LSTMs are capable of *remembering* relationships that span both short and long distances in time. Typically, they have been used in natural language processing; an example being prediction of the next word in a sentence. The context required for such a prediction depends not only on the word immediately preceding the prediction, but also the earlier parts of the sentence.

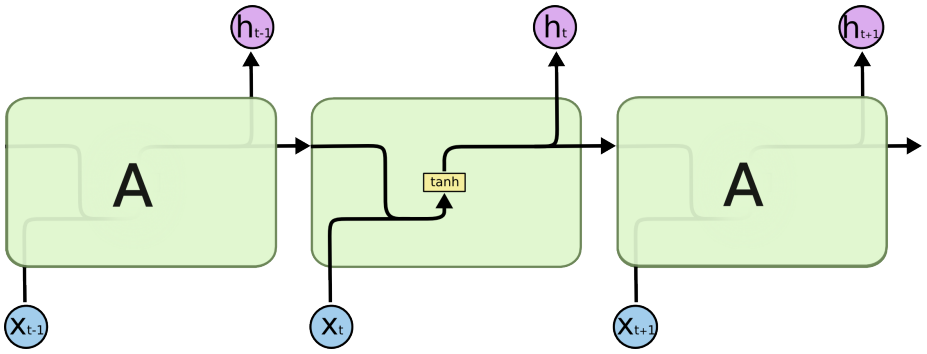


Figure 2.4: A repeating module in a standard RNN typically contains one component - a layer of neurons with an activation function. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

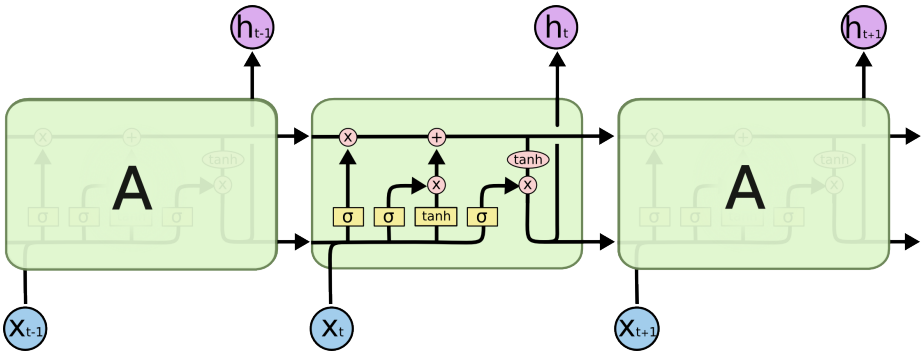


Figure 2.5: A repeating module in an LSTM contains four interacting systems. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

LSTMs attempt to overcome the shortcomings of RNNs, chiefly the *exploding and vanishing gradient problem*. Several components comprise a LSTM *cell*. Continuing with the brain analogy, we replace the *feed-forward neuron* with a *memory cell* (or more commonly, just cell). The term cell is used because each cell now contains a small amount of memory - it has a particular *state* over time. In fact, the default LSTM implementation holds two states - the cell and history states. These states have a particular size, chosen by the user.

The components of an LSTM cell differ over various papers and implementations, however the following components are common[22]:

- Forget gate
- Input gate
- Output gate
- Block input
- Cell state

The cell state $c_{(t)}$ can be thought of as the long-term state as it does not pass through any transformation, save for the input addition and forget gates. Some *memories* are added and others are dropped. The short-term $h_{(t)}$ state passes through the forget, input and output gates.

The components of an LSTM block, and their associated functions are:

- The forget gate — $f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$
- The input gate — $i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$
- The output gate — $o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$
- The cell state vector — $c_t = f_t \cdot c_{t-1} + i_t \cdot \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$
- The output vector — $h_t = o_t \cdot \sigma_h(c_t)$

where

- x_t is the input vector at time t.

- h_t is the output vector at time t.
- c_t is the cell state vector at time t.
- W,U and b are the weights and biases of the neural network in question.
- f_t is the forget gate vector — the weights of remembering old information.
- i_t is the input gate vector — the weights for acquiring new information.
- o_t the output gate vector — the candidate for output that feeds back.
- σ_g is a sigmoid activation function.
- σ_c is a hyperbolic tangent activation function.
- σ_h is a hyperbolic tangent activation function (replaced with other functions in various LSTM variants).

One key aspect of the LSTM is the output at t, which becomes the input at $t+1$, has no activation function, therefore it does not degrade in the way in which a normal RNN would, thus avoiding the vanishing gradient problem [73].

The forget gate allows a LSTM cell to forget its own state - effectively resetting the cell.

Long Short Term Memory (LSTM) neural networks are designed to learn from sequential data; a subclass of RNNs. LSTMs are capable of scaling to longer time frames and are not specific to any particular problem [22]. They were designed to overcome the problems involved with RNNs, and have been used in a variety of tasks — from handwriting recognition to polyphonic music modelling.

LSTM networks are built from LSTM *blocks*. Several kinds of blocks exist with various features added or removed. The *traditional* LSTM block (figure 2.5) contains 4 different block with different activation functions. The behaviour of the block is controlled by a set of *gates*: the *forget gate*, the *input gate* and the *output gate*. Each of these is connected to one of the neural networks.

2.1.7 Sequence Labelling, Classification and Sequence to Sequence

Recurrent neural networks, built of LSTMs, produce outputs at each time step through-out their run (h_t in figure 2.5). There are two main approaches one can take when processing this state:

- Pass each h_t through a dense layer of shared weights.
- Take the final state only, passing through a dense layer of shared weights.

The first approach is often referred to as *Sequence Labelling*. One common example of this technique is classifying each word in a sentence. As each word is presented, the state of the particular LSTM cell is passed through the output-dense-layer and a label is given. This has the effect of providing the same number of labels as their are entries.

The disadvantage of this technique is the earlier items in the sequence do not have any previous history to guide their decision, which can result in earlier

items in a sequence being misclassified. This can be somewhat ameliorated by using a *bi-directional LSTM* (discussed in next section). We test the sequence labelling approach with a discretised representation of the torsion angles discussed in section 2.4.7.

The second approach is often referred to as *Sequence Classification*; the entire sequence is given only one classification based on the output of the final LSTM cell. This *last-relevant-output* is feed into a dense-output-layer and the final classification is considered. Rather than output a class, we take the final state and use the dense layer to produce a set of real numbers, representing the ϕ and ψ angles.

Sequence-to-Sequence networks (*Seq2Seq*) are somewhat different from the other two approaches. At their core, they contain an RNN, usually built of LSTM units, but they differ considerably in the overall architecture, making use of a *decoder* and *encoder*. Seq2Seq networks have found considerable use in human language translation [70] among other variable sequence-based tasks[3].

Seq2Seq builds on the concept of the *autoencoder*. Rather than provide a network with a dense data format decided by hand, an autoencoder is fed both the input and desired result. It then *learns* its own internal representation, sometimes referred to as a *meaning* or *thought* vector.

With the encoding learned, the *Seq2Seq* network can be tested by providing the input sequence only. This is converted to a meaning vector which is then passed to a decoder, which converts the meaning vector back into our desired output. Figure 2.6 shows this process in more detail.

[3]See also `https://research.googleblog.com/2016/09/a-neural-network-for-machine.html`

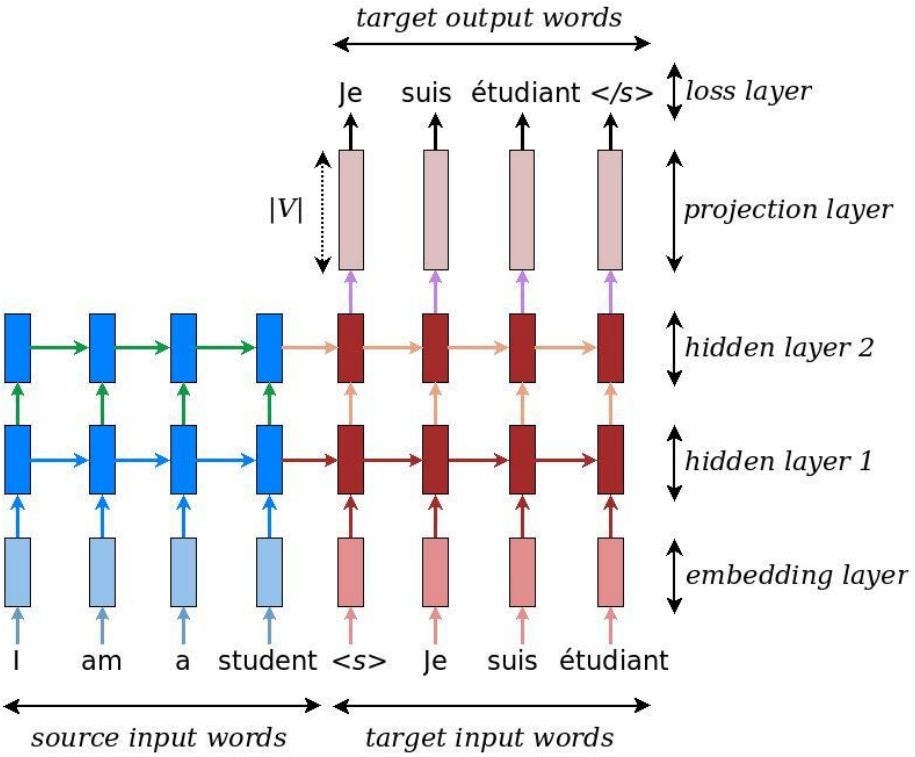


Figure 2.6: An overview of a Seq2Seq network, translating a sentence from English to French. The hidden layers are both RNNs with a dense, fully-connected projection layer converting the cell states into classifications.

Another difference is the addition of *start and end sequence markers*. In the example given in figure 2.6, the French translation contains fewer words. The counters $< s >$ and $< /s >$ denote the end of the input and the end of the translation respectively. In the case of natural language translation, this is an essential feature. In our case however, the output sequence will always be identical in length to the input sequence.

2.1.8 Machine learning, Bioinformatics and Modelling

Machine learning has been used in various areas of bioinformatics. Several examples exist related to modelling proteins.

GenTHREADER, mentioned in section 1.3.1, generates a set of scores which are fed into a neural network to produce a final rating of the sequence to fold alignment [32].

The Neural Network for Promoter Prediction (NNPP) is an example of using a neural network to locate areas in a sequence that might be gene signals [85]. It is an example of a TDNN.

Sønderby and Winther use an LSTM to predict secondary structure from sequence [73].

2.1.9 Neural networks and CDR-H3

Reczko and Martin [60] attempted to predict the conformation of CDR-H3 using a TDNN. They achieved an accuracy of around 2.652Å. The training set loops were between 8 and 17 residues long. Accuracy was sub 2.0Å in loops of 7 residues or less. The training set consisted of 1046 loops, from a total set of 1976.

Knowledge of the environment is not part of the input data to the neural network. The authors suggest that this is what accounts for the drop-off in accuracy for longer sequences, pointing out that similar accuracies have been achieved by other methods that also do not consider the environment.

2.1.10 Tensorflow

Tensorflow is a library published by Google for machine learning[4]. It is built around the concept of passing tensors through a graph, built from various operations on these tensors, one after another. Figure 2.7 describes how the classic model neural network - neurons, weighted connected edges and a single output - map to a matrix or tensor representation.

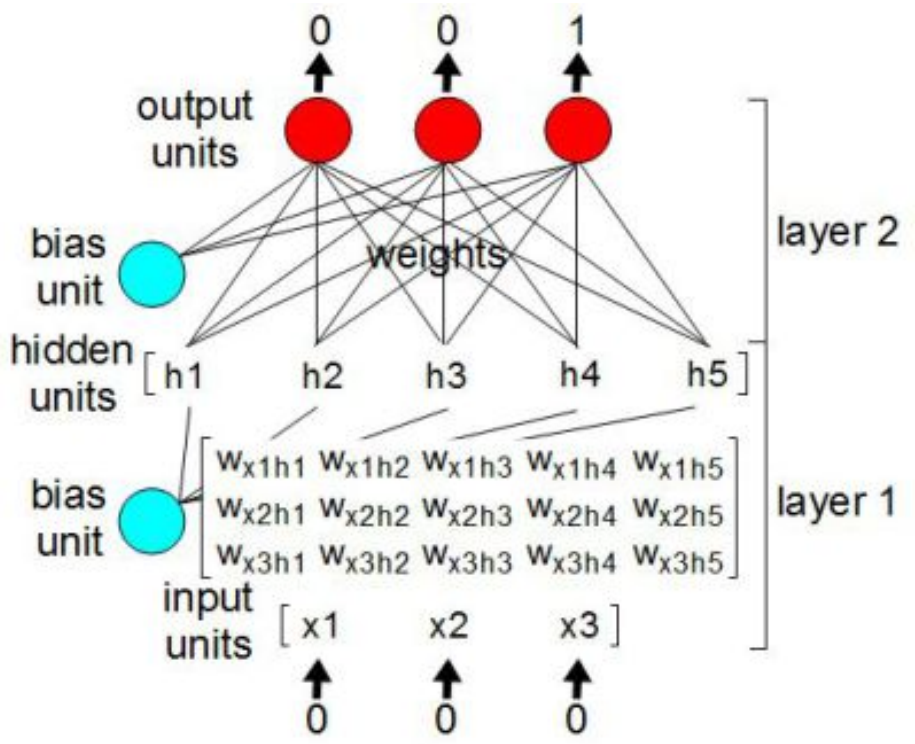


Figure 2.7: A two layer ANN with the first layer described in terms of tensor / matrix operations. `https://hackaday.com/2017/04/11/introduction-to-tensorflow/`.

[4]Available from `https://www.tensorflow.org`

2.1.11 Variable lengths

Neural networks have a fixed size for each layer; there are a fixed certain number of neurons and connections. In image classification tasks for example, the input images are resized to fit the fixed size before processing.

In the *AbDb* dataset the longest CDR loop is 28 residues long, with the shortest being 2 residues in length. The majority of loops cluster around the 8 to 12 range.

Reczko *et al*[60] built a network that supports loops of up to 22 residues. Loops that are shorter are padded with zero values. This particular architecture considers residues in pairs, with the first residue being paired with the last, the second residue being paired with the penultimate residue, and so on. When the number of residues is exhausted, the remaining pairs (referred to as *spinors*) are set to zero.

We create a mask for each training example, based upon the input data. This mask is set to 1 where a value should be considered, and 0 otherwise. The cost functions and output functions are multiplied by this mask, effectively setting the neuron outputs to 0 and ignoring their contribution to the error. [5]

2.1.12 Epochs and batches

When a single training example is presented to the network at a time, one can be said to be performing *stochastic gradient descent*. In *batch gradient descent*, the entire training set is presented at once. Typically, most networks perform *mini-batch gradient descent*, which presents a subset of the training data at each training step.

Mini-batch gradient descent has three advantages over the other methods. Firstly, it takes advantage of new hardware, particularly GPU hardware, which is optimised for matrix multiplications on a large scale. Using a mini-batch decreases the learning time[23]. Secondly, using the entire dataset is often infeasible due to memory limitations and the number of parameters required to calculate the gradient of the cost function. Finally, using a single value results in a very noisy cost value over time. Using a number of simultaneous samples result in a smoother gradient, avoiding some of the smaller local minima and maxima. This can be seen as a form of *normalisation*, improving the robustness of the network.

The size of the batch is one of a number of *hyper-parameters* the user of the network must tune for optimum performance. Another such parameter is the number of *epochs* a network will run for. A single epoch is defined as the the period of training between where a network sees a repeated datum. Once a network has seen the entire training set, an epoch has passed. The network will likely be trained over the course of several epochs, the number of which is decided by the user, potentially with some early stopping criteria in mind (discussed in 2.2.1).

Tensorflow encourages the user to pass training examples as batches (to the point where certain functions do not work unless you pass in a batch). Typically, we use values between 5 and 50 in step with the training-set size.

[5] approach is adapted from the work by Danijar Hafner `https://danijar.com/variable-sequence-lengths-in-tensorflow/`

When training the network, a batch is created at random from the training set, then discarded. Per epoch, each batch contains a random set of unique loops.

2.1.13 Gradients

Batch normalization is a technique one can apply, when using batches, to address the problem of exploding and vanishing gradients. Given a mini-batch, Batch normalization zero-centres and normalises the inputs, following with a scale and shift on the result. This operation lets the *"model learn the optimal scale and mean of the inputs for each layer"*[23]. In order to centre and normalise the inputs, the mean and standard deviation is computed across the current batch of input data.

Gradient clipping helps to reduce the exploding and vanishing problem by limiting the minimum and maximum gradients to a particular value (another hyper-parameter), usually -1.0 and 1.0 respectively. Certain networks we have tested have a tendency to create large gradients resulting in numerical error; gradient clipping fixes this problem. Clipping to this range makes particular sense as the output of the network represents the sine and cosine of the angle in question.

2.1.14 Basic error function

Equation 2.3 describes the basic error function - the mean of the *sum-squared-difference* between the predicted (x) and actual(t) angles. Here, M is a mask, set to 1 or 0, with L being the length of the CDR loop in question. I is the maximum length of a CDR in the current dataset (the input tensors within our neural network must be of a fixed size).

$$\frac{\sum_{i=1}^{I}(x_i - t_i) * (x_i - t_i) * M_i}{L} \tag{2.3}$$

x and t in equation 2.3 are tensors that contain the sine and cosine of the ϕ and ψ angles (as radians). While this error function is simple to compute, it only considers each angle on an individual basis, rather than the global structure. While it is expected that neurons with larger errors will be adjusted more than these with smaller errors, the error function does not model the explicit criterion that a loop must start and end at particular locations.

We refer to the number returned by this function over the training set as the *training error*. We can also apply this function to items from the validation set, referred to as the *validation error*.

2.1.15 Optimizers

Tensorflow provides a set of *optimizers* one can use in order to minimise the cost function and train the network. In our experiments we use *Gradient Descent*, *Adagrad*[13] or *Adam*[36] optimizers.

Adagrad treats each feature seperately, creating a learning rate for each. The sum of the squares of the past gradients are used to alter the learning rate over time, reducing the size of the steps the optimizer will take. *Adagrad* is recommended for use with sparse datasets, such as the bit-field representation

of our amino acids[57]. It is the optimizer used in our networks that produce a range of continuous values along with standard gradient descent.

Adam appears to be the most widely used of the various optimizers[23]. In addition to storing the squares of past gradients, Adam also keeps an exponentially decaying average of past gradients, providing a *momentum*. We use *Adam* in these networks where a classification is required, described in section 2.1.7.

Arguably, the most important *hyper-parameter* is the *learning rate*. If this number is too large, optimizers will make large changes to the neurons, most likely never converging on the lowest minima. If this number is too small, the optimizer will get stuck in local minima. Both *Adagrad* and *Adam* start with one particular rate, then gradually adjust the rate downwards with each step. In the case of *Adagrad* we tend to start with 0.45, whereas *Adam* begins at 0.004.

It has been suggested that altering the learning rate whilst training improves performance [23]. Many techniques exist for altering the learning rate across time, such as exponential decay and performance scheduling among others. These can be implemented manually within Tensorflow, however both *Adagrad* and *Adam* monitor their own learning rates and adjust accordingly based on past performance, so manually changing their rates is not recommended.

2.2 Neural networks in Tensorflow

2.2.1 Regularisation

Neural networks can suffer from the problem of *over-fitting*, also referred to as *memorising*. Rather than learning an underlying relationship in the data, the network merely remembers the examples it has been shown. This can detected by comparing the training error with the validation error. Should the validation error begin to increase as the training error continues to decrease, over-fitting is likely.

Regularisation is particularly important to this work, as the *AbDb* is relatively small, when compared to other data-sets in use in other deep-learning networks. Once the network begins to see similar examples, after a certain number of epochs, the tendency to over-train increases.

The process of constraining a model in order to reduce the possibility of over-fitting is called regularisation. One simple example of regularisation in a polynomial model is to reduce the number of polynomial degrees.

Ridge regression (or Tikhonov regularisation) [23] involves applying a regularisation term to the loss function. Equation 2.4 describes the term that is added to the existing cost.

$$\alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2 \tag{2.4}$$

$$\frac{1}{2}(\|w\|_2)^2 \tag{2.5}$$

In the case of a neural network layer, θ refers to the weights; equation 2.5 describes the same equation in different terms. $\|w\|_2$ represents the l_2 norm of the weight vector. This form of regularisation is often referred to as l_2 regularisation.

Lasso (Least Absolute Shrinkage and Selection Operator Regression) is similar to Ridge regression but uses the l_1 normalisation of the weight vectors instead of half of the square of the the l_2 norm[23], shown in equation 2.6.

$$\alpha \sum_{i=1}^{n} |\theta_i| \tag{2.6}$$

Tensorflow has several functions one can apply to a tensor in order to apply regularisation [6]. For example, l_1 regularisation can be added thusly:

```
base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")
reg_losses = tf.reduce_sum(tf.abs(weights1))
        + tf.reduce_sum(tf.abs(weights2))
loss = tf.add(base_loss, scale * reg_losses, name="loss")
```

Early-stopping is considered a form of regularisation. Typically, when a network begins to over-fit the training data, the error on the validation set begins to increase as the training error decreases. It is preferable to stop training before this occurs. Recording the error over each epoch or step, an algorithm can decide to stop training early as soon as the validation error begins to significantly increase after a period of decreasing. This necessitates the use of averaging in order to smooth out local fluctuations in the error rate. In our early experiments, we did not use early-stopping, rather running for a set number of epochs. Later networks would monitor the learning rate, saving the most accurate version generated.

Drop-out is another regularisation method proposed by Hinton *et al*[28]. The process is relatively simple - every neuron is given a chance of being ignored during a particular step in the training process. Once training is complete, all neurons are considered. The drop-out probability is usually 50% but other values can be used[23]. Drop-out is easily implemented in Tensorflow but results in another *hyper-parameter* to tune.

2.2.2 Convolutional nets

Tensorflow provides several functions for creating convolutional networks. We use a single dimensional window, 5 units long and a single convolutional layer. Adding extra layers with different window sizes and depths is also possible and we briefly consider up to 3 convolutional layers, each with a smaller window size than the last. This hierarchical approach is an attempt to capture useful information at different scales.

The output of our network is usually fed into a fully-connected or dense layer, which presents the final result. In the majority of our experiments, this layer is $max\_cdr\_length * 4$, as each residue produces 4 numbers corresponding to the sine and cosine of the ϕ and ψ angles. This layer may also have a *bias* term applied.

[6] `https://www.tensorflow.org/api_docs/python/tf/contrib/layers/apply_regularization`

2.2.3 LSTMs

The Tensorflow implementation of the LSTM cell is implemented based on the work of Zaremba *et al*[83]. We investigate a small variety of sizes - from 64 to 512. Typically, only a single LSTM layer is included due to time and resource constraints. A single LSTM RNN layer takes longer to train in practice and requires more more memory depending on the number of steps to unroll.

Like the convolutional net, the LSTM layer is fed into a fully connected layer, $max\_cdr\_length*4$ units wide. Bidirectional LSTMs perform a combination step on the two passes before passing to the output layer.

2.2.4 Regularisation in LSTMs

"Unfortunately, dropout, the most powerful regularization method for feedforward neural networks, does not work well with RNNs."[83]. Fortunately, Zaremba *et al*[83] provide a solution. Drop-out is applied only between the non-recurrent connections, i.e the h_t state when being fed into a second RNN layer.

Drop-out cannot be applied to the final dense layer, as the primary purpose of this layer is to change the data into the final required format. Some of the networks in our experiment only have a single RNN layer and are therefore at greater risk of over-fitting.

2.2.5 Variations and approaches

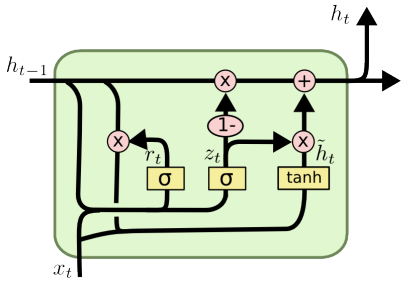
Our initial investigation looked at a naive LSTM implementation, listed in appendix B.4.

The number of time-steps is analogous to the number of residues. At each step, the LSTM considers the current step and all the previous steps, however all steps are available once the entire sequence has been considered. One may take all the time-steps and combine them with a fully connected layer, or just the last relevant step. We consider both approaches.

In many problems where LSTMs are applied, the size of the input data is not known. In our case however, the entire sequence for the CDR loop is known at prediction time. We can therefore use what is known as a *bi-directional LSTM*. As the name suggests, such a network considers both directions of the sequence, forwards and backwards. A separate function is used to combine the two passes. In our case, we use addition, both of the entire sequence of steps, and with just the last step of both passes.

It is debatable whether addition is the best choice for combining passes. We also consider both taking an average of the two passes, and concatenating both passes into a double wide layer for sending to the final dense layer.

Figure 2.4 shows a typical LSTM unit, complete with several gates. Other units based on the original LSTM exist. We consider two major variations: *gated recurrent units (GRUs)* and *peephole gates*.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 2.8: A gated recurrent unit (GRU). The equations for each output are shown on the right. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

GRUs were proposed by Cho *et al*[10]. The major differences are the combination of the cell and hidden states, a single controller for both the forget and input gates, and the removal of the output gate. Figure 2.8 describes the unit, with the associated equations for each gate within the unit.

Peephole gates were proposed in 2000 by Gers and Schmidt[21]. This technique adds a weighted connection between the cell state and the forget, input and output gates of the same cell. The effect allows the unit to occasionally ignore input signals from previous states, or error signals during back propagation.

We use GRUs where possible in our networks as they are simpler than a traditional LSTM, take less time to train and provide results of similar accuracy[23].

2.3 Recreating structure

2.3.1 NeRF algorithm

There are several methods to convert a series of torsion angles to cartesian coordinates, most of which are described by Parsons *et al*[56]. In their work, they compare general rotations, Rodriguez-Gibbs and quaternion based solutions, with their *natural reference methods*, known as *NeRF* and *SN-NeRF*.

The intuitive approach is to perform a series of trigonometric rotations, possibly creating a series of matrices or quaternions. Parsons describes a second method where the atom is placed in two steps. The first step uses only the bond angles and distances to place the atom into position. The second step rotates the atom into the reference frame, based on the previous three atoms.

The NeRF method is summarised as follows:

$$\overrightarrow{D}_2 = (Rcos(\theta), Rcos(\phi)sin(\theta), Rsin(\phi)sin(\theta)) \tag{2.7}$$

Where $R = bond_{CD}, \theta = angle_{BCD}, \phi = torsionBC$.

The atom to be placed is labelled D, with atoms A, B, C already placed in space. D is placed at a starting position equal to the bond distance between C and D. Equation 2.7 describes the first step in placing the new atom at $\overrightarrow{D}_2$

$$\hat{\hat{M}} = [\hat{bc}, \hat{n} \times \hat{bc}, \hat{n}] \tag{2.8}$$

Equation 2.8 describes the construction of the final rotation matrix, where bc is the normalised vector between atoms B and C, and n is defined as the normalised cross-product between $\overrightarrow{AB}$ and $\hat{bc}$.

The bond angles and lengths are derived from the work of Laskowski *et al*[37], and Fox *et al*[20]. The final values are listed in table 2.1 and table 2.2.

Atom pair	Distance
N - Ca	1.4615Å
Ca - C	1.53Å
C - N	1.32Å

Table 2.1: Distances between atoms used in the NeRF algorithm.

Atom trio	Bond angle
Ca to C	109°
C to N	115°
N to Ca	121°

Table 2.2: Angles between atoms used in the NeRF algorithm.

Fox *et al* note that the distance between the nitrogen and carbon alpha atom within a proline residue is closer to 1.355Å and that the angle between carbon alpha and carboxyl carbon, via nitrogen, tends to vary by 5± degrees. These differences result in a slight increase in RMSD between $C\alpha$ atoms, versus GenLoop.

Some reconstruction algorithms ignore the omega (ω) angle, making the assumption that the amino-acid planar bond between the carboxyl carbon and nitrogen is always 180°. In certain cases, proline can be found in its *cis* conformation, affecting the omega angle. Within the *AbDb* dataset, 390 loops have omega angles are more than 30° away from 180°, representing 7.5% of the total.

To assess the accuracy of reconstruction from torsion angles, NeRF created structures were compared to real atom positions derived from the PDB files. The torsion angles themselves are derived from the original PDB files, using the code in appendix B.2. These angles were verified against the output of the program *torsions*[7]. In addition, a third set of coordinates, generated by the program *genloop*, derived from the same set of torsion angles was also used as a comparison case.

We compare three sets of PDBs : these generated by the program *genloop*, these generated by NeRF and the original loop extracted from the full antibody PDB. Using *PDBfit*, the generated loops are matched to the original loop and an RMSD score based on the backbone atoms is derived. The PDBs are drawn from the *AbDb* dataset. The experiment is performed twice; once with the real omega angles and once with omega set to 180°. The mean scores are shown in table 2.3.

In addition to generating a mean value, several models were found with with RMSDs over 1.0.

[7] `http://www.bioinf.org.uk/software/torsions/index.html`

Creation method	RMSD with real Omega	RMSD with 180 omega
GenLoop	0.785	1.52
NeRF	0.846	1.539

Table 2.3: Difference in Ångströms between loops created with NeRF and GenLoop.

Creation method	Total over 1.0 RMSD with real omega	Total over 1.0 RMSD with 180 omega
GenLoop	514 (17.9%)	1269 (44.2%)
NeRF	679 (23.6%)	1442 (50.2%)

Table 2.4: Comparing real and fixed omega angles with GenLoop and NeRF.

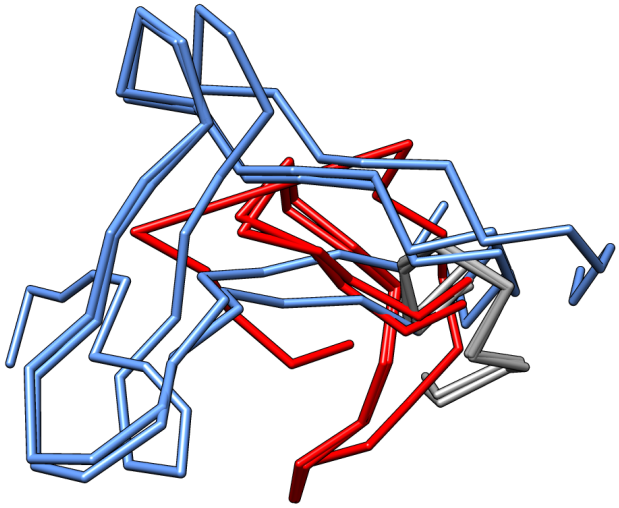


Figure 2.9: Three examples of reconstructed loops from *AbDb*: 1RUR_1 (grey), 1RZG_2 (red) and 3MME_2 (blue). 1RUR_1 shows a good alignment of both methods. 1RZG_2 shows both both methods unable to converge on a real loop that appears to be incorrect. Finally 3MME_2 shows a reasonable alignment with NeRF but a poor alignment with GenLoop.

Figure 2.9 shows three examples of reconstructed loops, compared against the ground truth. The loop in red, 1RZG_2, has a number of missing residues in the centre, though both algorithms still attempt the reconstruction. 3MME_2 (blue) shows a good alignment with NeRF but a poor one with GenLoop.

2.4 Datasets and data representation

Our experimental datasets are derived from two major sources: *AbDb* and the Protein Databank (this latter set is referred to as *LoopDB*). Both sets of PDB

files were entered into a database for ease of batching and examination.

2.4.1 AbDb dataset

The initial dataset we consider is derived from *AbDb*[18]. This data set is derived from a compilation of antibody structures in the Protein Databank. *AbDb* provides antibody numbers with Chothia, Kabat and Martin numbering schemes, described in 1.2.2. At the time of writing the database has a number of antibody structures, described in table 2.5.

At the time of writing, *AbDb* contains 2935 entries, of which 1716 are considered duplicates or redundant. We performed experiments on the entire set with no restriction, removing duplicate entries from all the datasets used, and removing duplicate entries in the validation and test sets only.

Datasets	Complex type	Processed PDB files	Resultant antibodies	non-redundant antibodies
Complete antibody	Protein	976	1591	673
	Non-protein	275	374	194
	Free antibody	580	973	531
	Complete dataset	1794	2938	1184
Light chains	Protein	12	17	5
	Non-protein	9	17	5
	Light only	77	137	48
	Complete dataset	86	171	52
Heavy chains	Protein	88	162	74
	Non-protein	5	11	5
	Heavy only	47	94	51
	Complete dataset	134	267	121

Table 2.5: Counts of the various types of structures in the AbDb dataset, taken from `http://www.bioinf.org.uk/abs/abybank/abdb/`

A certain amount of processing takes place for each PDB file extracted from the Protein Data Bank. To briefly summarise, chains are identified, non-antibody chains are checked for redundancy and whether or not they can be considered an antigen, and if so, does they contain contact CDRs. The complete processing pipeline is described by Ferdous *et al* [18].

Although this resource is invaluable to researchers from several fields, the number of training samples is still low in comparison to other training sets in similar fields. For example, the famous MNIST database[8] contains 70,000 images. Nevertheless, this dataset forms the basis of our experiments.

We process each entry in *AbDb* by extracting the atoms and residues from 95 to 102 inclusive, on the heavy (H) chain. These extracted loops are stored within a database for use by our networks. Atom positions and residues are stored first. A second step generates the torsion angles for each loop, entering these into the database for later retrieval. Finally, the redundancy data is also stored in the database to allow for the creation of non-redundant sets.

[8] `http://yann.lecun.com/exdb/mnist/`

2.4.2 LoopDB dataset

LoopDB is the name given to a much larger set of structures that are similar to CDR-H3 loops, extracted automatically from the Protein Databank.

All the available antibody structures within the PDB are analysed to find the distances between the three residues are the N-terminus (i.e H92, H93 and H94) and the C-terminus (H103, H104, H105). The mean and standard deviation of these 9 distances are then calculated and used as criteria to select protein loops from the entire Protein Databank. If a particular stretch of protein lies within two standard deviations of these distances, the loop is included in the dataset.

At the time of writing, the *LoopDB* set consists of 561477 loops. Of these, 403298 are between 3 and 32 residues long, with no errors or unknown residues. Each loop contains three extra residues at both ends, which are removed in a pre-processing step before being added to our database for use by our networks. Again, just as with *AbDb* we store the atom positions and residues first, before the second step - generating and storing the torsion angles.

2.4.3 Rejections and redundancies

Both *AbDB* and *LoopDB* contain loops that cannot be used for the following reasons:

- Models do not contain complete lists of atoms, making complete backbone angle determination impossible.
- Residues labelled as UNK, GLX, CSO or ASX.

Many of the loops in *AbDb* are duplications of others. The complete list of duplications can be found at `http://www.bioinf.org.uk/abs/abdb/Data/Redundant_files/Redundant_LH_Combined_Martin.txt`. This information is incorporated into our database. *LoopDB* also has several redundant loops, but no list currently exists. We classify a loop as redundant if another loop within the database has exactly the same residues in exactly the same order.

2.4.4 Dataset size versus parameter count

The Vapnik-Chervonenkis dimension (*VC* dimension)[6] is a number that "can be viewed as a measure of the richness (or diversity) of the collection of all functions $x \rightarrow N(\theta, x)$ that can be computed by N for different values of its internal parameters θ"[6] where N is the number of outputs. This number is proportional to the number of training examples that are needed to train a network to approximate a target function. *VC* can be applied to many functions, not just neural networks, however in this case, *VC* is dependent on the number of weights, neurons and the activation function in use.

Shai Shalev-Shwartz & Shai Ben-David define the largest possible VC for a neural net with weights taken from a finite family and sigmoid activation functions as $O(|E|)$[64]. In this context, the finite family refers to real numbers represented by 32 bits within a computer. This equation is considered a rough guide as it assumes the network is acting as a classifier.

Taking network 02 (presented in appendix B.3) as an example, we can calculate the number of parameters as follows:

- $convlayer = windowsize(5) * numacids(20) * numacids(20) + numbiases(20) = 2020$
- $connectedlayer = numacids(20) * maxCDRlength(28) * outputsize(112) + numbiases(112) = 62832$
- $outputlayer = outputsize(112) * outputsize(112) + numbiases(112) = 12656$
- $total = 77508$

This suggests we need at least 77,508 examples in our training set. The number of parameters is considerably smaller than other networks in use today, such as VGGNet [67] which contains a number of parameters in the order of 138 million[9].

2.4.5 Input representation

One simple way to represent the loops is to use a bit-field. Each amino acid is represented as a 1 dimensional vector, 20 bits long. All the bits are set to 0 except for a single 1, representing the amino acid in question. This can be fed directly to the neural network's input layer neurons, effectively making one neuron sensitive to one amino acid.

This method is referred to as a *sparse representation*; there is a large amount of redundancy. In some applications, such as natural language processing, words from a dictionary could be represented this way, but would results in vectors with unmanageable lengths. In such cases, a *dense representation* is often used; a method that encodes more information per bit. We consider two dense methods - BLOSUM and 5D encoding.

BLOSUM

Block Substitution Matrices (BLOSUM)[25] are used to generate a score between alignments of amino acid sequences. When aligning such sequences, there are different probabilities that a particular amino acid will be replaced by another. Rather than having a single score (such as $+1$ for a match, -1 for a mismatch), the score is related to how likely the particular substitution is.

BLOSUM are the same length as out bit-field representation, but provide more relevant information to the network.

5D encoding

There have been a number of vector representations of amino acids. One of the more recent approaches by Li and Koehl[39] derive a 3D representation of an amino acid by performing principle component analysis (PCA) and multi-dimensional scaling (MDS) on substitution matrices such as BLOSUM. They claim an increase in performance when classifying proteins into folds.

They perform several reductions in different dimensions. In this work we use their published values for 5 dimensions, listed in table C.0.1.

[9]See `http://cs231n.github.io/convolutional-networks/#case` for a breakdown of the calculation.

3-mer representation

Rather than map a single residue onto a pair of angles, we can take a particular triple of residues and map these to a single pair of angles. This has the potential to create a denser encoding, whilst considering any potential effects the local neighbourhood may have on a particular pair of angles, rather than holding a single residue responsible. This idea is discussed in Sutcliffe's thesis[69].

We can do this in a number of ways with our existing data encoding schemes.

- Concatenate the three bit-field vectors, creating a single vector 60 units long, with three positive integers instead of one.
- Concatenate the three 5D vectors into a vector of length 15.
- Perform an addition on the three 5D vectors together to form a new 5D vector.

2.4.6 Internal representation and activation functions

Internally, the torsion angles are represented as the sine and cosine of the original angle. This removes the discontinuity between 0° and 360° as well as mapping to the range of the *tanh* activation function.

Neurons produce a value based on their input weights and some *activation function*. Several functions over the weights can be used. Popular functions include the *logistic function*,*the sigmoid function*, *hyperbolic tangent* and the *Rectified linear unit* or *ReLU*.

Originally the sigmoid function was used to generate the output from a neuron. This function has largely been dropped in favour of the *ReLU* and *tanh*[23].

The hyperbolic tangent (or *tanh*) activation function is expressed as shown in equation 2.9.

$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.9}$$

Both sigmoid and tangent functions begin to saturate as their output value approaches the maximum or minimum (for sigmoid functions, the minimum is 0.0, the maximum 1.0. The tangent function is bounded from -1.0 to 1.0). The gradient at these points approaches 0, effectively vanishing. This is another example of the *vanishing gradients* problem (see section 2.1.6).

The ReLU has *"revolutionized deep learning"*[57]; it appears to be the most popular activation function, as it is easy to compute and provides a constant gradient for positive values, no matter how large or small. Equation 2.10 describes how a ReLU is computed.

$$y = max(0, w^T x + b) \tag{2.10}$$

Despite their success, ReLUs do suffer from a form of the vanishing gradient problem called *dying ReLUs*[23]. During training, some neurons begin to only output 0. This may happen, for example, if the learning rate is too high and the inputs sum to less than 0. At this point a 0 gradient is returned, effectively halting any further updates to that neuron.

This problem is addressed by a family of ReLU functions called *leaky ReLUs*. These functions do essentially the same thing: provide a small gradient for values below zero. The basic leaky ReLU is described in equation 2.11

$$LeakyReLU\alpha(z) = max(\alpha z, z) \tag{2.11}$$

Variants include the *parametric ReLU* (PReLU), *randomized leaky ReLU*(RReLU) and the *exponential linear unit*(ELU).

Neither the ELU or *tanh* functions cover the range -1.0 to 1.0 completely. ELU begins to saturate close to -1.0, whereas *tanh* will never reach -1.0 or 1.0.

For the majority of networks tested, we used the *tanh* activation function throughout. In some networks, we use ReLUs or ELUs for the internal layers, and *tanh* for the output layer, converting our internal values into the final angle range.

Whilst non-linearity is a defining feature of a neural network, using *tanh* implies that values close to 0 will change more rapidly than these nearer -1 or 1; certain angles may not be reached or quickly passed over.

We investigate *leaky ReLUs* where possible within our experiments, but the majority of networks tested use the *tanh* function.

2.4.7 Discrete classes

Many of the example networks given in the Tensorflow documentation[10] are *classifiers*. Rather than attempting to match a series of real-valued numbers to another such set, the network returns a vector of probabilities. These correspond to the network's confidence that a particular input belongs to a certain class.

Typically, these probabilities are passed through the *softmax* activation function. If there k classes and the weight for the i-th class is $w^{(i)}$ then the predicted probability for this i-th class given by the input vector x is given by equation 2.12[57].

$$P(y_{i=1/x}) = \frac{e^{w^{(i)T}x+b(i)}}{\sum_{j=1}^{k} e^{e^{(j)T}x+b^{(j)}}} \tag{2.12}$$

Softmax is a generalisation of the logistic regression, supporting multiple classes rather than just two. The probabilities returned from this function are normalised. *Softmax* is usually combined with the *cross-entropy loss* function, defined in equation 2.13[57].

$$J = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log(\hat{p}_k^{(i)}) \tag{2.13}$$

As CDRs are variable length lists of amino acids, we need a method that can support classifying each residue in a variable list. Using an LSTM based RNN within tensorflow, we can take the state at each time step, passing this state through a fully connected, dense layer, producing a classification. This approach is known as *sequence labeling*. We share the weights between each step in the RNN; as each step is computed, the same tensor of output weights is updated.

[10]available at `https://www.tensorflow.org`

2.4.8 Validation and test sets

Regardless of the choice of dataset, three subsets were created for *training*, *validation* and *testing*. In the early experiments, 80% of the available data was used for training, 10% on validation, and 10% for testing. With the larger sets we alter this to a 70%, 20%, 10% split to better gauge on-going performance for early stopping. These sets are chosen at random from the database.

2.5 Deriving errors and accuracies

In order to derive an accuracy score for the trained network, the sine and cosine angles produced from the output layer of the neural network must be combined, pairwise using the $atan2$ function, creating a set of torsion angles.

Using the NeRF algorithm, the complete backbone (i.e the nitrogen, carboxyl carbon and carbon alpha for each residue) is reconstructed, both for the predicted angles and the real angles. These loops are aligned and compared, resulting in a *root-mean-squared-deviation* (RMSD) between the $C\alpha$ atoms, in Ångströms.

As the generated loops are created within their own cartesian coordinate system using the NeRF algorithm, their orientation and final position will be different from that found in the PDB file. We use the program *PDBfit*[74][11] to superimpose the real loop onto the predicted loop and generate an RMSD. *PDBfit* implements the McLachlan fitting algorithm[50].

We refer to the loop generated by our neural network from the existing residues as the *predicted loop*, the loop reconstructed from the original torsion angles as the *intermediate loop*, and the experimentally derived loop from the PDB file as the *real loop*.

In the first stage of comparison, the predicted loop is compared against the intermediate loop. This removes any differences caused by the assumptions made by the NeRF algorithm, such as average bond lengths, a fixed angle for omega, and the use fixed, average angles between backbone atoms. The final accuracy of the network is determined by comparison with the real loops.

[11] Available at `http://www.bioinf.org.uk/software/bioptools/`

Chapter 3

Results

3.1 Organisation

The number of possible architectures and combinations of hyper-parameters available to us is vast. In practice, certain architectures previously discussed, have been shown to perform better in particular circumstances. Hyper-parameters have limits and there is some guidance in choosing the correct activation functions, error functions and other such components.

The approach taken in this work starts with a wide investigation of some of the possible architectures. We take the most promising approaches and submit them to further analysis. We briefly investigate some of the more esoteric approaches such as *sequence-labelling* and *3-mer* data representations. Finally, we consider selection of candidate loops based on results in torsion space.

The majority of the experiments were performed on a single machine with an Intel Core i5 CPU, running at 3.33GHz, with 16Gb of main memory and a GeForce GTX 760. Some of the larger networks were trained using resources provided by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service (`http://www.csd3.cam.ac.uk/`), provided by Dell EMC and Intel using Tier-2 funding from the Engineering and Physical Sciences Research Council (capital grant EP/P020259/1), and DiRAC funding from the Science and Technology Facilities Council (`www.dirac.ac.uk`).

3.2 Initial Experiments

To begin with, we looked briefly at two major architectures, convolutional networks and LSTMs, with different hyper-parameters and datasets. Each experiment was given a number, with the most promising experiments being studied in more depth.

Over 32 variations of neural networks were tested, with variations in architecture, datasets, dataset representations and hyper-parameters. The following networks were the most promising and were subjected to more rigorous testing. They are referred to by their number throughout the reminder of this thesis.

- 02 - initial convolutional neural network with *AbDb* data.

- 02a - net 02 but with restrictions on loops based on distances between endpoints
- 06 - bi-directional LSTM with *AbDb* data
- 06a - net 06 but with restrictions on loops based on distances between endpoints
- 13 - bi-directional LSTM with 5D Amino acid coding of *AbDb* data.
- 23 - bi-directional LSTM with last relevant selection, with *AbDb* data.
- 23a - net 23 but with restrictions on loops based on distances between endpoints

We describe these networks in more detail in the following sections, starting with network 02 and progressing down the list.

3.2.1 Convolutional nets

The first architecture investigated was the convolutional neural net (section 2.1.4), with one convolutional layer, one drop-out layer, and one fully-connected layer. Each loop in the training data-set is converted to the bit-field representation; each input tensor has a size of *(batch_size, max_cdr_length, 20)* with a corresponding mask. Data is presented to the network in a batch size of 5 vectors at a time. The input layer feeds into a convolutional layer with a window size of 5, and a depth of 20. This layer is passed through a drop-out layer with a 50% drop-out rate. The final layer contains 112 neurons, mapping to the maximum length of 28 multiplied by the 4 values representing the sine and cosine of ϕ and ψ. The *tanh* activation function is used throughout. The code listing for this net can be found in appendix B.3.

Each run randomly draws loops from *AbDb*, creating new training, validation and test sets. Each loop was tested to see whether or not it met the criterion of having end points within one standard deviation of the mean. Column two of table A.1.1 shows the percentage of loops from the *AbDb* set that meet this criterion, while column three shows the percentage of predicted loops that meet the same criterion. Each net was trained for 2000 epochs.

The final errors reported in appendix A.1 are these generated by the error function over the test set, described in section 2.1.14. They can be viewed as the mean average of the square of the difference between the real and predicted angles.

The main parameters under investigation were the window-size and the number of convolutional nets. Adding extra convolutional layers with smaller heights and widths, but more depth, did not improve performance and took slightly longer to train. Altering the window size from the default of 5 resulted in slightly worse performance.

Initial results were promising. Table A.1.1, in appendix A.1 shows the results after running the network on 10 separate occasions. Seven of the ten mean $C\alpha$ RMSD scores are below 2Å.

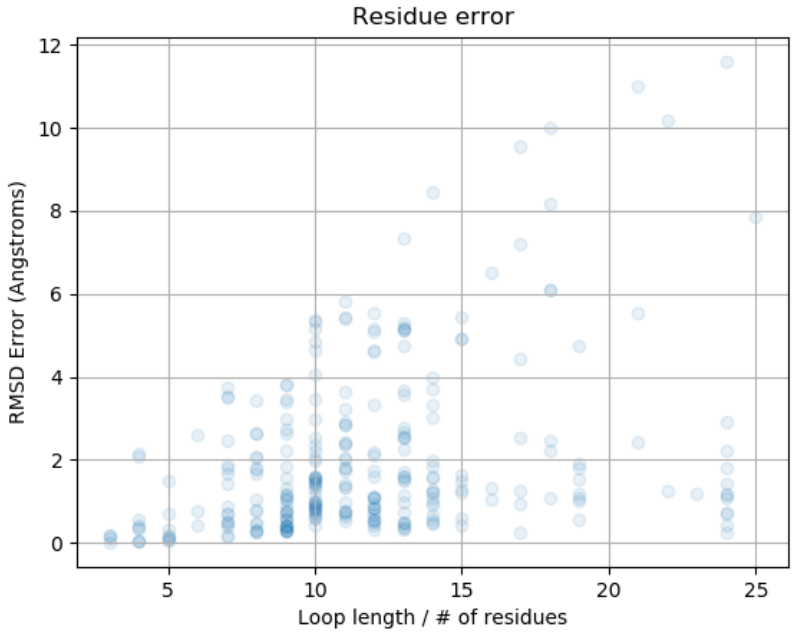


Figure 3.1: A plot of CDR-H3 length against RMSD $C\alpha$ accuracy for run #5 of network 02 (convolutional architecture, *AbDb* dataset with redundant loops using bit-field encoding) in Ångströms. The Spearman correlation is 0.358 ($pvalue = 4.2 \times 10^{-10}$). As the length of loop increases, the accuracy decreases.

Network 02a removes all loops from all datasets whose endpoints are greater than 1 standard deviation away from the mean distance. The database containing the loops also contains the distance between the first and last $C\alpha$ atoms, including the overall average and standard deviation. When creating the datasets for training, validation and test, each loop is checked against this criterion for inclusion.

Table A.2.1, in appendix A.2 lists the results, whilst figure 3.2 plots the length of the CDR against the RMSD accuracy for run #7 of network 02a. Performance is slightly improved with nine runs having sub 2Å $C\alpha$ RMSD.

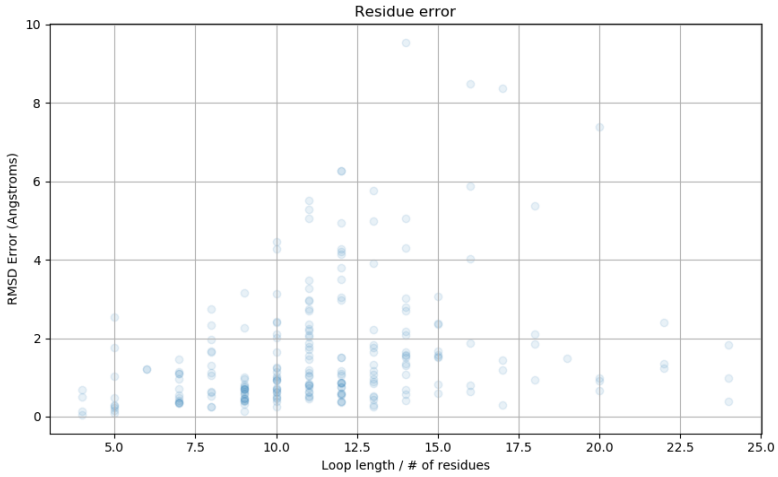


Figure 3.2: A plot of CDR-H3 length against RMSD $C\alpha$ accuracy for run #7 of network 02a (convolutional network, *AbDb* dataset with redundant loops, removing these with end-points more than one standard deviation away from the mean, using bit-field encoding) in Ångströms. The Spearman correlation is 0.372 ($pvalue = 1.2 \times 10^{-8}$). Accuracy slightly decreases as loop length increases, though several outliers can be seen.

3.2.2 Bi-directional LSTM

Given the CDR length is fixed and known at training time, we can combine both a forward pass and a backwards pass to make the network less order-dependent - more closely reflecting the problem at hand. Such networks are known as bi-directional LSTMs.

All-steps

In network 06 both the forward and backward passes are summed to create a tensor which is subsequently reshaped and attached to a final, fully connected layer. In this manner, all steps are considered. The input layer is the same as that found in network 02. This layer is fed into Tensorflow's *bidirectional_dynamic_rnn* function, along with the lengths of each loop in the batch. The output of this layer is two tensors and states, representing the current output and state of both directions of the LSTM. In this *all-steps* network, the output from all the steps of the forward and backwards passes are added together and passed into a final layer, which is the same as the one found in network 02. The code for this network is listed in appendix B.4.

The current step in an LSTM is a summation of the current information and all the preceding steps. If one performs a summation on all steps, the information presented in the very first step is counted *cdr_length* times, with each subsequent step represented ($cdr\_length - N$) times, where N is the position in the sequence. A loop 12 residues long will feature 12 LSTM steps, with the first residue appearing in all steps and the last residue appearing only once.

Alternatively, we can select only the *last-relevant* step. To do this, we select only the final state after all residues in the CDR have been considered; this is the approach taken in network 23.

Table A.3.1, in appendix A.3 shows the results for 10 runs of network 06, on the *AbDb* dataset. Each network was run for 2000 epochs, with new training, validation and test datasets randomly drawn from *AbDb*. Figure 3.3 shows a plot of RMSD error against CDR length (Spearman's correlation 0.309 $pvalue = 1.26 \times 10^{-7}$).

The performance of this network improves on network 02, with no runs having a $C\alpha$ RMSD greater than 1.8Å.

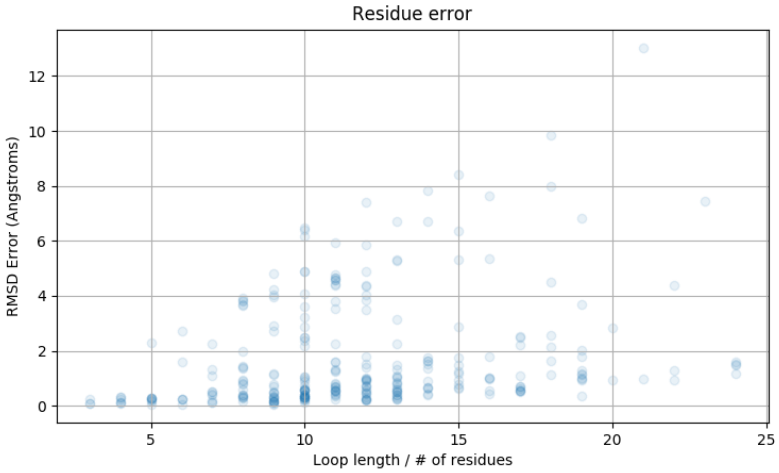


Figure 3.3: A plot of CDR-H3 length against RMSD $C\alpha$ accuracy for run #8 of network 06 (bidirectional all-steps LSTM, with *AbDb* dataset with redundant loops included, bit-field representation), in Ångströms. Accuracy decreases slightly, as loop length increases, though a number of outliers can be seen.

3.2.3 Last-relevant step

Network 23 is very similar to network 06 but rather than combine all the time steps of both passes, the last step in both the forward and backward passes are extracted and combined together. The program can be found listed in appendix B.6; the only key difference between this network and network 06 is the *last_relevant* function that takes the entire output from the LSTM layer and returns only the final step.

Table A.5.1, in appendix A.5 shows the results for 10 runs of network 23, on the AbDb dataset. Each network was run for 2000 epochs, with new training, test and validation sets, randomly drawn from AbDb.

Counting only the recurrent step in both directions shows an improvement over counting all steps. The mean RMSD is slightly improved, along with the final error scores. The results are plotted in figure 3.4 (Spearman correlation 0.372 with a p-value of 1.32^{-10}).

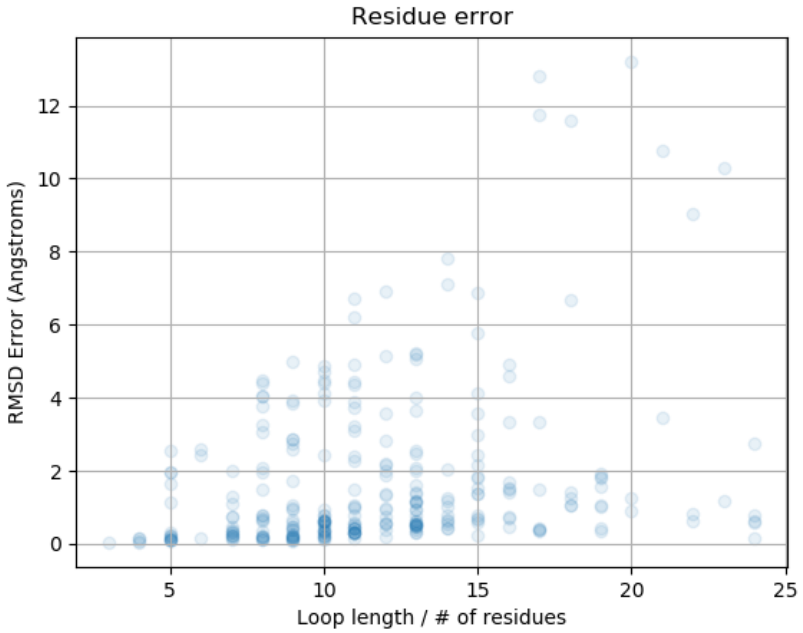


Figure 3.4: A plot of CDR-H3 length against RMSD $C\alpha$ accuracy for run #3 of network 23 (bi-directional LSTM with last relevant step, *AbDb* dataset with redundant loops included, bit-field representation), in Ångströms.

Bi-directional net with data restriction

Based on the success of network 23, we investigated the possibility that the NeRF algorithm, when set with 180.0° omega angles, would not always recreate the structure correctly (discussed in section 2.3.1) Network 23 was re-trained with only these CDR loops that both pass the endpoint test - the same restriction as network 02a - and do not contain problematic omega angles. Problematic angles are defined as these omega angles that deviate by more than 30.0° from the default 180.0°.

The results are presented in table A.6.1 in appendix A.6. Like network 02a, performance is slightly improved, with the mean $C\alpha$ RMSD scores below 1.73Å.

3.2.4 5D

Network 13 is very similar to network 06, but uses the 5D encoding described in section 2.4.5. The number of layers and the size of the LSTM units remains the same. However the input layer is changed. Rather than vectors 20 units long, our vectors are now 5 units long and contain floating point numbers as opposed to a zero or one. The LSTM and output layers remain the same as these in net 06. The code can be found in appendix B.5.

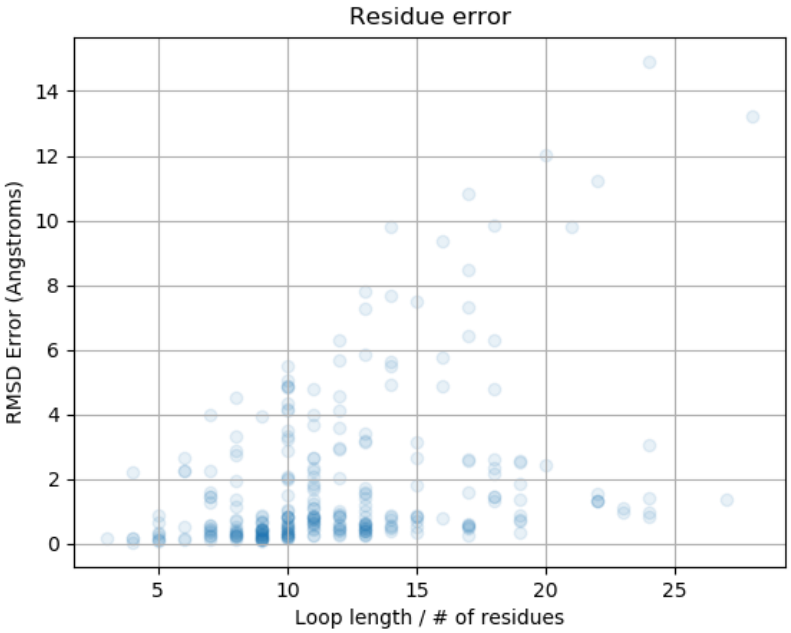


Figure 3.5: A plot of CDR-H3 length against RMSD $C\alpha$ accuracy for run #8 on network 13 (bi-directional LSTM with *AbDb* dataset with redundant loops included, 5D representation), in Ångströms (Spearman correlation 0.448 with a p-value of 3.12×10^{-15}).

Figure 3.5 and table A.4.1 (appendix A.4), show that changing the input representation results in slightly worse scores, with higher mean average RMSDs (maximum of 1.847Å) and higher standard deviations (2.459 being the largest, compared with 2.07 from network 06).

3.3 Non-redundant & LoopDB tests

In section 2.4.3, it was noted that many of the models within *AbDb* are redundant; duplicates of existing items. We performed the same set of experiments again, using only non-redundant data from *AbDb*. This results in a training dataset consisting of 1171 loops. The following tables in appendix A.7 show the results of running networks 02, 02a, 06, 06a, 13 and 23. Each network is run 10 times on a random datasets, drawn from the 1171 *AbDb* loops.

Networks 02 and 23 appear to be the best performers in terms of their average $C\alpha$ RMSD scores (3.53Å and 3.52Å respectively). The remaining networks perform very poorly in comparison to their initial scores with redundant loops included. For example, network 06 has mean $C\alpha$ RMSD scores that double when redundant data is removed: a maximum of 4.204Å compared with 1.79Å (see table A.7.3).

These results point towards the first run of networks memorising certain loops. Redundant data may appear in all three datasets, thus the network is

trained and tested on data that is extremely similar, if not identical. This results in a low validation error that is misleading.

Rather than restrict the dataset completely, another option is to allow redundant data in the training set, but have training and validation sets that contain no examples that are redundant with these in the training set.

Table A.8.1, in appendix A.8 shows the results from the run of network 02 with redundant data appearing in the training set only. The maximum mean $C\alpha$ RMSD score is 3.79Å, compared with 4.056Å in the non-redundant set - a slight improvement.

We can compare the accuracy of the network over the test set, with accuracy over the training set in order to gain some insight into whether or not the network has over-trained. Although the training error will almost always be lower, it should not be significantly lower than the validation error. Table A.8.2, in appendix A.8 shows the results of run 1, but using the training data set. The difference between the two results suggests that network 02 has still over-trained. The mean $C\alpha$ RMSD score is 1.335Å - a full 2Å lower than the lowest validation score.

Table A.8.3 shows the results from the run of network 23 with redundant data appearing in the training set only. The worst mean $C\alpha$ RMSD score is 3.886Å - slightly better than network 02.

Again, we consider a run of network 23, only with the training set instead of the test set. Table A.8.4 shows the result of re-running the net 1 evaluation but with the training set. Again, the network appears to have over-trained, with a mean $C\alpha$ RMSD score of 0.957Å.

It is worth considering the final loops themselves, in Cartesian space, and how the RMSD $C\alpha$ score relates to the final structure. Figure 3.6 shows an intermediate and predicted loop based on the PDB 3F12_2. Both loops are incorrect due to a glycine with an omega angle of $-61.8°$ within the actual structure. Only some residues are well aligned.

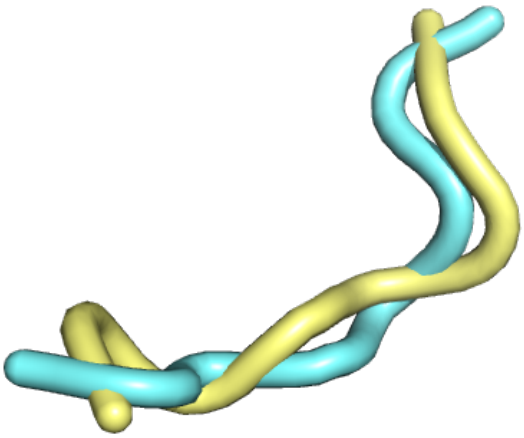


Figure 3.6: An image of 3F12_2, predicted in yellow and intermediate in cyan. The intermediate loop has been aligned, resulting in an RMSD score of 2.066Å. While the right hand portion of the loop is well aligned, the left-hand endpoint is clearly impossible.

One of the better results is shown in figure 3.7, for the PDB file 2H1P_1. Both local and global structure have been reproduced correctly. This loop contains no atypical omega angles.

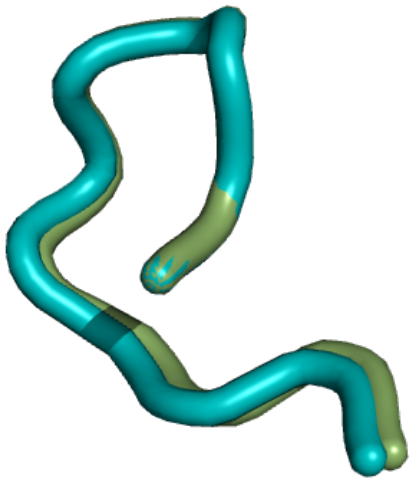


Figure 3.7: An image of 2H1P_1, predicted in green and intermediate in blue. The RMSD score is 0.261Å

One of the worst performing models is 4FQL_1, shown in figure 3.8. The structure bears very little resemblance to the original with a RMSD score of 11.185Å. Again, the intermediate loop appears straight with some local variation that superficially appears similar to the local variation on the actual loop, but the global structure is completely incorrect. This model has no ω angles that differ from 180° or −180° by more than 18°.

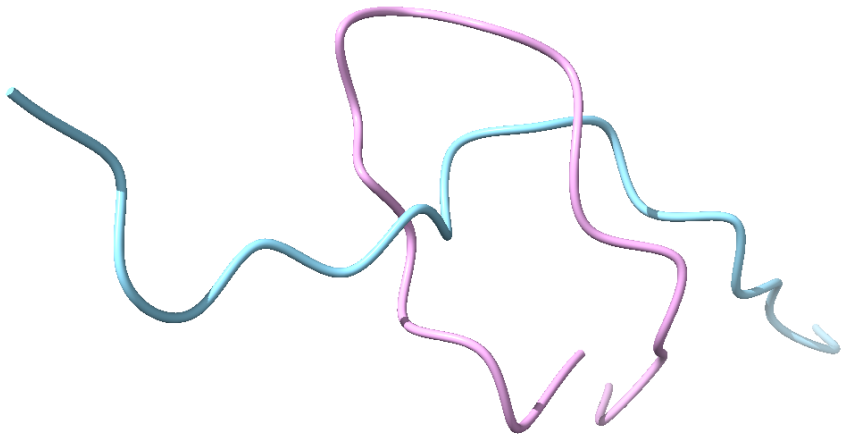


Figure 3.8: An image of 4FQL_1, predicted in purple and intermediate in blue. The RMSD score is 11.185Å

To combat the over-fitting, the best performing network 23, was retrained using a combination of non-redundant data from both *AbDb* and *LoopDB* combined, resulting in a dataset of 12248 loops in size. The input data was formatted using the bit-field method (section 2.4.5). The program itself remains the same. Table 3.1 shows the final result.

Set	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
Test set	6.181	0.011	29.24	5.423
Training set	5.09	0.08	29.08	4.69

Table 3.1: Results of independent runs of neural network 23 with non-redundant data in all sets. Scores on both the test set and training set are given.

Performance decreases compared with just *AbDb* non-redundant sets, with the overall accuracy being lower. However, the difference between the average RMSD on the training and test sets is small when compared with the same network running on the *AbDb* set only. This suggests that the larger dataset prevents over-training at the cost of some accuracy.

3.4 Sequence labelling

As described in section 2.1.7, it is possible to represent a particular pair of angles with a single label. In the following experiment we divide 360° into 10° increments, resulting in $36\phi * 36\psi = 1296$ combinations or possible labels.

This network consists of a 3 layer LSTM network, with sizes of 256, 128 and 64 respectively. The final layer feeds into a dense layer with 1296 outputs representing the probabilities that each label corresponds to the residue at the current time step. The weights and biases comprising this layer are shared

across every LSTM step. We train this network on *AbDb*, with redundant data appearing in the training set only.

We employ an early stopping technique, where training is halted once the error stops decreasing. The input data is encoded using both the bit-field and the 5D representation as described in section 2.4.5.

The results are shown in tables A.9.1, A.9.2 and figure 3.9. Comparing against the real loops shows slightly worse performance than intermediate loops. Again, there is a considerable difference in accuracy between the test and training sets, suggesting that this network has over-fitted.

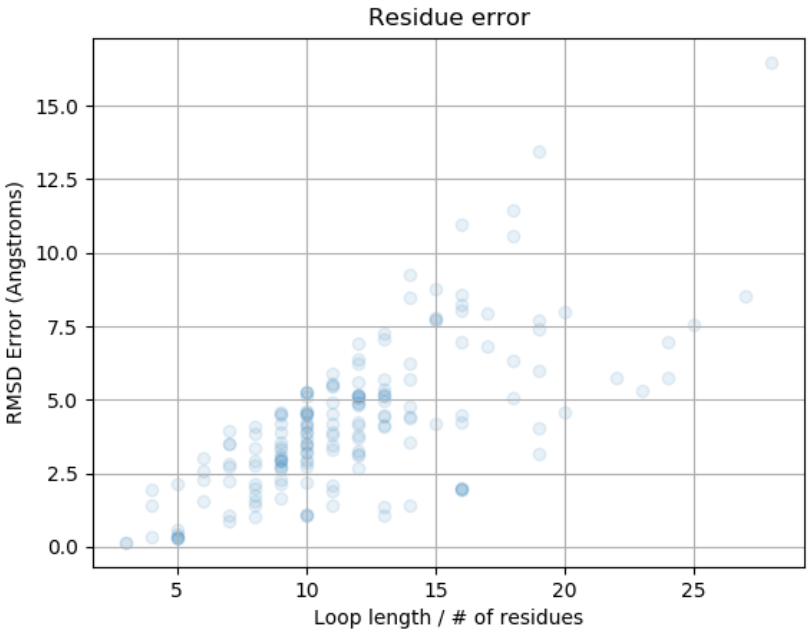


Figure 3.9: A plot of RMSD error against the length of residue for our labelling network, trained on the *AbDb* dataset. The error is derived by comparing the $C\alpha$ RMSD between predicted and real loops.

We have suggested that many of the networks memorise their inputs, rather than learn underlying patterns or rules, due to the poor performance on the validation and test sets. When training this particular network, we noticed the following irregularities in the training error. Figure 3.10 shows a repeating pattern of decreasing error, followed by a sharp increase back to previous levels. This *bounce* is suggestive of memorisation. As the network quickly over-fits, the error rate is reduced, but when a new epoch starts, the network must quickly adjust to the new data, resulting in a high error rate. It is possible to see this effect as the training set is rather small.

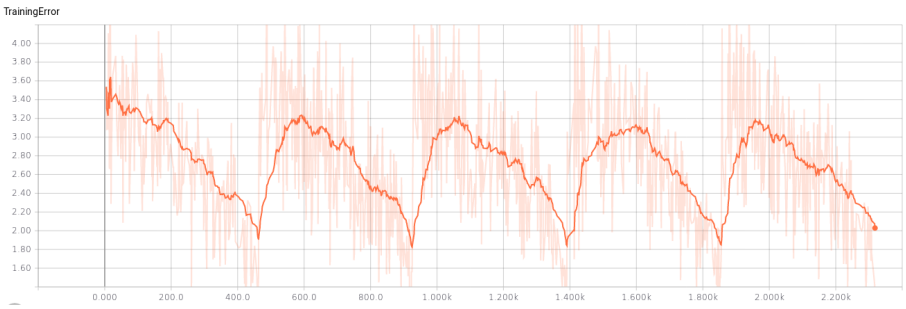


Figure 3.10: A plot of training error against step number for our sequence labelling network.

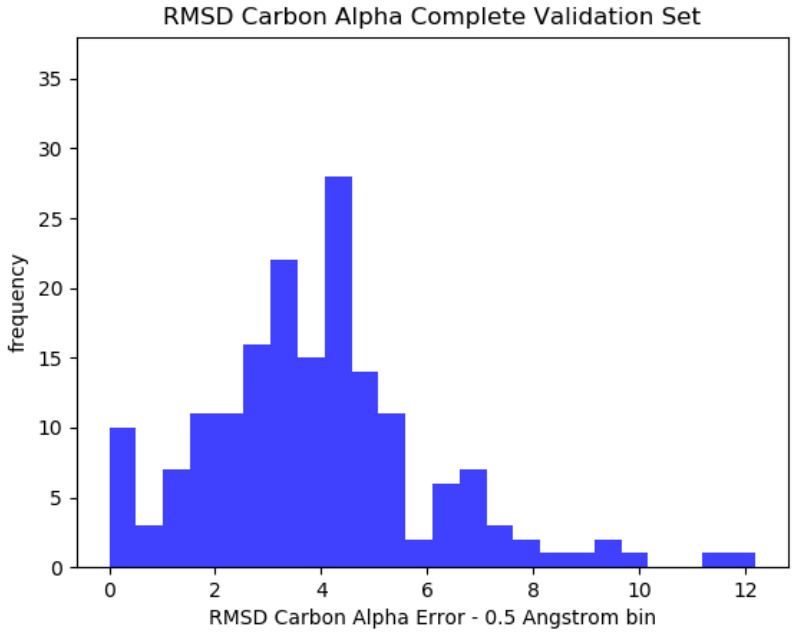


Figure 3.11: A histogram showing the frequency of particular sizes of RMSD $C\alpha$ errors for the labelling network, trained on AbDb.

Tables A.9.1 and A.9.2 shows the accuracy of this network on just the *AbDb* set alone. These scores are based on these loops from *AbDb* that were not included in training or validation sets. There is some improvement in the mean and maximum RMSD scores in both intermediate and real loops. The code listing for the labelling network can be found in appendix B.8.

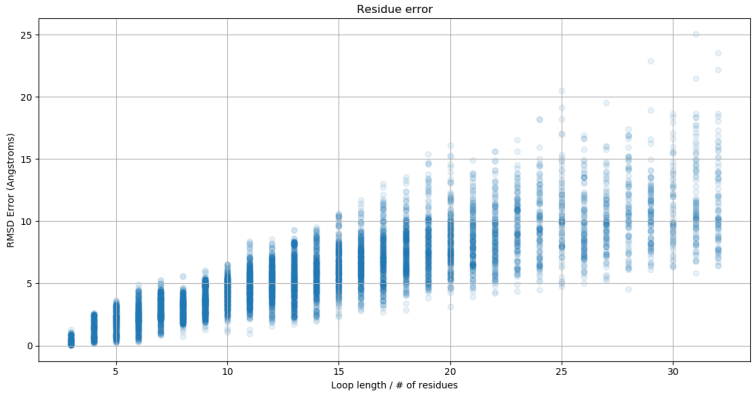


Figure 3.12: A plot of RMSD error against the length of residue for our labelling network, trained on non-redundant data from *AbDb* and *LoopDB* datasets. The loops are encoded using the bit-field approach. The error is derived by comparing the $C\alpha$ RMSD between predicted and real loops.

The size of the dataset (and the number of epochs in training as a direct result of dataset size) could be the cause of memorisation, as the network sees the same loops a large number of times. We therefore run the same network on the much larger *LoopDB* dataset, combined with *AbDb*. Figure 3.12 and table A.9.3 shows the results of this network when run on a training dataset with no redundant models. The final training set contains 51993 loops. The data is encoded with the bit-field encoding described in section 2.4.5.

This network achieves a mean $C\alpha$ RMSD score of 5.751Å - a small improvement over the 6.181Å achieved by our final network 23.

3.5 3-mer networks

We tested two kinds of 3-mer networks. Each uses the sequence labelling method of the previous section to generate a set of discrete angles. The network only differs in its input layer to accommodate the longer input vectors.

The first method uses the 5D approach, resulting in a vector of 15 values per residue. The second network concatenates 3 bit-field vectors, creating a final vector 60 values long. Table A.10.1 shows the results of both networks.

We briefly considered the 3-mer sequence where three, 5D vectors are added, to create a final vector. This particular approach resulted in much poorer performance and was discounted.

3.6 Further analyses

We decided to focus on two of the networks for further analysis: network 23 with a training dataset built from the non-redundant elements of *AbDb* and *LoopDB*, and the labelling network with the same dataset and the bit-field representation. Both score around 6Å mean $C\alpha$ RMSD. We improve on the code in network 23, shown in appendix B.7. This network is trained for a maximum of 2000 epochs,

stopping early when the validation error ceases to improve, recording the best network found up until that point.

3.6.1 Versus AMA-II

The second Antibody Modelling Assessment (AMA-II)[5] lists the performance of the current *state-of-the-art* approaches to antibody loop modelling. The following 11 structures are predicted and compared:

- 4MA3
- 4KUZ
- 4KQ3
- 4KQ4
- 4M6M
- 4M6O
- 4MAU
- 4M7K
- 4KMT
- 4M61
- 4M43

Models with these names appear in the *AbDb* data set. The AMA-II uses different numbering schemes and defines CDR-H3 somewhat differently than Martin, with an additional 2 residues at the beginning of the loop, and one at the end. In addition, some of the residues (aside from these additional residues at each end) are different in our dataset.

Seven different programs are tested in this study. Table A.11.1 lists their performance compared with our final network results. Our network performs worse on the majority of scores, with some exceptions such as 4MA3. While the scores for our final net are presented directly with these from AMA-II it is important to note that these loops are not identical so a true, direct comparison is not possible. However figure 3.13 highlights the problems with the predictions. 4MA3 is quite well aligned, as its RMSD score would suggest. However 4M6O shows a model that should look like a hair-pin, however the predicted loop appears not to be a loop at all. Where a sharp turn should appear near the middle, there is none. 4M60 contains no atypical omega angles. 4M43 appears to be a mixture of accurate and poor predictions throughout. Despite the score of 3.431Å, the only valid loop with realistic endpoints is 4MA3.

Figure 3.13: Three examples from the AMA-II models, 4M6O (in pink), 4M43 (in blue) and 4MA3 (in orange). Each pair is aligned with *PDBfit*.

3.6.2 Particular acids

We break down the errors in ϕ and ψ angles, by individual residue and the 3-mer occurrences within our final network's test set. Figure 3.14 shows the distribution of amino acids within the test set, with glycine and valine being the most prevalent.

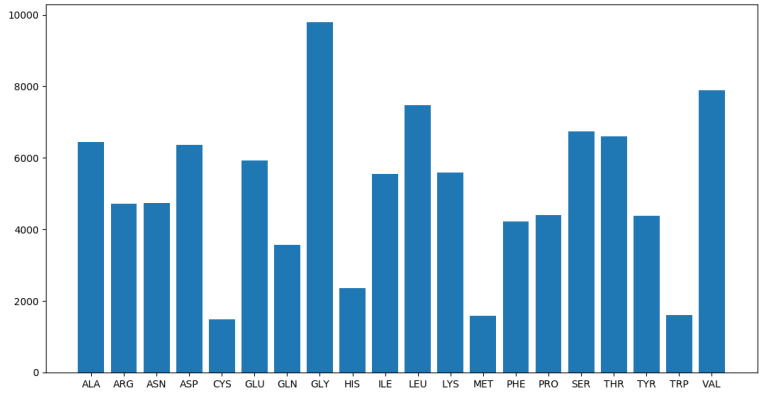


Figure 3.14: The number of occurrences of each amino acid within the test set of our final network.

We plot the mean and median averages of the error in degrees for each amino acid in figures 3.15 and 3.16. Both of these figures show very little difference in distribution, mean or median scores by amino acid. One might expect to see greater variability in the more frequent acids but this does not appear to be the case.

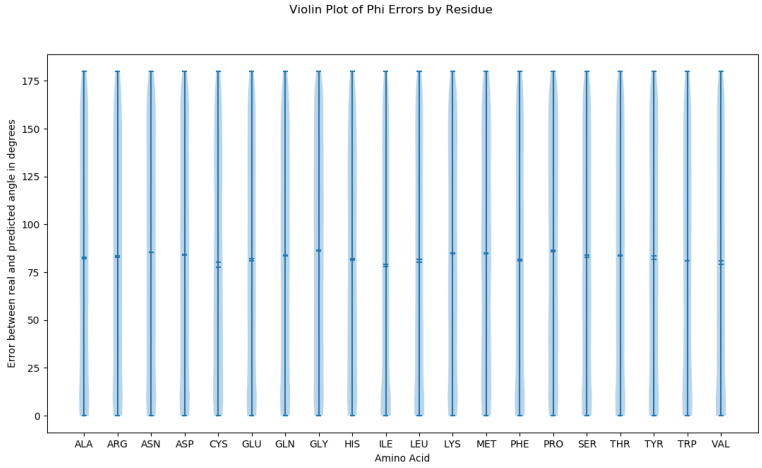


Figure 3.15: Violin plot of the errors in the ϕ angle, organised by amino acid, in our finalnetwork.

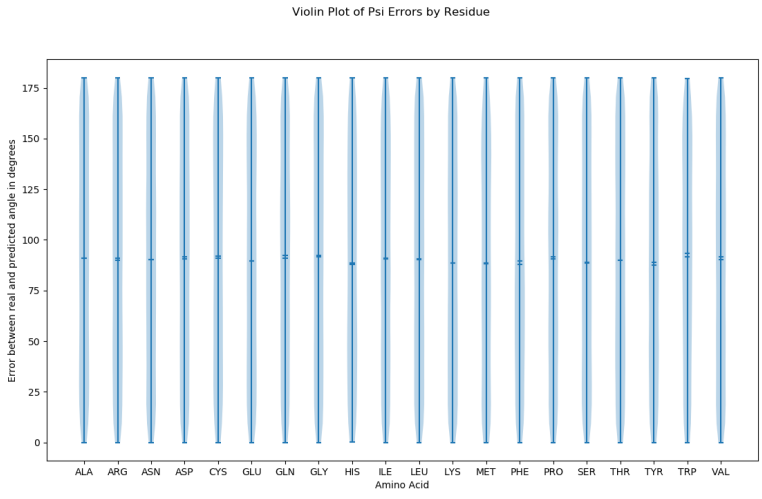


Figure 3.16: Violin plot of the errors in the ψ angle, organised by amino acid, in our finalnetwork.

3.6.3 Ramachandran plots

We plot the ϕ and ψ angles for both the predicted and real loops (figure 3.17) from the final network test set. The plot of the real values roughly follows the

Ramachandran plot for all acids, though many angles appear in areas that are normally not populated.

The predicted plot does not completely conform to the ideal Ramachandran plot, with areas towards the centre being too populated. The bottom left and right of the plot are extremely under-populated, with the top left area also under-populated.

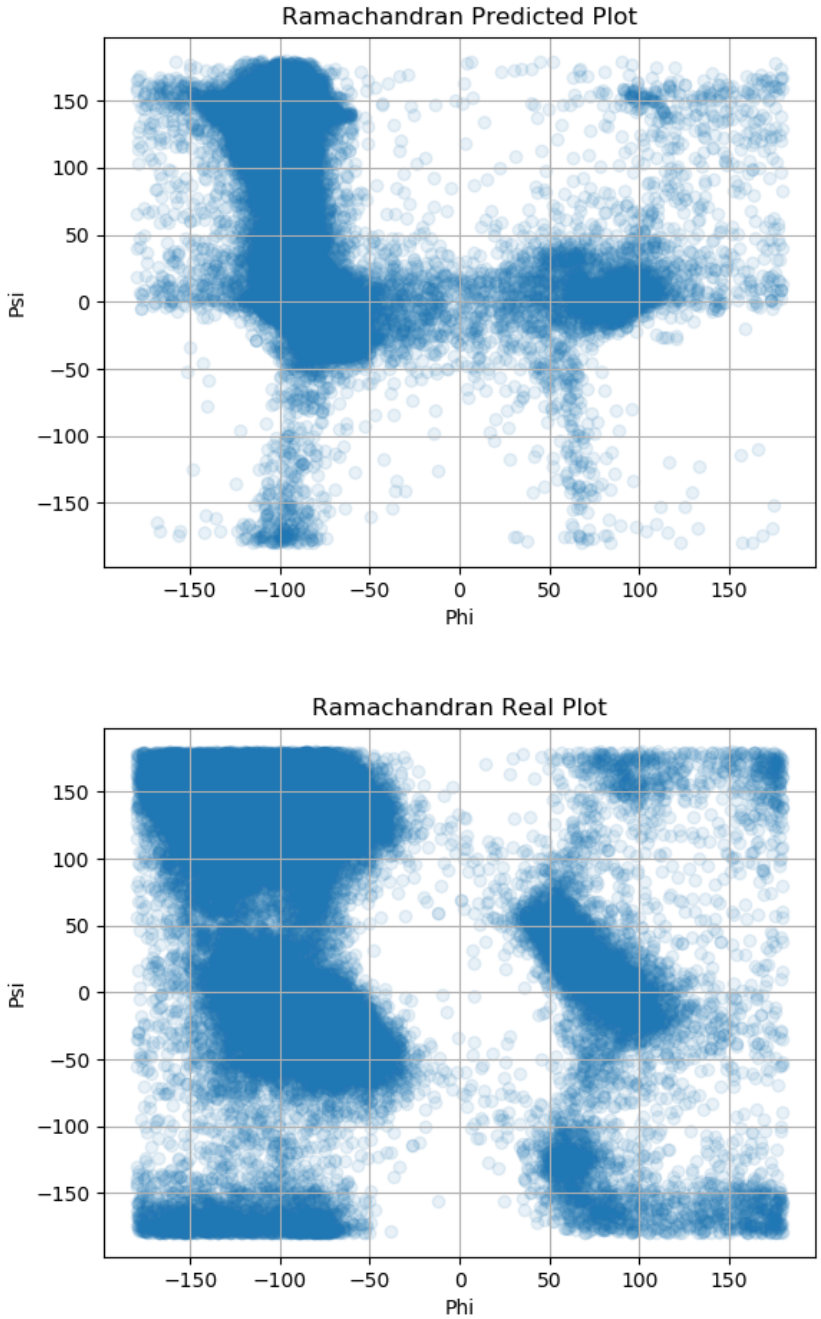


Figure 3.17: Phi and psi angles (in degrees) plotted against each other for the predicted loops (above) and real loops(below) from the test set of our final network.

The plot of the real loop angles contains several points in areas that should

be sparsely populated which suggests there may be errors or poor quality loops within the dataset. In the predicted plot, there are areas that should be populated that have not been explored by the neural network, such as the area around $-160°\psi, -160°\phi$.

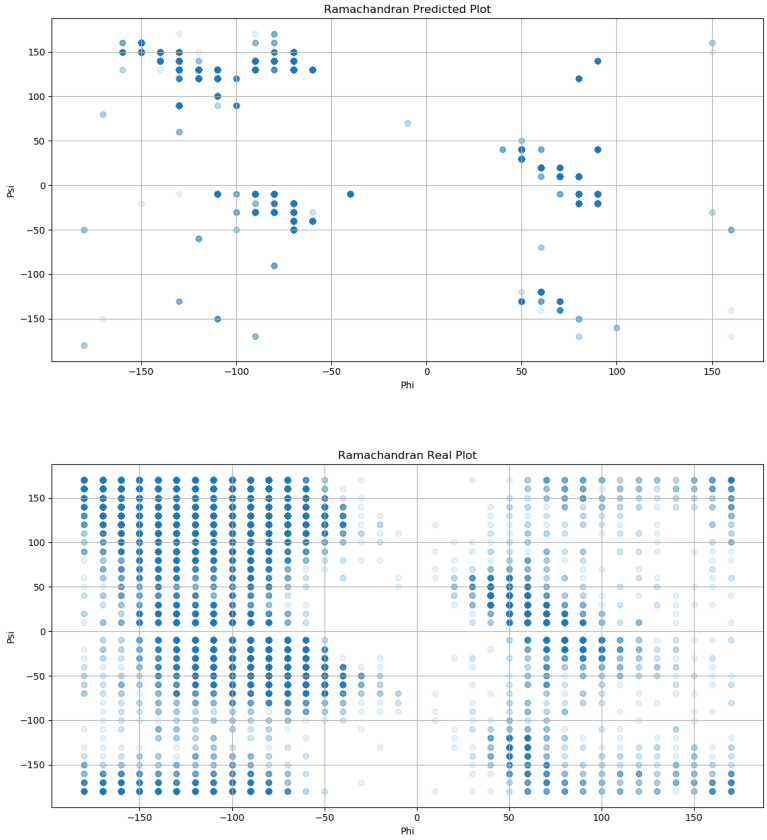


Figure 3.18: Phi and psi angles (in degrees) plotted against each other for the predicted loops (above) and real loops(below) from the test set of the labelling network. The angles are discretised into 10° wide categories.

Figure 3.18 plots the angles of the real and predicted loops within the test set of the labelling network. The angles have been discretised into 10° increments. A similar pattern emerges where large areas of the plot that are populated in the real set are not populated in the predicted results.

3.6.4 Endpoint analysis and error location

When attempting to build antibody loops it is important that the endpoints be within a specific distance of each other so that said loop can be grafted onto a framework. We compare the loop lengths against the RMSD $C\alpha$ errors of the entire test set (in figure 3.19) with a subset where all the loops with bad endpoint distances removed. We take the Cartesian distance between the

first and last atom in the backbone loop and compare with the mean average distance distance, derived from the *AbDb* dataset. If the distance is more than two standard deviations away from this mean, we reject it. The results are plotted in figure 3.20. In this second graph we can see that longer loops begin to disappear from the results entirely, and that the average error remains below 2Å until loops reach 11 residues or more in length.

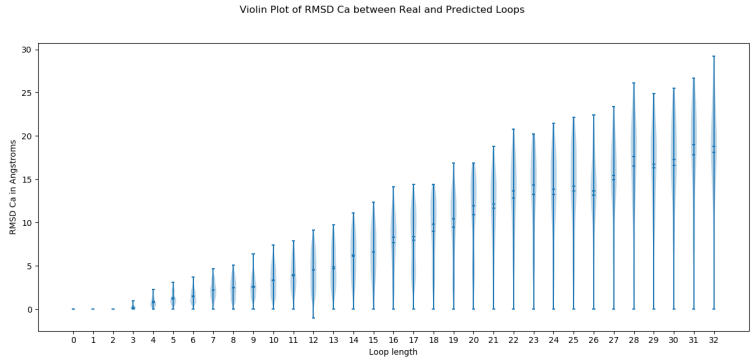


Figure 3.19: Violin plot of loop length against RMSD between $C\alpha$ atoms for a single run of our final LSTM based network, using loops from both AbDb and LoopDB. Mean and median averages are shown.

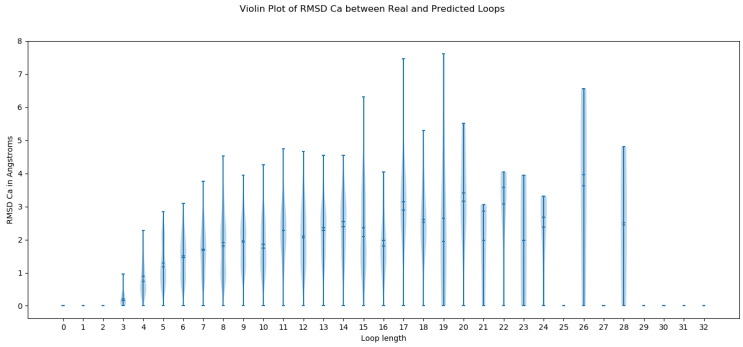


Figure 3.20: Violin plot of loop length against RMSD between $C\alpha$ atoms, where loops with a endpoint distance greater than 2 standard deviations from the mean are removed. Mean and median averages are shown.

Through-out the experiments, visualisations were made of the predicted loops. Occasionally, we would notice a loop that should appear similar to a hair-pin would be *'straightened out'*. This lead to a suspicion that the worst error would appear towards the centre of the loop.

Taking the mean position of the worst error in torsion space across all the test loops supports this position. Normalising the position we arrive at a mean of 0.464 with a standard deviation of 0.281. If we restrict the results to loops of length 8 or more, this value increases to 0.482 with a standard deviation of 0.28. Taking each loop length individually, the position does not change significantly,

and neither does the deviation. As the standard deviation is quite large, there appears to be little or no pattern to where the worst error occurs.

The mean and median of the errors within each loop do not differ considerably from each other, whilst the range of the errors across all rejected loops remains between 32° and 69°. This suggests that rejected endpoint loops are not rejected purely on the basis of a single large error, towards the centre of the loop.

It is possible that one particularly bad angle in an otherwise good prediction set can result in an unusable structure in Cartesian space, whereas the error in torsion space may be comparatively small. Figure 3.21 shows 1RZI_3 intermediate and predicted. A single atypical omega angle on the second residue causes a large change in structure.

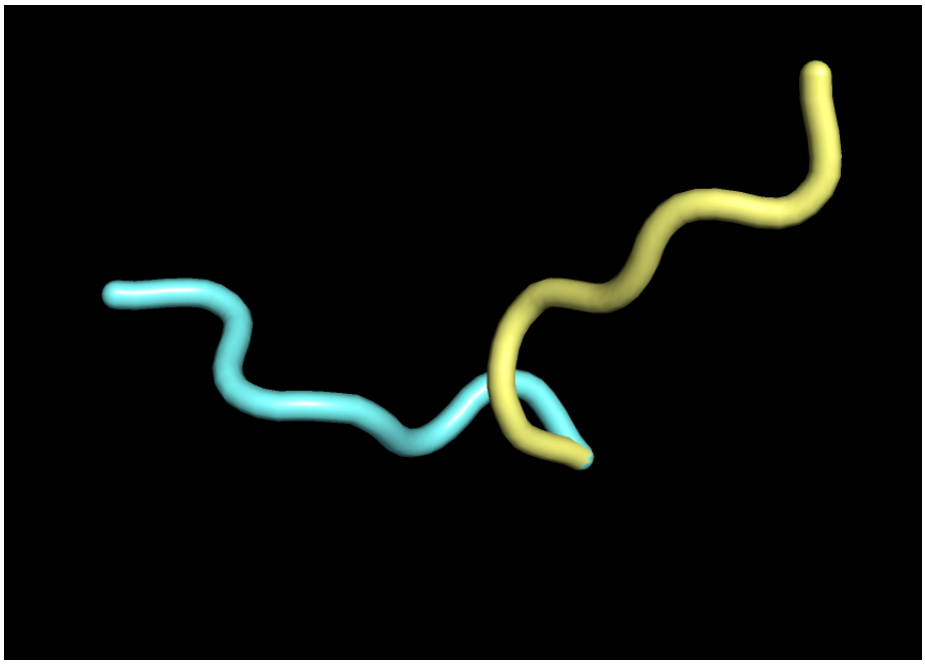


Figure 3.21: 1RZI_3 intermediate loop in yellow and the predicted loop in blue.

Removing these structures with atypical omega values does not affect the overall performance in a significant way however, as their prevalence within the combined *AbDb* and *LoopDB* set is relatively low. More common is a structure where a small number of ϕ and ψ angles are incorrect, but the overall structure retains a similar shape. Figure 3.22 shows one such example, 1TZI_1.

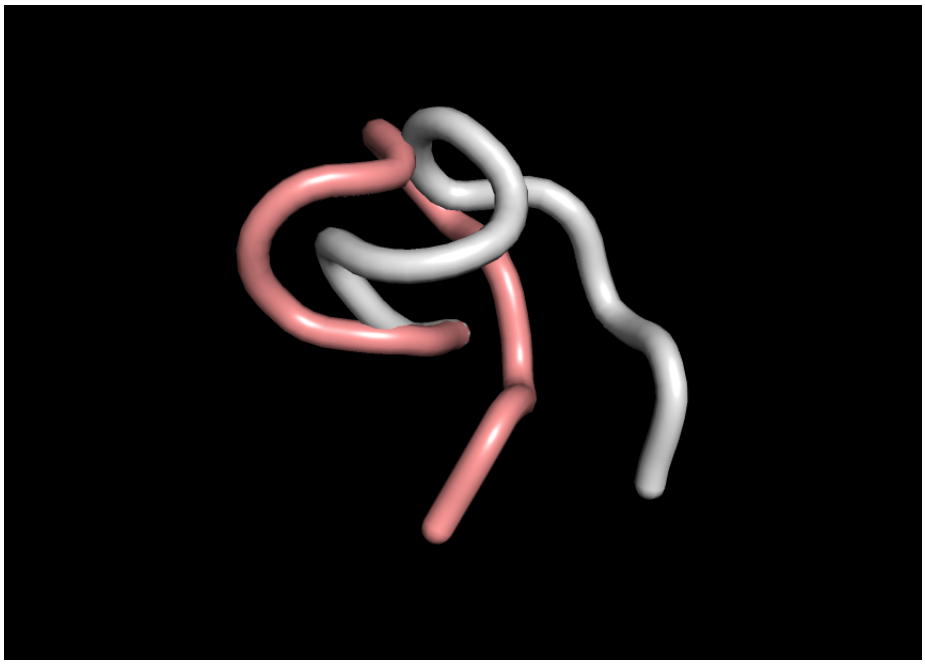


Figure 3.22: 1TZI_1 intermediate loop in grey and the predicted loop in pink.

3.6.5 Altering the mask position

Until this point, our experiments have used the same masking function for loops shorter than the maximum length (either 28 or 32 depending on the dataset). The output vector representing the neurons in the output layer is multiplied against a vector with zeros at the end, representing these unfilled residues. The cost function and the input data are also masked in the same way.

In section 2.1.11, Reczko *et al*[60] place their equivalent of our mask in the middle of the output vector, effectively splitting the loop in two, with one half represented by the output neurons at one end, and the other half represented by neurons at the opposing end of the output vector. We take our final network and copy this approach, masking the centre of the output vector, adjusting the cost function and input vector accordingly.

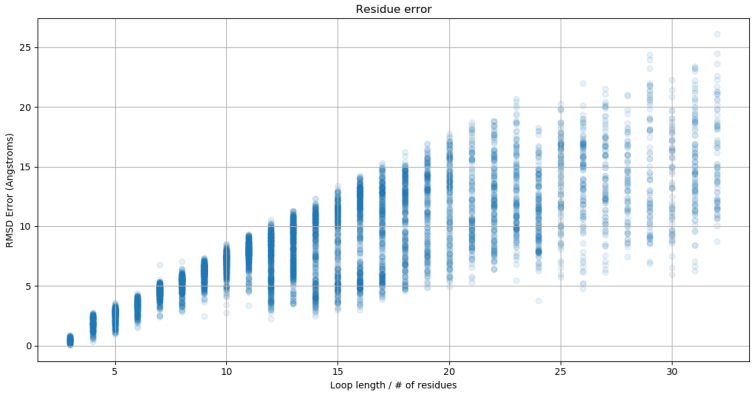


Figure 3.23: A plot of CDR-H3 lengths against RMSD $C\alpha$ accuracy of our final network (bi-directional last step LSTM, with *AbDb* and *LoopDB* combined dataset with no redundant loops, bit-field representation), in Ångströms. The mask is moved to the centre of the loop.

RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
7.968	0.028	26.1	4.242

Table 3.2: The results of running our final network, moving the mask to the middle of the loop. We use the combination of *AbDb* and *LoopDB* with no redundant data.

Figure 3.23 and table 3.2 show a lower overall standard deviation between the predicted and real loops, whilst the overall mean accuracy is lower. The comparison between the standard deviation in this network and the original final network is most pronounced in these loops of length 11 or less.

3.7 Selection by RMSD score in torsion space

Rather than rebuild the loops in Cartesian space, we considered whether or not the best performing network could make an accurate suggestion of which loop within the test set best matches the list of amino acids presented. The algorithm proceeds as follows:

- Train network and generate angle predictions as before.
- For every prediction, generate RMSDs in *torsion space* against every other loop in the test set of the same length.
- Take the lowest scoring loop, convert it to cartesian space and generate an RMSD over the $C\alpha$ positions against the original loop.

We can then see whether or not the network chose a good candidate loop from a large set, given an input sequence. For this experiment, we choose our final LSTM based network, using the non-redundant *LoopDB* and *AbDB*

combined, with a test set size of 7660 loops. The torsion angles are generated as before but we do not recreate the structure. Instead we compare the predicted set of torsion angles against all models in the test set, selecting the loops with the lowest mean RMSD score in torsion space. We then compare this chosen structure with the actual structure the input residues were taken from, deriving a final RMSD $C\alpha$ score in Cartesian space.

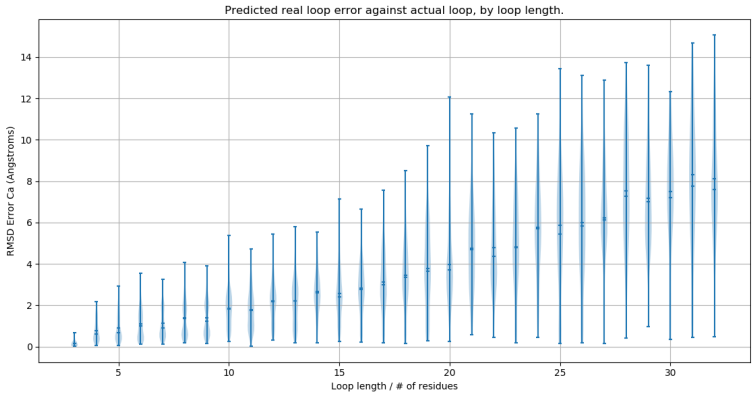


Figure 3.24: Violin plot of loop length against RMSD $C\alpha$ error between real and selected loops in Cartesian space.

Figure 3.24 shows the results of this experiment. The mean and median averages remain below 2Å until the length of the loop reaches 12. This suggests that selecting from a large list of possible candidates results in higher accuracy than generating the loops directly, and comparing using RMSD $C\alpha$ scores - for these loops of length 12 and below.

Chapter 4

Discussion

4.1 Conclusion

The results do not support rejecting the null hypothesis; we have not improved on the the accuracy of modelling CDR-H3 loop using neural networks, over that already provided by other methods. However, the modelling of certain loops and the selection of a well-fitting loop from a large set both appear to be quite accurate.

4.1.1 Indifference to architecture

The most glaring problem appears to be the indifference to changing architecture. Although some improvements were made when moving from convolutional networks to LSTMs, changing the internal architectures (such as adding more layers for example) appeared not to change the results by any significant amount.

Altering the window size in the convolutional networks did not change the results, unless the window was made extremely small (1 or 2 units long) or extremely large (around 10 and above). Adding more convolutional layers or making each layer deeper resulted only in a longer training time.

When considering the LSTM networks, changing the size of the LSTM units seemed to make little difference. Sizes of units exceeding 256 in size proved difficult to train, due to a lack of computer memory. Replacing the addition function with either a concatenation operation or average function made no difference to the final outcomes. Adding more layers resulted in slightly worse performance. GRU cells were quicker to train but the results were not significantly different to these networks utilising standard LSTM cells.

The sequence labelling network appeared to show the best performance on the combined *LoopDB* and *AbDb* non-redundant set. However, this network was tested and trained only once due to time and resource constraints so no firm conclusion can be drawn at this point.

In the later networks when redundant data was removed, optimizers such as *Adagrad*, and in particular *Adam*, performed better than simple gradient descent alone. This is in-line with the current literature [23].

4.1.2 Exploring the Ramachandran plot

Analysis of both the final network 23 and the labelling network show that certain combinations of ϕ and ψ angles that appear in the real loops are not seen in the final predictions. Certain areas of the Ramachandran plot do not appear to have been explored. Whether these areas were considered and rejected, or not considered at all remains unclear.

An area not considered by this work is *weight initialisation*. Tensorflow examples typically use a random weight with a normal distribution centred on zero, with a standard deviation of 0.1; this is the approach we adopt throughout. Engelbrecht [16] describes several approaches, but states that our adopted method is one of the most successful. Nevertheless, it is not entirely certain that such an approach is the best in our particular case; a poorly chosen weight initialisation strategy will affect convergence and may result in zero derivatives. It is conceivable that our weight initialisation may have resulted in certain angle combinations never being explored.

4.1.3 Altering the mask position

During the initial experiments, the mask for the input and output layers consisted of zeros at the end. These neurons towards the end of the output and input layers would receive less training as a result. The neurons towards the centre of these layers would effectively represent both the middle and end of the loops passed in, depending on the lengths of the loops.

Changing the mask so the zeros appear in the centre forces the neurons at the beginning of the layers to represent the starting point of the loop, and the neurons at the end of the layers to represent the end point of the loop, better reflecting the underlying structure. However, this approach resulted in slightly worse performance overall, despite a reduction in the standard deviation.

4.1.4 Memorisation and over-training

Over-training is a serious risk with small data-sets. We encountered this with the first set of networks trained on AbDb only. Regularisation techniques, such as drop-out, may have helped to ameliorate this risk, but it remains unclear how effective such techniques may have been.

In later networks, trained on the *LoopDB* and *AbDb* networks combined, over-training became less of a problem; the differences between accuracies on the training and test sets were much smaller.

4.1.5 Datasets

Even with the addition of *LoopDB* to the *AbDb* set, an argument could be made that it is still too small. With the removal of redundant and incorrect data, the final set was of the order of 50,000 loops in size. The distribution of these loops is not uniform, with the longest loops being under-represented.

It is a widely accepted practice in deep learning for image recognition tasks, to generate extra data from an existing set. Typically, this involves rotations, scaling, colour swapping (in some cases) and any other operation that does not alter the information the network requires to learn. Indeed, adding rotated

versions of the same image, for example, leads to the desired property of a neural network becoming rotation invariant when attempting to classify images.

Considering such invariances in the antibody loop data (if they exist at all) would be one potential step towards both learning new information and increasing the number of examples available for training, further reducing the over-training problem.

Restricting the range of loops in the dataset to a specific range (8 to 21 residues in length specifically) did not appear to alter the accuracy of our final network in any significant way.

In order to keep the size of the dataset as large as possible, no restrictions based upon *quality* were applied. For example, we did not reject any low-resolution loops, or loops with large *B-Factors*. The Ramachandran plots in section 3.6.3 show some uncommon angle combinations which suggest poor data. Further restrictions in addition to the endpoint restrictions might increase accuracy, though the dataset would have to be enlarged in some way to compensate.

4.1.6 Data representation

Changing the representation of the input appeared to make little difference on the final results. In some cases, it appeared to make the final accuracy worse. This was unexpected and appears to go against the machine-learning consensus that dense representations perform better [82][39]. This suggests the various networks are operating in a way different to that expected.

The 3-mer representations performed worse than both the 5D and bit-field, single residue approaches. Adding the 5D representation of the 3-mer into a new 5D vector also performed poorly.

4.1.7 Time and budget contraints

Although networks with datasets derived from *AbDb* alone can be trained in a matter of hours, larger networks trained on *LoopDB* take several days to complete their run. Typically, each net will consume the entire memory resources of a particular system (in our case 2 or 4GB of GPU memory depending on size and number of layers), limiting the number of models that can be trained in parallel. With a large number of *hyper-parameters* to explore, several architectures to investigate and various filters to apply to the data, the possible number of approaches remains quite large. Running each network repeatedly is time and cost prohibitive.

4.1.8 With respect to the aims of the project

Looking back at section 1.3.4 we have evaluated a variety of neural network architectures, experimented with different data representations and taken steps towards scoring potential solutions.

Assessing whether a loop, built by other means, is reasonable has yet to be attempted.

4.2 Future

There are several avenues one can pursue in the quest to improve the modelling of CDR-H3.

4.2.1 Cost functions

The cost function is perhaps the most important feature of a neural network. In our approach, we evaluated only two such functions, both of which consider each residue individually, whether it be a pair of angles or a single classification (in the former case, each angle is treated independently).

Because the NeRF algorithm builds in a single direction (from the nitrogen onwards) errors tend to compound, resulting in a final endpoint too distant from the desired endpoint. Although some accuracy was obtained by using a bi-directional LSTM, as opposed to a single direction, the cost function itself only considers local structure.

We briefly considered two approaches to global structure improvement: using the NeRF algorithm as a cost function, and inverse kinematics as a refinement step.

Inverse Kinematics

Inverse kinematics is used in a variety of fields - from robotics to animation. It is also used in Rosetta to help close the loop so it reaches the required endpoint[68].

In this algorithm, a set of constrained, rigid bodies form a kinematic chain where each body has certain degrees of freedom of rotation. A common example is the human arm, starting with the shoulder joint, progressing down the arm to the wrist. If this system is fixed at the shoulder and there is a target to reach, t, it is possible to construct a function, f, that calculates the position of the hand in space and its distance from t in terms of the angles of the joints. This is known as *forward kinematics*. It is often possible to differentiate f, resulting in a gradient for each angle, which can be minimised by a gradient descent algorithm. This is inverse kinematics.

Refining an existing result with inverse kinematics has been shown to improve CDR-H3 modelling in Rosetta[68]. It is possible to write a refinement algorithm in Tensorflow that can be automatically differentiated.

There is a problem with this approach. Inverse kinematics has a pattern in its solution. Rather than greatly alter a single angle to reach an end-point, it will alter all angles by a smaller amount. This may not always be the correct solution.

NeRF as cost function

We can reconstruct our prediction and our target loops from their angles using NeRF; the Cartesian difference between predicted and real carboxyl endpoint provides the error. Gradients can be derived automatically from the NeRF algorithm by Tensorflow resulting in a working cost function.

One problem with this approach is combining local structure with the global structure. Simply using the endpoint distance as a global function would result in a lack of accuracy in the local structure. Both approaches would need to be

combined in order to be effective. Our end-point analysis suggests that several loops are still poorly predicted, even if their endpoints are within an acceptable distance of the target endpoints.

Secondly, the NeRF algorithm is very expensive in terms of computation time. There are many steps that require differentiation by Tensorflow, to the point where such a cost function is prohibitive with resources available to us at the time.

Other approaches

Using an energy minimisation function as a cost function could potentially remove these predictions that are energetically unfavourable or impossible. None of the neural networks explored in this thesis use any form of force-field or energy calculations within their cost functions.

Restricting the cost function to certain areas of the Ramachandran plot was considered but rejected in the hope that the neural network would create predictions that resulted in a Ramachandran plot rather than enforce this restriction explicitly (it was hoped the network would *learn* the underlying reason for such a plot). However, it is possible to enforce this constraint by approximating the plot using a set of equations that define an *error surface*, upon which an optimizer can derive gradients. However, this would result in another hyper-parameter - the scaling of this second term within the existing cost function, in relation to the primary error of RMSD between predicted and existing angles.

4.2.2 Combined Convolutional and LSTM networks

Quang and Xie[59] present a network that combines both convolutional and LSTM layers in order to quantify the functions of particular DNA sequences. Their problem maps well to this approach as rather than consider an entire sequence of bases, they consider sequences of *motifs*. These motifs emerge as a result of the convolutional layers, and their organisation is considered by the LSTM.

4.2.3 Sequence to Sequence networks

Sequence-to-sequence networks, discussed in section2.1.7 were briefly investigated, with limited success; their complexity resulted in several bugs, precluding their inclusion in the final results. Their ability to produce sequences of any length, despite the input length is essential in natural language processing but undesirable in our particular problem, where the mapping between the number of residues to the number of angles is fixed.

Never-the-less, the use of an auto-encoder is a particularly interesting approach. Rather than decide on a dense or sparse encoding scheme ourselves, having the network discover a useful encoding is a potential avenue of investigation.

Hochreiter *et al*[29] and Quang & Xie[59] have shown improvements in their respective problem domains by having the network learn the rules for motifs automatically, rather than explicitly programming such rules.

4.2.4 Orientation network

Our discretised, classification networks suffer from the problem that each class is distinct, with no relationship to each other. As each class represents an angle, some classes are very much closer to one-another in numerical terms. A miss-classification should have an error that reflects this distance between classes.

Hara *et al*[24] describe a convolutional network that attempts to determine an object's orientation from an image. Like our classification networks, they discretise angles using two values, creating M independent N sized *softmax* classification layers. These are all trained jointly then combined into a final loss function that uses the von-Mises distribution. Although this network is designed to run on images, and produces a single angle, it might be possible to adapt it to produce multiple angles from our antibody data. The functions and distributions used throughout are *wrapped*; they take into account circularity from the outset.

4.2.5 Discrete space

Approaches to creating 3D objects using neural networks include selecting components from a library and combining into a final object, and discretising space into **voxels**, or a point set[17], each of which has a likelihood of being *filled*[1]. Some networks combing both approaches [84].

Rather than discretise the torsion angles, one would discretise space into cubes with a probability of containing an atom (and perhaps a bond). The output layer would be considerably larger with one neuron per cube. Many of these cubes would be unoccupied resulting in a sparse output representation.

The effects of neighbouring loops upon the conformation of CDR-H3 has not been taken into account in this work, however a discrete space approach could be used both as input and output, with the input space including these loops. Discrete cubes of space could be labelled as containing atoms from other loops and presented as part of the input.

4.2.6 Polar coordinate representation

One approach that may be worth considering is a polar coordinate encoding of the output vector, with a cost function that incorporates the notion of rotation. Using our sine & cosine encoding we require 4 numbers, which represent two actual values. A slightly more dense encoding would be to represent both ϕ and ψ angles as a single vector with length 1. ϕ and ψ would map to the polar co-ordinates of this vector. This reduces the size of the output from four digits, to three.

The cost function for such a representation would need to be changed, in order to introduce the notation of rotation, as oppose to simply changing each value in the vector individually. This is certainly possible and results in a slightly more complex function.

This representation couples ϕ and ψ angles together as rotating a vector from its current position to a target would involve a change in both angles, resulting in an implicit relationship. Whilst the Ramachandran plot illustrates that certain

[1]Several examples of model generation by neural nets can be found at the 3DDL Conference proceedings 2016 - `http://3ddl.cs.princeton.edu/2016/`.

combinations of ϕ and ψ are more common than others, this approach may perform best combined with a cost function taking the Ramachandran plot into account as described in section 4.2.1.

As with the 5D vector representation, adding these vectors together would result in a 3D structure that could be used as the basis of a global cost function, albeit one not based on the real positions of the atoms. Such a function would be much easier and faster to compute than NeRF.

4.2.7 Building from both ends

In the original work that inspired this thesis Reczko *et al*[60] consider pairs of residues, starting with the first and last residue pair, proceeding along the loop till the middle is reached. When we experimented with changing the masking to reflect this approach, a change in the results could clearly be seen.

Using a bi-directional LSTM improved performance. Combining both the forward and backward passes using addition was the approach we took, however there are several other approaches available such as taking the mean average.

Rebuilding the loop in Cartesian space with the NeRF algorithm results in errors that compound with each additional residue. Whilst alignment with *PDBfit* can counter this effect when assessing the accuracy of the network, it cannot be used to create original loop models; the original loop not being available.

Building in one direction, then reversing the NeRF algorithm - rebuilding from the last residue to the first - then taking an average of these generated positions might reduce these errors. Taking this one step further, the reverse bi-drectional pass from our final-network could also be output directly, rather than being additively combined with the forward pass, then used to generate a separate reverse loop.

Bibliography

[1] Abhinandan, K., and Martin, A. C. Analysis and improvements to Kabat and structurally correct numbering of antibody variable domains. *Molecular Immunology 45*, 14 (Aug. 2008), 3832–3839.

[2] Abhinandan, K. R., and Martin, A. C. R. Analysis and prediction of VH/VL packing in antibodies. *Protein Engineering Design and Selection 23*, 9 (Sept. 2010), 689–697.

[3] Al-Lazikani, B., Lesk, A. M., and Chothia, C. Standard conformations for the canonical structures of immunoglobulins11edited by I. A. Wilson. *Journal of Molecular Biology 273*, 4 (Nov. 1997), 927–948.

[4] Allcorn, L. C., and Martin, A. C. R. SACS–Self-maintaining database of antibody crystal structure information. *Bioinformatics 18*, 1 (Jan. 2002), 175–181.

[5] Almagro, J. C., Teplyakov, A., Luo, J., Sweet, R. W., Kodangattil, S., Hernandez-Guzman, F., and Gilliland, G. L. Second antibody modeling assessment (AMA-II): 3d Antibody Modeling. *Proteins: Structure, Function, and Bioinformatics 82*, 8 (Aug. 2014), 1553–1562.

[6] Bartlett, P. L., and Maass, W. Vapnik-Chervonenkis Dimension of Neural Nets. In *The Handbook of Brain Theory and Neural Networks* (1995), MIT Press, pp. 1000–1003.

[7] Biasini, M., Bienert, S., Waterhouse, A., Arnold, K., Studer, G., Schmidt, T., Kiefer, F., Cassarino, T. G., Bertoni, M., Bordoli, L., and Schwede, T. SWISS-MODEL: modelling protein tertiary and quaternary structure using evolutionary information. *Nucleic Acids Research 42*, W1 (July 2014), W252–W258.

[8] Bruccoleri, R. E., and Karplus, M. Prediction of the folding of short polypeptide segments by uniform conformational sampling. *Biopolymers 26*, 1 (Jan. 1987), 137–168.

[9] Canutescu, A. A., and Dunbrack, R. L. Cyclic coordinate descent: A robotics algorithm for protein loop closure. *Protein Science 12*, 5 (May 2003), 963–972.

[10] Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv:1406.1078 [cs, stat]* (June 2014). arXiv: 1406.1078.

[11] Chothia, C., Lesk, A. M., Tramontano, A., Levitt, M., Smith-Gill, S. J., Air, G., Sheriff, S., Padlan, E. A., Davies, D., Tulip, W. R., Colman, P. M., Spinelli, S., Alzari, P. M., and Poljak, R. J. Conformations of immunoglobulin hypervariable regions. *Nature 342*, 6252 (Dec. 1989), 877–883.

[12] Deane, C. M., and Blundell, T. L. CODA: a combined algorithm for predicting the structurally variable regions of protein models. *Protein Science 10*, 3 (2001), 599–612.

[13] Duchi, J., Hazan, E., and Singer, Y. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. 39.

[14] Dunbar, J., Krawczyk, K., Leem, J., Baker, T., Fuchs, A., Georges, G., Shi, J., and Deane, C. M. SAbDab: the structural antibody database. *Nucleic Acids Research 42*, D1 (Jan. 2014), D1140–D1146.

[15] Dunbar, J., Krawczyk, K., Leem, J., Marks, C., Nowak, J., Regep, C., Georges, G., Kelm, S., Popovic, B., and Deane, C. M. SAbPred: a structure-based antibody prediction server. *Nucleic Acids Research 44*, W1 (July 2016), W474–W478.

[16] Engelbrecht, A. P. *Computational Intelligence: An Introduction*. John Wiley & Sons, Oct. 2007. Google-Books-ID: IZosIcgJMjUC.

[17] Fan, H., Su, H., and Guibas, L. A Point Set Generation Network for 3d Object Reconstruction from a Single Image. IEEE, pp. 2463–2471.

[18] Ferdous, S., and Martin, A. C. R. AbDb: Antibody structure database — A database of PDB derived antibody structures. 17.

[19] Fiser, A., and Sali, A. Comparative Protein Structure Modeling with MODELLER: a Practical Approach.

[20] Fox, R. O., and Yang, H.-W. Computer-based strategy for peptide and protein conformational ensemble enumeration and ligand affinity analysis, Sept. 2002.

[21] Gers, F. A., Schmidhuber, J., and Cummins, F. Learning to Forget: Continual Prediction with LSTM. *Neural Computation 12*, 10 (Oct. 2000), 2451–2471.

[22] Greff, K., Srivastava, R. K., Koutnik, J., Steunebrink, B. R., and Schmidhuber, J. LSTM: A Search Space Odyssey. *IEEE Transactions on Neural Networks and Learning Systems* (2016), 1–11.

[23] Géron, A. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*.

[24] Hara, K., Vemulapalli, R., and Chellappa, R. Designing Deep Convolutional Neural Networks for Continuous Object Orientation Estimation. *arXiv:1702.01499 [cs]* (Feb. 2017). arXiv: 1702.01499.

[25] Henikoff, S., and Henikoff, J. G. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences of the United States of America 89*, 22 (Nov. 1992), 10915–10919.

[26] Heo, S., Lee, J., Joo, K., Shin, H.-C., and Lee, J. Protein Loop Structure Prediction Using Conformational Space Annealing. *Journal of Chemical Information and Modeling 57*, 5 (May 2017), 1068–1078.

[27] Hildebrand, P. W., Goede, A., Bauer, R. A., Gruening, B., Ismer, J., Michalsky, E., and Preissner, R. SuperLooper—a prediction server for the modeling of loops in globular and membrane proteins. *Nucleic Acids Research 37*, suppl_2 (July 2009), W571–W574.

[28] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580 [cs]* (July 2012). arXiv: 1207.0580.

[29] Hochreiter, S., Heusel, M., and Obermayer, K. Fast model-based protein homology detection without alignment. *Bioinformatics 23*, 14 (July 2007), 1728–1736.

[30] Holtby, D., Li, S. C., and Li, M. LoopWeaver: Loop Modeling by the Weighted Scaling of Verified Proteins. *Journal of Computational Biology 20*, 3 (Mar. 2013), 212–223.

[31] Jacobson, M. P., Pincus, D. L., Rapp, C. S., Day, T. J., Honig, B., Shaw, D. E., and Friesner, R. A. A hierarchical approach to all-atom protein loop prediction. *Proteins: Structure, Function, and Bioinformatics 55*, 2 (Mar. 2004), 351–367.

[32] Jones, D. T. GenTHREADER: an efficient and reliable protein fold recognition method for genomic sequences. *Journal of molecular biology 287*, 4 (1999), 797–815.

[33] Kaas, Q., Ruiz, M., and Lefranc, M.-P. IMGT/3dstructure-DB and IMGT/StructuralQuery, a database and a tool for immunoglobulin, T cell receptor and MHC structural data. *Nucleic Acids Research 32*, suppl_1 (Jan. 2004), D208–D210.

[34] Kabat, E. A., Wu, T. T., Foeller, C., Perry, H. M., and Gottesman, K. S. *Sequences of Proteins of Immunological Interest.* DIANE Publishing, June 1992. Google-Books-ID: 3jMvZYW2ZtwC.

[35] Kim, Y. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[36] Kingma, D. P., and Ba, J. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]* (Dec. 2014). arXiv: 1412.6980.

[37] Laskowski, R. A., Moss, D. S., and Thornton, J. M. Main-chain bond lengths and bond angles in protein structures. *Journal of Molecular Biology 231*, 4 (June 1993), 1049–1067.

[38] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (Nov. 1998), 2278–2324.

[39] Li, J., and Koehl, P. 3d representations of amino acids—applications to protein sequence comparison and classification. *Computational and Structural Biotechnology Journal 11*, 18 (Aug. 2014), 47–58.

[40] Liang, S., Zhang, C., and Zhou, Y. LEAP: Highly accurate prediction of protein loop conformations by integrating coarse-grained sampling and optimized energy scores with all-atom refinement of backbone and side chains. *Journal of Computational Chemistry 35*, 4 (Feb. 2014), 335–341.

[41] MacCallum, R. M., Martin, A. C. R., and Thornton, J. M. Antibody-antigen Interactions: Contact Analysis and Binding Site Topography. *Journal of Molecular Biology 262*, 5 (Oct. 1996), 732–745.

[42] Mandal, C., Kingery, B. D., Anchin, J. M., Subramaniam, S., and Linthicum, D. S. ABGEN: A Knowledge-Based Automated Approach for Antibody Structure Modeling. *Nature Biotechnology 14*, 3 (Mar. 1996), 323–328.

[43] Mandell, D. J., Coutsias, E. A., and Kortemme, T. Sub-angstrom accuracy in protein loop reconstruction by robotics-inspired conformational sampling. *Nature Methods 6*, 8 (Aug. 2009), 551–552.

[44] Marcatili, P., Olimpieri, P. P., Chailyan, A., and Tramontano, A. Antibody structural modeling with prediction of immunoglobulin structure (PIGS). *Nature Protocols 9*, 12 (Nov. 2014), 2771–2783.

[45] Marcatili, P., Rosi, A., and Tramontano, A. PIGS: automatic prediction of antibody structures. *Bioinformatics 24*, 17 (Sept. 2008), 1953–1954.

[46] Marks, C., and Deane, C. Antibody H3 Structure Prediction. *Computational and Structural Biotechnology Journal 15* (2017), 222–231.

[47] Martin, A. C., Cheetham, J. C., and Rees, A. R. Modeling antibody hypervariable loops: a combined algorithm. *Proceedings of the National Academy of Sciences 86*, 23 (1989), 9268–9272.

[48] Martin, A. C., Cheetham, J. C., and Rees, A. R. Modeling antibody hypervariable loops: a combined algorithm. *Proceedings of the National Academy of Sciences 86*, 23 (1989), 9268–9272.

[49] Martin, A. C. R. Protein Sequence and Structure Analysis of Antibody Variable Domains. In *Antibody Engineering*, R. Kontermann and S. D?bel, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 33–51.

[50] McLachlan, A. Rapid Comparison of Protein Structures. *Acta Crystallographica Section A - ACTA CRYSTALLOGR A 38* (Nov. 1982), 871–873.

[51] Messih, M. A., Lepore, R., Marcatili, P., and Tramontano, A. Improving the accuracy of the structure prediction of the third hypervariable loop of the heavy chains of antibodies. *Bioinformatics 30*, 19 (Oct. 2014), 2733–2740.

[52] Morea, V., Tramontano, A., Rustici, M., Chothia, C., and Lesk, A. M. Conformations of the third hypervariable region in the VH domain of immunoglobulins. *Journal of molecular biology 275*, 2 (1998), 269–294.

[53] Moult, J., Fidelis, K., Kryshtafovych, A., Schwede, T., and Tramontano, A. Critical assessment of methods of protein structure prediction (CASP) — round x. *Proteins: Structure, Function, and Bioinformatics 82* (Feb. 2014), 1–6.

[54] North, B., Lehmann, A., and Dunbrack, R. L. A New Clustering of Antibody CDR Loop Conformations. *Journal of Molecular Biology 406*, 2 (Feb. 2011), 228–256.

[55] Orengo, C., Jones, D., and Thornton, J., Eds. *Bioinformatics: Genes, Proteins and Computers*, 1 edition ed. Taylor & Francis, Oxford : New York, July 2003.

[56] Parsons, J., Holmes, J. B., Rojas, J. M., Tsai, J., and Strauss, C. E. M. Practical conversion from torsion space to Cartesian space forin silico protein synthesis. *Journal of Computational Chemistry 26*, 10 (July 2005), 1063–1068.

[57] Pattanayak, S. *Pro Deep Learning with TensorFlow.* Apress, Berkeley, CA, 2017.

[58] Pedersen, J., Searle, S., Henry, A., and Rees, A. R. Antibody modeling: beyond homology. *Immunomethods 1*, 2 (1992), 126–136.

[59] Quang, D., and Xie, X. DanQ: a hybrid convolutional and recurrent deep neural network for quantifying the function of DNA sequences. *Nucleic Acids Research 44*, 11 (June 2016), e107–e107.

[60] Reczko, M., Martin, A. C., Bohr, H., and Suhai, S. Prediction of hypervariable CDR-H3 loop structures in antibodies. *Protein engineering 8*, 4 (1995), 389–395.

[61] Regep, C., Georges, G., Shi, J., Popovic, B., and Deane, C. M. The H3 loop of antibodies shows unique structural characteristics: CDR H3 of Antibodies Shows Unique Structural Characteristics. *Proteins: Structure, Function, and Bioinformatics* (Apr. 2017).

[62] Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review 65*, 6 (Nov. 1958), 386–408.

[63] Samuel, A. L. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development.* (1959).

[64] Shalev-Shwartz, S., and Ben-David, S. Understanding Machine Learning. 416.

[65] Shaw, D. E., Chao, J. C., Eastwood, M. P., Gagliardo, J., Grossman, J. P., Ho, C. R., Lerardi, D. J., Kolossváry, I., Klepeis, J. L., Layman, T., McLeavey, C., Deneroff, M. M., Moraes, M. A., Mueller, R., Priest, E. C., Shan, Y., Spengler, J., Theobald, M., Towles, B., Wang, S. C., Dror, R. O., Kuskin, J. S., Larson, R. H., Salmon, J. K., Young, C., Batson, B., and Bowers, K. J. Anton, a special-purpose machine for molecular dynamics simulation. *Communications of the ACM 51*, 7 (July 2008), 91.

[66] Shirai, H., Kidera, A., and Nakamura, H. Structural classification of CDR-H3 in antibodies. *FEBS letters 399*, 1-2 (1996), 1–8.

[67] Simonyan, K., and Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv:1409.1556 [cs]* (Sept. 2014). arXiv: 1409.1556.

[68] Stein, A., and Kortemme, T. Improvements to robotics-inspired conformational sampling in rosetta. *PLoS One 8*, 5 (2013), e63090.

[69] Sutcliffe, M. J. *An automated approach to the systematic model building of homologous proteins*. PhD Thesis, 1988.

[70] Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to Sequence Learning with Neural Networks. 9.

[71] Swindells, M. B., Porter, C. T., Couch, M., Hurst, J., Abhinandan, K., Nielsen, J. H., Macindoe, G., Hetherington, J., and Martin, A. C. abYsis: Integrated Antibody Sequence and Structure—Management, Analysis, and Prediction. *Journal of Molecular Biology 429*, 3 (Feb. 2017), 356–364.

[72] Swindells, M. B., Porter, C. T., Couch, M., Hurst, J., Abhinandan, K., Nielsen, J. H., Macindoe, G., Hetherington, J., and Martin, A. C. abYsis: Integrated Antibody Sequence and Structure—Management, Analysis, and Prediction. *Journal of Molecular Biology 429*, 3 (Feb. 2017), 356–364.

[73] Sønderby, S. K., and Winther, O. Protein Secondary Structure Prediction with Long Short Term Memory Networks. Studies in computational intelligence, Springer. OCLC: 782967514.

[74] T Porter, C., and C.R. Martin, A. BiopLib and BiopTools - A C programming library and toolset for manipulating protein structure. *Bioinformatics (Oxford, England) 31* (Aug. 2015).

[75] Teplyakov, A., Luo, J., Obmolova, G., Malia, T. J., Sweet, R., Stanfield, R. L., Kodangattil, S., Almagro, J. C., and Gilliland, G. L. Antibody modeling assessment II. Structures and models: Antibody Modeling Assessment. *Proteins: Structure, Function, and Bioinformatics 82*, 8 (Aug. 2014), 1563–1582.

[76] Van den Oord, A., Dieleman, S., and Schrauwen, B. Deep content-based music recommendation. In *Advances in neural information processing systems* (2013), pp. 2643–2651.

[77] Waibel, A. Phoneme Recognition Using Time-Delay Neural Networks.

[78] Weitzner, B. D., Kuroda, D., Marze, N., Xu, J., and Gray, J. J. Blind prediction performance of RosettaAntibody 3.0: Grafting, relaxation, kinematic loop modeling, and full CDR optimization. *Proteins 82*, 8 (Aug. 2014), 1611–1623.

[79] Whitelegg, N. R. J., and Rees, A. R. WAM: an improved algorithm for modelling antibodies on the WEB. *Protein Engineering, Design and Selection 13*, 12 (Dec. 2000), 819–824.

[80] Wu, T. T., and Kabat, E. A. An analysis of the sequences of the variable regions of Bence Jones proteins and myeloma light chains and their implications for antibody complementarity. *The Journal of experimental medicine 132*, 2 (1970), 211–250.

[81] Yamashita, K., Ikeda, K., Amada, K., Liang, S., Tsuchiya, Y., Nakamura, H., Shirai, H., and Standley, D. M. Kotai Antibody Builder: automated high-resolution structural modeling of antibodies. *Bioinformatics 30*, 22 (Nov. 2014), 3279–3280.

[82] Zamani, M., and Kremer, S. C. Amino acid encoding schemes for machine learning methods. IEEE, pp. 327–333.

[83] Zaremba, W., Sutskever, I., and Vinyals, O. Recurrent Neural Network Regularization. *arXiv:1409.2329 [cs]* (Sept. 2014). arXiv: 1409.2329.

[84] Zou, C., Yumer, E., Yang, J., Ceylan, D., and Hoiem, D. 3d-PRNN: Generating Shape Primitives with Recurrent Neural Networks. *arXiv:1708.01648 [cs, stat]* (Aug. 2017). arXiv: 1708.01648.

[85] Zvelebil, M., and Baum, J. *Understanding Bioinformatics*. Garland Science, Sept. 2007. Google-Books-ID: Li0WBAAAQBAJ.

Appendix A

Result tables

A.1 Net 02 results

Run	% end input	% end predicted	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	67.6	56.1	1.89	0.054	12.01	1.85
2	60.63	52.61	1.97	0.06	14.27	2.01
3	68.29	50.17	1.9	0.026	12.51	1.8
4	65.51	47.04	1.94	0.067	10.16	1.85
5	66.55	53.66	1.89	0.1	11.85	1.94
6	65.16	46.69	2.14	0.115	12.16	1.88
7	63.41	48.08	2.0	0.001	11.61	2.02
8	64.46	44.6	1.96	0.097	9.221	1.83
9	71.1	54.36	1.83	0.042	8.91	1.7
10	67.24	50.87	2.14	0.115	12.16	1.88

Table A.1.1: Results of independent runs of neural network 02.

A.2 Net 02a results

Run	% end input	% end predicted	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	73.18	48.23	5.44	0.144	18.91	2.88
2	79.55	59.09	1.72	0.067	11.126	1.852
3	76.36	59.55	1.489	0.088	6.776	1.369
4	79.55	59.55	1.8	0.069	12.024	2.052
5	77.27	65.91	1.641	0.082	8.636	1.596
6	75.91	64.55	1.709	0.106	9.55	1.829
7	73.64	55.91	1.862	0.122	10.896	1.881
8	75.91	58.64	1.658	0.053	9.532	1.618
9	79.09	61.36	1.72	0.098	10.96	1.649
10	81.36	69.09	1.617	0.068	9.493	1.74

Table A.2.1: Results of independent runs of neural network 02a.

A.3 Net 06 results

Run	% end input	% end predicted	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	63.21	51.07	1.625	0.078	15.921	1.96
2	66.79	53.93	1.57	0.079	9.26	1.78
3	68.93	62.14	1.53	0.035	11.397	1.93
4	68.57	58.93	1.5	0.051	7.85	1.8
5	70.0	56.43	1.575	0.035	15.5	2.07
6	72.14	55.37	1.67	0.046	9.578	1.897
7	67.5	58.21	1.79	0.018	10.26	2.06
8	69.64	53.93	1.7	0.038	13.04	2.02
9	63.93	52.86	1.68	0.066	14.91	2.01
10	61.79	53.21	1.6	0.079	10.148	1.79

Table A.3.1: Results of independent runs of neural network 06. Values are given in Ångstroms

A.4 Net 13 results

Run	% end input	% end predicted	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	67.857	58.929	1.528	0.048	12.206	1.957
2	68.214	55.0	1.667	0.088	9.891	1.901
3	65.714	54.286	1.773	0.073	12.695	2.163
4	68.571	57.857	1.564	0.062	12.885	1.914
5	66.429	53.929	1.687	0.068	9.722	1.95
6	67.143	56.786	1.734	0.058	17.458	2.459
7	66.071	56.071	1.646	0.005	12.134	2.076
8	68.929	56.786	1.761	0.035	10.204	1.986
9	65.0	56.786	1.837	0.032	14.927	2.432
10	67.143	52.5	1.847	0.067	11.68	2.228

Table A.4.1: Results of independent runs of neural network 13.

A.5 Net 23 results

Run	% end input	% end predicted	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	68.929	60.357	1.525	0.026	9.353	1.77
2	68.214	61.071	1.4	0.004	12.0	1.765
3	67.5	58.571	1.635	0.011	11.239	2.06
4	63.929	57.857	1.425	0.029	11.87	1.859
5	65.0	55.0	1.472	0.034	13.986	1.914
6	71.071	52.5	1.726	0.03	13.218	2.225
7	64.643	58.571	1.559	0.011	12.227	1.912
8	68.929	60.0	1.393	0.014	10.146	1.817
9	67.143	60.357	1.427	0.002	11.185	1.76
10	67.5	61.429	1.651	0.023	12.837	2.077

Table A.5.1: Results of independent runs of neural network 23.

A.6 Net 23a results

Run	% end input	% end predicted	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	79.091	69.545	1.536	0.049	9.509	1.941
2	77.727	69.091	1.384	0.046	10.929	1.719
3	75.909	71.364	1.35	0.014	8.775	1.66
4	79.545	64.545	1.534	0.067	10.516	1.915
5	79.545	67.272	1.514	0.057	11.451	2.017
6	83.636	70.0	1.301	0.073	9.074	1.513
7	74.545	64.091	1.362	0.071	10.191	1.743
8	80.91	77.727	1.286	0.022	9.561	1.713
9	80.91	67.727	1.416	0.023	9.642	1.643
10	82.272	70.91	1.21	0.033	7.724	1.541

Table A.6.1: Results of independent runs of neural network 23a.

A.7 Non-redundant data results

Run	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	3.36	0.025	8.653	2.013
2	3.54	0.105	10.049	2.469
3	4.056	0.123	9.828	2.234
4	3.934	0.214	14.603	2.499
5	3.536	0.097	12.612	2.295
6	3.653	0.031	13.004	2.573
7	3.48	0.061	9.281	2.19
8	3.472	0.015	13.06	2.35
9	3.646	0.079	11.622	1.995
10	3.676	0.017	11.831	2.585

Table A.7.1: Results of independent runs of neural network 02 on non redundant data from AbDb.

Run	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	3.74	0.172	14.8	2.566
2	3.55	0.089	10.471	2.605
3	3.358	0.39	11.365	2.075
4	3.641	0.078	9.972	2.241
5	3.399	0.037	10.013	2.3
6	3.51	0.109	12.494	2.315
7	3.838	0.06	11.29	2.533
8	3.419	0.167	10.558	2.206
9	3.315	0.151	8.662	2.065
10	3.54	0.082	13.573	2.67

Table A.7.2: Results of independent runs of neural network 02a on non redundant data from AbDb.

Run	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	3.499	0.094	13.241	2.715
2	3.42	0.075	9.924	2.383
3	3.507	0.073	11.365	2.629
4	3.614	0.145	10.928	2.782
5	4.204	0.077	12.525	2.655
6	4.159	0.035	12.397	2.784
7	2.983	0.066	8.938	2.05
8	3.727	0.004	12.006	2.708
9	3.697	0.112	18.301	2.785
10	3.65	0.06	10.801	2.471

Table A.7.3: Results of independent runs of neural network 06 on non-redundant data from AbDb.

Run	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	4.045	0.056	11.337	2.609
2	4.215	0.083	10.153	2.232
3	3.795	0.173	13.081	2.488
4	3.694	0.085	10.54	2.291
5	4.017	0.091	12.888	2.97
6	2.239	0.039	14.873	2.734
7	3.715	0.065	10.701	2.435
8	3.215	0.036	13.515	2.614
9	3.497	0.062	12.814	2.599
10	3.248	0.216	9.59	2.13

Table A.7.4: Results of independent runs of neural network 06a on non-redundant data from AbDb.

Run	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	3.562	0.077	12.239	2.449
2	3.755	0.072	11.251	2.593
3	3.702	0.058	11.161	2.471
4	3.402	0.017	13.814	2.777
5	4.223	0.077	13.85	2.693
6	3.867	0.073	12.874	2.769
7	3.891	0.04	11.766	2.837
8	3.904	0.014	13.359	2.834
9	3.992	0.008	10.148	2.424
10	4.1	0.194	10.751	2.397

Table A.7.5: Results of independent runs of neural network 13 on non-redundant data from AbDb.

Run	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	3.318	0.06	11.877	2.471
2	3.483	0.035	12.297	2.569
3	3.586	0.089	11.072	2.787
4	3.476	0.078	12.991	2.623
5	3.815	0.007	10.406	2.595
6	3.622	0.027	13.867	2.75
7	3.725	0.008	9.071	2.499
8	3.568	0.016	17.256	3.143
9	3.247	0.076	12.153	2.623
10	3.352	0.199	15.871	2.469

Table A.7.6: Results of independent runs of neural network 23 on non-redundant data from AbDb.

Run	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	3.362	0.067	11.721	2.494
2	3.229	0.1	9.842	2.466
3	3.317	0.242	9.233	2.526
4	4.055	0.038	11.366	2.44
5	3.349	0.129	12.332	2.428
6	3.261	0.175	11.807	2.512
7	3.144	0.083	12.555	2.528
8	3.573	0.067	11.839	2.787
9	3.707	0.087	17.763	2.997
10	3.303	0.25	8.31	1.98

Table A.7.7: Results of independent runs of neural network 23a on non-redundant data from AbDb.

A.8 Redundant data in training set only results

Run	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	3.28	0.017	14.2	2.27
2	3.691	0.121	14.048	2.531
3	3.648	0.081	13.48	2.43
4	3.443	0.127	9.591	2.246
5	3.683	0.14	12.478	2.618
6	3.568	0.061	11.991	2.281
7	3.794	0.23	14.912	2.438
8	3.548	0.176	15.563	2.263
9	3.585	0.039	11.572	2.428
10	3.796	0.035	12.92	2.667

Table A.8.1: Results of independent runs of neural network 02 with redundant data in the training set only.

Run	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	1.335	0.024	10.406	1.24

Table A.8.2: Results of independent runs of neural network 02 with redundant data in the training set only, run with the training set.

Run	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	3.886	0.006	13.193	3.046
2	3.661	0.095	11.926	2.43
3	3.735	0.051	14.814	2.97
4	3.383	0.03	9.418	2.422
5	3.586	0.05	13.004	2.677
6	3.886	0.01	11.247	2.606
7	3.742	0.133	13.387	2.532
8	3.723	0.006	11.959	2.782
9	3.508	0.009	12.942	2.606
10	3.1	0.038	15.49	2.529

Table A.8.3: Results of independent runs of neural network 23 with redundant data in the training set only.

Run	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
1	0.957	0.005	10.357	1.175

Table A.8.4: Results of independent runs of neural network 23 with redundant data in the training set only. Scored with the training set

A.9 Labelling network results

Loop type	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
Intermediate Test Set	4.765	0.053	20.103	2.882
Real Test Set	5.251	0.237	21.84	3.521
Intermediate Training Set	4.166	0.0	21.16	3.143
Real Training Set	4.629	0.006	22.33	3.448

Table A.9.1: Result of our sequence labelling network, running on the AbDb dataset using bitfield representation.

Loop type	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
Intermediate Test Set	3.898	0.0	12.202	2.143
Real Test Set	4.244	0.134	16.496	2.508
Intermediate Training Set	1.616	0.0	12.084	1.8
Real Training Set	2.747	0.007	11.637	1.999

Table A.9.2: Result of our sequence labelling network, running on the AbDb dataset using 5D representation.

Loop type	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
Intermediate Test Set	5.248	0.0	26.191	3.556
Real Test Set	5.751	0.006	27.125	2.963
Intermediate Training Set	4.579	0.0	20.287	3.279
Real Training Set	5.067	0.129	23.061	3.6

Table A.9.3: Result of our sequence labelling network, running on the combined LoopDB-AbDb dataset, with bitfield representation.

Loop type	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
Intermediate	6.01	0.08	27.304	3.646
Real	6.41	0.175	29.319	4.04

Table A.9.4: Result of our sequence labelling network, running on the combined LoopDB-AbDb dataset with 5D representation.

A.10 3-Mer network results

Loop type	RMSD Mean	Min RMSD	Max RMSD	RMSD StdDev
Intermediate 5D	7.151	0.038	28.955	4.591
Real 5D	7.698	0.014	31.385	5.043
Intermediate bitfield	5.575	0.0	32.859	3.488
Real bitfield	5.942	0.014	30.539	3.82

Table A.10.1: Result of our 3-mer sequence labelling network, running on the LoopDB and AbDb datasets. Values are in ngstroms

A.11 AMA-II network comparison

Model	ACC	CCG	JEF	JOA	MNT	SCH	PIG	FinalNet
4MA3	5.0	5.1	5.7	5.8	4.8	5.0	-	0.765
4KUZ	4.5	4.8	3.5	3.4	3.0	2.4	4.7	5.611
4KQ3	1.8	4.0	2.4	2.2	2.0	1.6	6.6	3.532
4KQ4	1.6	2.1	1.6	1.8	1.7	3.5	1.5	4.02
4M6M	3.2	3.3	2.9	2.8	2.1	3.4	-	4.477
4M6O	4.4	4.0	3.9	2.3	3.0	3.1	3.9	8.9
4MAU	2.7	3.1	1.3	1.0	2.1	2.0	3.0	3.47
4M7K	3.8	3.2	2.9	2.4	3.9	4.0	3.4	5.241
4KMT	2.6	1.5	1.8	2.0	5.1	2.5	2.1	3.048
4M61	1.8	3.6	2.1	2.5	3.0	2.4	3.7	4.42
4M43	3.3	3.5	3.2	2.2	3.0	3.3	2.2	3.431

Table A.11.1: Results from the AMA-II for CDRH3. Values are in Ångstroms and represent he RMSD between $C\alpha$ atoms within the backbone of the loop.

Appendix B

Program code listings

Code listings are also available from `https://github.com/OniDaito/MRes` and `https://doi.org/10.5281/zenodo.1319787`.

B.1 Generate statistics on reconstruction algorithms

```
"""
stats_recon.py
author : Benjamin Blundell
email : me@benjamin.computer

Test how good NeRF and Martin's reconstruction programs are

"""
import os, sys, signal, subprocess, math
from Bio.PDB import *
import numpy as np

# Import our shared util
parentdir =
↪  os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
os.sys.path.insert(0,parentdir)
import common.acids as acids

def do_stats(names, loops, reals, do_omega = True):
  ''' Perform some stats on our three loop types.'''
  import common.nerf as nerf
  import common.pdb as pdb

  idx = 0
  pairs = []
  for loop in loops:
    nf = nerf.NeRF()
    coords = nf.compute_positions_loop(loop)
```

```
  name = names[idx]
  n0 = name + "_nerf.pdb"
  pdb.coords_to_pdb( n0, name, loop._residues, coords)
  n1 = name + "_real.pdb"
  pdb.coords_to_pdb( n1, name, loop._residues, reals[idx])
  n2 = name + "_mrtn.pdb"
  gen_martin(loop, name, do_omega)
  pairs.append((n0,n1,n2))
  idx += 1

rmsds = []
rmsdm = []
for trio in pairs:
  try:
    pro = subprocess.run(["pdbfit", trio[0], trio[1] ],
    ↪  stdout=subprocess.PIPE)
    tr = pro.stdout.decode()
    tr = float(tr.replace("RMSD  ",""))
    rmsds.append(tr)

    if tr > 1.0:
      print("NeRF", trio[1], tr)

    pro = subprocess.run(["pdbfit", "-w", trio[0], trio[1] ],
    ↪  stdout=subprocess.PIPE)
    tr = pro.stdout.decode()
    with open(trio[1],"w") as f:
      f.write(tr)

    pro = subprocess.run(["pdbfit", trio[1], trio[2] ],
    ↪  stdout=subprocess.PIPE)
    tr = pro.stdout.decode()
    tr = float(tr.replace("RMSD  ",""))
    rmsdm.append(tr)

    if tr > 1.0:
      print("Mrtn", trio[1], tr)

    pro = subprocess.run(["pdbfit", "-w", trio[1], trio[2] ],
    ↪  stdout=subprocess.PIPE)
    tr = pro.stdout.decode()
    with open(trio[2],"w") as f:
      f.write(tr)

  except:
    print("Failed pdbfit on", trio[1])

print("NeRF average", sum(rmsds) / len(rmsds))
print("Martin average", sum(rmsdm) / len(rmsdm))
```

```
def pad(ss):
  pad = ""
  for i in range(0,8-len(ss)):
    pad += " "
  return pad + ss

def gen_martin(loop, name, do_omega = True):
  ''' Create the required output files for genloop.'''
  with open("seq.martin","w") as f:
    f.write("(")
    for res in loop._residues:
      f.write(acids.amino_to_letter(res._name))
    f.write(")\n")

  with open("torsion.martin","w") as f:
    f.write(str(len(loop._residues)))
    f.write(" 1\n\n")
    f.write("title line\n")
    f.write("-----------------------------------\n")

    for res in loop._residues:
      f.write("   " + acids.amino_to_letter(res._name))
      phi = pad("{0:.3f}".format(res.phid()))
      psi = pad("{0:.3f}".format(res.psid()))
      omega = pad("{0:.3f}".format(res.omegad()))
      if not do_omega:
        omega = pad("{0:.3f}".format(180.0))
      f.write("    " + phi + " " + psi + " " + omega + "\n")

  try:
    pro = subprocess.run(["genloop", "seq.martin",
    ↪  "torsion.martin", name + "_mrtn.pdb" ],
    ↪  stdout=subprocess.PIPE)
  except:
    print("Failed genloop on", name)

def gen_reals(names):
  ''' Given a set of model names, find the real atom
  ↪  positions.'''
  import psycopg2
  _db = "pdb_martin"
  _user = "postgres"
  conn = psycopg2.connect("dbname=" + _db + " user=" + _user)
  model_coords = []

  for name in names:
    mname = name.replace(" ","")
    cur_res = conn.cursor()
```

```
      cur_res.execute("SELECT * from atom where chainid='H' and
      ↪  resseq >= 95 and resseq <= 102 and model='" + mname + "'
      ↪  and (name='C' or name='CA' or name='N') order by serial")
      atoms = cur_res.fetchall()
      coords = []
      for atom in atoms:
        coords.append((atom[8], atom[9], atom[10]))

      model_coords.append(coords)

    conn.close()
    return model_coords

  def gen_loops(limit=-1, do_omega=True):
    ''' Generate the loops from our database.'''
    import psycopg2
    from common.gen_data import Loop, Residue

    _db = "pdb_martin"
    _user = "postgres"
    conn = psycopg2.connect("dbname=" + _db + " user=" + _user)
    final_loops = []
    names = []
    cur_model = conn.cursor()
    cur_model.execute("SELECT * from model order by code")
    models = cur_model.fetchall()

    for model in models:
      mname = model[0].replace(" ","")
      new_loop = Loop(mname)

      # Pull out the NeRFed end points
      cur_res = conn.cursor()
      cur_res.execute("SELECT * from nerf where model='" + mname +
      ↪  "'")
      endpoints = cur_res.fetchall()
      if len(endpoints) != 1:
        continue
      endpoint = endpoints[0]
      # Should only be one
      new_loop._endpoint = [endpoint[1], endpoint[2], endpoint[3]]

      cur_res = conn.cursor()
      cur_res.execute("SELECT * from residue where model='" + mname
      ↪  + "' order by resorder")
      residues = cur_res.fetchall()

      temp_residues = []
      for row in residues:
        residue = acids.label_to_amino(row[1])
```

```
      reslabel = row[2]
      resorder = row[3]
      temp_residues.append((residue,reslabel,resorder))

    cur_angle = conn.cursor()
    cur_angle.execute("SELECT * from angle where model='" + mname
    ↪  + "' order by resorder")
    angles = cur_angle.fetchall()

    if len(angles) == 0:
      print("ERROR with model " + mname + ". No angles returned")
      continue

    idx = 0
    for row in angles:
      phi = math.radians(row[1])
      psi = math.radians(row[2])
      omega = math.radians(row[3])
      if not do_omega:
        omega = math.pi

      new_residue = Residue(
          temp_residues[idx][0],
          temp_residues[idx][1],
          temp_residues[idx][2],
          phi,psi,omega)

      new_loop.add_residue(new_residue)
      idx+=1
    #print("Generated",mname)
    names.append(mname)
    if limit != -1:
      if len(final_loops) >= limit:
        break

    final_loops.append(new_loop)

  conn.close()
  return (final_loops, names)

if __name__ == "__main__":
  (loops, names) = gen_loops(limit=-1, do_omega=False)
  reals = gen_reals(names)
  do_stats(names, loops, reals)
```

B.2 Torsion angle creation algorithm

```
"""
torsions.py - module for working out backbone torsions
```

```
author : Benjamin Blundell
email : me@benjamin.computer

"""

import math

def cross(u,v):
  x = (u[1]*v[2]) - (u[2]*v[1])
  y = (u[2]*v[0]) - (u[0]*v[2])
  z = (u[0]*v[1]) - (u[1]*v[0])
  return (x,y,z)

def sub(u,v):
  return (u[0] - v[0], u[1] - v[1], u[2] - v[2])

def norm(u):
  l = 1.0 / length(u)
  return (u[0] *l, u[1] * l, u[2] * l)

def dot(u,v):
  return u[0] * v[0] + u[1] * v[1] + u[2] * v[2]

def length(u) :
  return math.sqrt(u[0] * u[0] + u[1] * u[1] + u[2] * u[2])

def res_atom (residue, label):
  for rr in residue:
    if rr[0] == label:
      return (rr[3], rr[4], rr[5])

# residue is as follows:
# atom.label, atom.resseq, atom.resnum, x, y, z

def derive_angles(residues):
  """ Given the residues, lets take a look at the atoms within
  ↪  and derive the angles. """
  angles = []
  Cap = (0,0,0)
  C = (0,0,0)
  Nn = (0,0,0)
  Ca = (0,0,0)
  N = (0,0,0)
  idx = 0

  for idx in range(0,len(residues)):

    phi = psi = omega = 0
    res = residues[idx]
```

```
N  = res_atom(res,'N')

if idx != 0:
  Cp = C
  C = res_atom(res,'C')
  Cap = Ca
  Ca = res_atom(res,'CA')

  # ORTHONORMAL FRAME (changing basis basically)
  # Results dont seem as close as cos-1 version but
  ↪  apparently, this is more accurate :/
  # Phi
  a = sub(Cp,N)
  b = sub(Ca,N)
  d = b
  n0a = cross(a, b)
  n1a = cross(sub(N,Ca), sub(C,Ca))
  cosphi = -dot(n0a,n1a) / (length(n0a) * length(n1a))
  nx = dot(cross(n0a,n1a),d)
  phi = math.degrees(math.acos(cosphi))
  if nx < 0:
    phi = -180 + phi
  else :
    phi = 180 - phi

if idx < len(residues) - 1:
  Ca = res_atom(res,'CA')
  resn = residues[idx+1]
  Nn = res_atom(resn,'N')
  Can = res_atom(resn,'CA')
  C = res_atom(res,'C')

  # Omega - needed to get Phi and Psi correct because it
  ↪  might flip things around
  # we need to adjust the next angle
  d  = sub(Nn,C)
  n0c = cross(sub(Ca,C), sub(Nn,C))
  n1c = cross(sub(Can,Nn), sub(C,Nn))
  cosomega = -dot(n0c,n1c) / (length(n0c) * length(n1c))
  nx = dot(cross(n0c,n1c),d)
  omega = math.degrees(math.acos(cosomega))
  if nx > 0:
    omega = -omega # this seems too simple :/

  # This method uses arctan and is more reliable apparently
  #m = cross(b,n0a)
  #x = dot(n0a,n1a)
  #y = dot(m,n1a)
  #phi = math.degrees(math.atan2(y,x))
```

```
      # Psi
      a = sub(N,Ca)
      b = sub(C,Ca)
      d = b
      n0b = cross(a, b)
      n1b = cross(sub(Ca,C), sub(Nn,C))
      # Same arctan method
      #m = cross(b,n0b)
      #x = dot(n0b,n1b)
      #y = dot(m,n1b)
      #psi = math.degrees(math.atan2(y,x))
      cospsi = -dot(n0b,n1b) / (length(n0b) * length(n1b))
      nx = dot(cross(n0b,n1b),d)
      psi = math.degrees(math.acos(cospsi))
      if nx < 0:
        psi = -180 + psi
      else :
        psi = 180 - psi

    angles.append((phi, psi, omega))
    idx += 1

  return angles

\begin{end}

\section{NeRF algorithm}
\label{appendix:nerf}
\begin{minted}[breaklines]{python}
"""
The NeRF algorithm

This program converts torsion angles to cartesian co-ordinates
for amino-acid back-bones. Based on the following resources:

http://onlinelibrary.wiley.com/doi/10.1002/jcc.20237/abstract
https://www.ncbi.nlm.nih.gov/pubmed/8515464
https://www.google.com/patents/WO2002073193A1?cl=en

"""

import numpy as np
import math, itertools

#bond_lengths = { "N_TO_A" : 1.4615, "PRO_N_TO_A" : 1.353,
↪  "A_TO_C" : 1.53, "C_TO_N" : 1.325 }
bond_lengths = { "N_TO_A" : 1.4615,  "A_TO_C" : 1.53, "C_TO_N" :
↪  1.325 }
bond_angles = { "A_TO_C" : math.radians(109), "C_TO_N" :
↪  math.radians(115), "N_TO_A" : math.radians(121) }
```

```
bond_order = ["C_TO_N", "N_TO_A", "A_TO_C"]

def next_data(key):
  ''' Loop over our bond_angles and bond_lengths '''
  ff = itertools.cycle(bond_order)
  for item in ff:
    if item == key:
      next_key = next(ff)
      break
  return (bond_angles[next_key], bond_lengths[next_key],
  ↪  next_key)

def place_atom(atom_a, atom_b, atom_c, bond_angle, torsion_angle,
↪  bond_length) :
  ''' Given the three previous atoms, the required angles and the
  ↪  bond
  lengths, place the next atom. Angles are in radians, lengths in
↪  angstroms.'''
  ab = np.subtract(atom_b, atom_a)
  bc = np.subtract(atom_c, atom_b)
  bcn = bc / np.linalg.norm(bc)
  R = bond_length

  # numpy is row major
  d = np.array([-R * math.cos(bond_angle),
      R * math.cos(torsion_angle) * math.sin(bond_angle),
      R * math.sin(torsion_angle) * math.sin(bond_angle)])

  n = np.cross(ab,bcn)
  n = n / np.linalg.norm(n)
  nbc = np.cross(n,bcn)

  m = np.array([
    [bcn[0],nbc[0],n[0]],
    [bcn[1],nbc[1],n[1]],
    [bcn[2],nbc[2],n[2]]])

  d = m.dot(d)
  d = d + atom_c
  return d

def compute_positions(torsions):
  atoms = [[0, -1.355, 0], [0, 0, 0], [1.4466, 0.4981, 0]]
  torsions = list(map(math.radians, torsions))

  key = "C_TO_N"
  angle = bond_angles[key]
  length = bond_lengths[key]

  for torsion in torsions:
```

```
    atoms.append(place_atom(atoms[-3], atoms[-2], atoms[-1],
    ↪  angle, torsion, length))
    (angle, length, key) = next_data(key)

  return atoms

if __name__ == "__main__":

  print ("3NH7_1 - using real omega")
  torsions = [ 142.951668191667, 173.2,
  -147.449854444109, 137.593755455898, -176.98,
  -110.137784727015, 138.084240732612, 162.28,
  -101.068226849313, -96.1690297398444, 167.88,
  -78.7796836206707, -44.3733790929788, 175.88,
  -136.836113196726, 164.182984866024, -172.22,
  -63.909882696529, 143.817250526837, 168.89,
  -144.50345668635, 158.70503596547, 175.87,
  -96.842536650294, 103.724939588454, -172.34,
  -85.7345901579845, -18.1379473766538, -172.98,
  -150.084356709565]

  atoms0 = compute_positions(torsions)
  print(len(atoms0))
  for atom in atoms0:
    print(atom)

  print ("3NH7_1 - using 180 omega")
  torsions = [142.951668191667, 179.0,
  -147.449854444109, 137.593755455898, -176.0,
  -110.137784727015, 138.084240732612, 175.0,
  -101.068226849313, -96.1690297398444, 179.0,
  -78.7796836206707, -44.3733790929788, 179.0,
  -136.836113196726, 164.182984866024, -179.0,
  -63.909882696529, 143.817250526837, 179.0,
  -144.50345668635, 158.70503596547, 179.0,
  -96.842536650294, 103.724939588454, -179.0,
  -85.7345901579845, -18.1379473766538, -179.0,
  -150.084356709565]

  atoms1 = compute_positions(torsions)
  print(len(atoms1))
  for atom in atoms1:
    print(atom)

  print("Diff")
  for idx in range(0,len(atoms0)):
    print(idx, np.subtract(atoms0[idx], atoms1[idx]))
```

B.3 nn02 - convolutional net example

```
"""
nn02.py - Dealing with variable length input
author : Benjamin Blundell
email : me@benjamin.computer

Based on https://www.tensorflow.org/get_started/mnist/pros
and https://danijar.com/variable-sequence-lengths-in-tensorflow/

This version performs the best so far and is probably closest
to the TDNN we want to check

"""

import sys, os, math, random

import tensorflow as tf
import numpy as np

# Import our shared util
parentdir =
↪  os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
os.sys.path.insert(0,parentdir)
from common.util import *
from common.batch_real import *
from common import gen_data

FLAGS = NNGlobals()
# A higher learning rate seems good as we have few examples in
↪  this data set.
# Would that be correct do we think?
FLAGS.learning_rate = 0.35
FLAGS.window_size = 4
FLAGS.pickle_filename = 'pdb_martin_02.pickle'
FLAGS.num_epochs = 2000

def weight_variable(shape, name ):
  '''For now I use truncated normals with stdddev of 0.1.
  ↪  Hopefully
  some of these go negative.'''
  initial = tf.truncated_normal(shape, stddev=0.1, name=name)
  return tf.Variable(initial)

def bias_variable(shape, name):
  initial = tf.constant(1.0, shape=shape, name=name)
  return tf.Variable(initial)

def conv1d(x, W):
  ''' Our convolution is what we use to replicate a TDNN though
```

```
  I suspect we need to do a lot more.'''
  return tf.nn.conv1d(x, W, stride=1, padding='SAME')

def create_graph() :
  ''' My attempt at creating a TDNN with the conv1d operation. We
  ↪  have one conv layer, and two fully
  connected layers (which are quite large). We take a batch x
↪  max_cdr x amino_acid layer and output
  a max_cdr * 4 layer for angle components. We use tanh
↪  activation functions throughout.'''

  graph = tf.Graph()

  with tf.device('/gpu:0'):
    with graph.as_default():
      # Input data - we use a 2D array with each 'channel' being
      ↪  an amino acid bitfield
      # In this case, we only use one example at a time as each
      ↪  is a different length
      tf_train_dataset = tf.placeholder(tf.bool,
          [None, FLAGS.max_cdr_length,
          ↪  FLAGS.num_acids],name="train_input")
      output_size = FLAGS.max_cdr_length * 4
      dmask = tf.placeholder(tf.float32, [None, output_size],
      ↪  name="dmask")
      x = tf.cast(tf_train_dataset, dtype=tf.float32)

      # According to the Tensorflow tutorial, the last two vars
      ↪  are input channels
      # and output channels (both 21)
      W_conv0 = weight_variable([FLAGS.window_size,
        FLAGS.num_acids, FLAGS.num_acids] , "weight_conv_0")
      b_conv0 = bias_variable([FLAGS.num_acids], "bias_conv_0")

      # Using tanh as an activation fuction as it is bounded over
      ↪  -1 to 1
      # Don't have to use it here but we get better accuracy
      h_conv0 = tf.tanh(conv1d(x, W_conv0) + b_conv0)

      # The second layer is fully connected, neural net.
      dim_size = FLAGS.num_acids * FLAGS.max_cdr_length
      W_f = weight_variable([dim_size, output_size],
      ↪  "weight_hidden")
      b_f = bias_variable([output_size], "bias_hidden")


      # Apparently, the convolutional layer needs to be reshaped
      # This bit might be key as our depth, our 21 amino acid
      ↪  neurons are being connected here
      h_conv0_flat = tf.reshape(h_conv0, [-1, dim_size])
```

```
        h_f = tf.tanh( (tf.matmul(h_conv0_flat, W_f) + b_f)) *
        ↪  dmask

        # It looks like I can't take a size < max_cdr and use it,
        ↪  because we have
        # fixed sized stuff so we need to dropout the weights we
        ↪  don't need per sample
        # Find the actual sequence length and only include up to
        ↪  that length
        # We always use dropout even after training
        # Annoyingly tensorflow's dropout doesnt work for us here
        ↪  so I need to
        # add another variable to our h_f layer, deactivating these
        ↪  neurons matched
        test = tf.placeholder(tf.float32, [None, output_size],
        ↪  name="train_test")

        # Output layer - we don't need this because the previous
        ↪  layer is fine but
        # we do get some accuracy increases with another layer/
        # the right number of variables for us but I'll add another
        ↪  one anyway so we have three.
        W_o = weight_variable([output_size, output_size],
        ↪  "weight_output")
        b_o = bias_variable([output_size],"bias_output")

        # I use tanh to bound the results between -1 and 1
        y_conv = tf.tanh( ( tf.matmul(h_f, W_o) + b_o) * dmask,
        ↪  name="output")
        variable_summaries(y_conv, "y_conv")

    return graph

def create_mask(batch):
    ''' create a mask for our fully connected layer, which
    is a [1] shape that is max_cdr * 4 long.'''
    mask = []
    for model in batch:
        mm = []
        for cdr in model:
            tt = 1
            if not 1 in cdr:
                tt = 0
            for i in range(0,4):
                mm.append(tt)
        mask.append(mm)
    return np.array(mask,dtype=np.float32)

def cost(goutput, gtest):
    ''' Our error function which we will try to minimise'''
```

```
    # We find the absolute difference between the output angles and
    ↪  the training angles
    # Can't use cross entropy because thats all to do with
    ↪  probabilities and the like
    # Basic error of sum squares diverges to NaN due to gradient so
    ↪  I go with reduce mean
    # Values of -3.0 are the ones we ignore
    # This could go wrong as adding 3.0 to -3.0 is not numerically
    ↪  stable
    mask = tf.sign(tf.add(gtest,3.0))
    basic_error = tf.square(gtest-goutput) * mask

    # reduce mean doesnt work here as we just want the numbers
    ↪  where mask is 1
    # We work out the mean ourselves
    basic_error = tf.reduce_sum(basic_error)
    basic_error /= tf.reduce_sum(mask)
    return basic_error

def run_session(graph, datasets):
    ''' Run the session once we have a graph, training methodology
    ↪  and a dataset '''
    with tf.device('/gpu:0'):
      with tf.Session(graph=graph) as sess:
        training_input, training_output, validate_input,
        ↪  validate_output, test_input, test_output = datasets
        # Pull out the bits of the graph we need
        ginput = graph.get_tensor_by_name("train_input:0")
        gtest = graph.get_tensor_by_name("train_test:0")
        goutput = graph.get_tensor_by_name("output:0")
        gmask = graph.get_tensor_by_name("dmask:0")
        stepnum = 0
        # Working out the accuracy
        basic_error = cost(goutput, gtest)
        # Setup all the logging for tensorboard
        variable_summaries(basic_error, "Error")
        merged = tf.summary.merge_all()
        train_writer =
        ↪  tf.summary.FileWriter('./summaries/train',graph)
        # So far, I have found Gradient Descent still wins out at
        ↪  the moment
        train_step =
        ↪  tf.train.GradientDescentOptimizer(FLAGS.learning_rate).minimize(basic_error)
        #train_step =
        ↪  tf.train.AdagradOptimizer(FLAGS.learning_rate).minimize(basic_error)
        #train_step =
        ↪  tf.train.AdamOptimizer(1e-4).minimize(basic_error)
        #train_step =
        ↪  tf.train.MomentumOptimizer(FLAGS.learning_rate,
        ↪  0.1).minimize(basic_error)
```

```
      tf.global_variables_initializer().run()
      print('Initialized')

      for i in range(0,FLAGS.num_epochs):
        stepnum = 0
        FLAGS.next_batch = 0
        print("Epoch",i)

        while has_next_batch(training_input, FLAGS):
          item_is, item_os = next_batch(training_input,
          ↪  training_output, FLAGS)
          mask = create_mask(item_is)
          summary, _ = sess.run([merged, train_step],
              feed_dict={ginput: item_is, gtest: item_os, gmask:
              ↪  mask})

          # Find the accuracy at every step, but only print every
          ↪  100
          mask = create_mask(validate_input)
          train_accuracy = basic_error.eval(
              feed_dict={ginput: validate_input, gtest:
              ↪  validate_output,  gmask : mask})

          if stepnum % 50 == 0:
            print('step %d, training accuracy %g' % (stepnum,
            ↪  train_accuracy))

          #dm = gmask.eval(feed_dict={ginput: item_is, gtest:
          ↪  item_os, gmask: mask})
          #print(dm)
          stepnum += 1

      # save our trained net
      saver = tf.train.Saver()
      saver.save(sess, 'saved/nn02')

def run_saved(datasets):
  ''' Load the saved version and then test it against the
  ↪  validation set '''
  with tf.Session() as sess:
    graph = sess.graph
    saver = tf.train.import_meta_graph('saved/nn02.meta')
    saver.restore(sess, 'saved/nn02')
    training_input, training_output, validate_input,
    ↪  validate_output, test_input, test_output = datasets
    goutput = graph.get_tensor_by_name("output:0")
    ginput = graph.get_tensor_by_name("train_input:0")
    gmask = graph.get_tensor_by_name("dmask:0")
    mask = create_mask(validate_input)
```

```
    res = sess.run([goutput], feed_dict={ginput: validate_input,
    ↪  gmask: mask })

    # Now lets output a random example and see how close it is,
    ↪  as well as working out the
    # the difference in mean values. Don't adjust the weights
    ↪  though
    r = random.randint(0, len(validate_input)-1)

    print("Actual              Predicted")
    for i in range(0,len(validate_input[r])):
      sys.stdout.write(bitmask_to_acid(FLAGS,
      ↪  validate_input[r][i]))
      phi = math.degrees(math.atan2(validate_output[r][i*4],
      ↪  validate_output[r][i*4+1]))
      psi = math.degrees(math.atan2(validate_output[r][i*4+2],
      ↪  validate_output[r][i*4+3]))
      sys.stdout.write(": " +
      ↪  "{0:<8}".format("{0:.3f}".format(phi)) + " ")
      sys.stdout.write("{0:<8}".format("{0:.3f}".format(psi)) + "
      ↪  ")
      phi = math.degrees(math.atan2(res[0][r][i*4],
      ↪  res[0][r][i*4+1]))
      psi = math.degrees(math.atan2(res[0][r][i*4+2],
      ↪  res[0][r][i*4+3]))
      sys.stdout.write(" | " +
      ↪  "{0:<8}".format("{0:.3f}".format(phi)) + " ")
      sys.stdout.write("{0:<8}".format("{0:.3f}".format(psi)))
      print("")

def print_error(datasets):
  with tf.Session() as sess:
    graph = sess.graph
    saver = tf.train.import_meta_graph('saved/nn02.meta')
    saver.restore(sess, 'saved/nn02')
    training_input, training_output, validate_input,
    ↪  validate_output, test_input, test_output = datasets
    goutput = graph.get_tensor_by_name("output:0")
    ginput = graph.get_tensor_by_name("train_input:0")
    gmask = graph.get_tensor_by_name("dmask:0")
    gtest = graph.get_tensor_by_name("train_test:0")
    mask = create_mask(test_input)
    basic_error = cost(goutput, gtest)
    test_accuracy = basic_error.eval(
              feed_dict={ginput: test_input, gtest: test_output,
              ↪  gmask : mask})

    print ("Error on test set:", test_accuracy)

def generate_pdbs(datasets):
```

```
''' Load the saved version and write a set of PDBs of both the
↪  predicted
and actual models.'''
with tf.Session() as sess:
  graph = sess.graph
  saver = tf.train.import_meta_graph('saved/nn02.meta')
  saver.restore(sess, 'saved/nn02')
  training_input, training_output, validate_input,
  ↪  validate_output, test_input, test_output = datasets
  goutput = graph.get_tensor_by_name("output:0")
  ginput = graph.get_tensor_by_name("train_input:0")
  gmask = graph.get_tensor_by_name("dmask:0")
  mask = create_mask(test_input)
  res = sess.run([goutput], feed_dict={ginput: test_input,
  ↪  gmask: mask })

  for midx in range(0,len(test_input)):
    torsions_real = []
    torsions_pred = []
    residues = []

    # Put the data in the correct arrays for PDB printing
    for i in range(0,len(test_input[midx])):
      tres = bitmask_to_acid(FLAGS, test_input[midx][i])
      if tres == "***": break
      residues.append((tres,i))
      phi = math.atan2(test_output[midx][i*4],
      ↪  test_output[midx][i*4+1])
      psi = math.atan2(test_output[midx][i*4+2],
      ↪  test_output[midx][i*4+3])
      torsions_real.append([phi,psi])
      phi = math.atan2(res[0][midx][i*4], res[0][midx][i*4+1])
      psi = math.atan2(res[0][midx][i*4+2],
      ↪  res[0][midx][i*4+3])
      torsions_pred.append([phi,psi])

    torsions_pred[0][0] = 0.0
    torsions_real[0][0] = 0.0
    torsions_pred[len(torsions_pred)-1][1] = 0.0
    torsions_real[len(torsions_real)-1][1] = 0.0

    from common import torsion_to_coord as tc

    mname = str(midx).zfill(3) + "_real.pdb"
    with open(mname,'w') as f:
      pf = {}
      pf["angles"] = torsions_real
      pf["residues"] = residues
      entries = tc.process(pf)
      f.write(tc.printpdb(mname, entries, residues))
```

```
        mname = str(midx).zfill(3) + "_pred.pdb"
        with open(mname,'w') as f:
          pf = {}
          pf["angles"] = torsions_pred
          pf["residues"] = residues
          entries = tc.process(pf)
          f.write(tc.printpdb(mname, entries, residues))

        mname = str(midx).zfill(3) + "_real.txt"
        with open(mname,'w') as f:
          for i in range(0, len(residues)):
            f.write(residues[i][0] + ": " +
            ↪  str(torsions_real[i][0]) + ", " +
            ↪  str(torsions_real[i][1]) + "\n")

        mname = str(midx).zfill(3) + "_pred.txt"
        with open(mname,'w') as f:
          for i in range(0, len(residues)):
            f.write(residues[i][0] + ": " +
            ↪  str(torsions_pred[i][0]) + ", " +
            ↪  str(torsions_pred[i][1]) + "\n")

if __name__ == "__main__":
  from common import gen_data
  # If we just want to run the trained net
  datasets = init_data_sets(FLAGS, gen_data)

  if len(sys.argv) > 1:
    if sys.argv[1] == "-r":
      run_saved(datasets)
      sys.exit()

    elif sys.argv[1] == "-e":
      print_error(datasets)
      sys.exit()

    elif sys.argv[1] == "-g":
      generate_pdbs(datasets)
      sys.exit()

  graph = create_graph()
  run_session(graph, datasets)
  run_saved(datasets)
```

B.4 nn06 - Bi-directional LSTM

```
"""
nn06.py - A bidirectional LSTM attempt
```

```
author : Benjamin Blundell
email : me@benjamin.computer

We pad out the data to the maximum, but for each input
we find the real length and both stop the LSTM unrolls
at that point (dynamic) and mask out the output layers
so the cost function doesn't take these padded values
into account.

"""

import sys, os, math, random

import tensorflow as tf
import numpy as np

# Import our shared util
parentdir =
↪  os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
os.sys.path.insert(0,parentdir)
from common.util import *
from common.batch_real import *
from common import gen_data

FLAGS = NNGlobals()
# A higher learning rate seems good as we have few examples in
↪  this data set.
# Would that be correct do we think?
FLAGS.learning_rate = 0.45
FLAGS.pickle_filename = 'pdb_martin_06.pickle'
FLAGS.lstm_size = 256   # number of neurons per LSTM cell do we
↪  think?
FLAGS.num_epochs = 2000 # number of loops around the training set
FLAGS.batch_size = 20

def weight_variable(shape, name ):
  ''' For now I use truncated normals with stdddev of 0.1.'''
  initial = tf.truncated_normal(shape, stddev=0.1, name=name)
  return tf.Variable(initial)

def bias_variable(shape, name):
  initial = tf.constant(1.0, shape=shape, name=name)
  return tf.Variable(initial)

def lstm_cell(size, kprob):
  ''' Return an LSTM Cell or other RNN type cell. We
  have a few choices. We can even throw in a bit of
  dropout if we want.'''

  cell= tf.nn.rnn_cell.BasicLSTMCell(size)
```

```
  #cell = tf.nn.rnn_cell.GRUCell(size)
  #cell = tf.nn.rnn_cell.BasicRNNCell(size)
  cell = tf.nn.rnn_cell.DropoutWrapper(cell=cell,
  ↪  output_keep_prob=kprob)
  return cell

def create_graph() :
  graph = tf.Graph()

  with tf.device('/gpu:0'):
    with graph.as_default():
      # Input data. We take in padded CDRs but feed in a length /
      ↪  mask as well
      # Apparently the dynamic RNN thingy can cope with variable
      ↪  lengths
      # Input has to be [batch_size, max_time, ...]
      tf_train_dataset = tf.placeholder(tf.int32, [None,
      ↪  FLAGS.max_cdr_length,
      ↪  FLAGS.num_acids],name="train_input")
      output_size = FLAGS.max_cdr_length * 4
      dmask = tf.placeholder(tf.float32, [None, output_size],
      ↪  name="dmask")
      x = tf.cast(tf_train_dataset, dtype=tf.float32)

      # Since we are using dropout, we need to have a
      ↪  placeholder, so we dont set
      # dropout at validation time
      keep_prob = tf.placeholder(tf.float32, name="keepprob")


      single_rnn_cell = lstm_cell(FLAGS.lstm_size, keep_prob)
      # 'outputs' is a tensor of shape [batch_size,
      ↪  max_cdr_length, lstm_size]
      # 'state' is a N-tuple where N is the number of LSTMCells
      ↪  containing a
      # tf.contrib.rnn.LSTMStateTuple for each cell
      length = create_length(x)
      initial_state =
      ↪  single_rnn_cell.zero_state(FLAGS.batch_size,
      ↪  dtype=tf.float32)
      outputs, states =
      ↪  tf.nn.bidirectional_dynamic_rnn(cell_fw=single_rnn_cell,
      ↪  cell_bw=single_rnn_cell, inputs=x, dtype=tf.float32,
      ↪  sequence_length = length)

      output_fw, output_bw = outputs
      states_fw, states_bw = states

      # We flatten out the outputs so it just looks like a big
      ↪  batch to our weight matrix
```

```
        # apparently this gives us weights across the entire set of
        ↪  steps

        output_fw = tf.reshape(output_fw, [-1, FLAGS.lstm_size],
        ↪  name="flattened_fw")
        output_bw = tf.reshape(output_bw, [-1, FLAGS.lstm_size],
        ↪  name="flattened_bw")

        output = tf.add(output_fw, output_bw)

        test = tf.placeholder(tf.float32, [None, output_size],
        ↪  name="train_test")

        W_i = weight_variable([FLAGS.lstm_size, 4],
        ↪  "weight_intermediate")
        b_i = bias_variable([4],"bias_intermediate")
        y_i = tf.tanh( ( tf.matmul( output, W_i) + b_i),
        ↪  name="intermediate")

        # Now reshape it back and run the mask against it
        y_b = tf.reshape(y_i, [-1, output_size], name="output")
        y_b = y_b * dmask



    return graph

def create_mask(batch):
    ''' create a mask for our fully connected layer, which
    is a [1] shape that is max_cdr * 4 long.'''
    mask = []
    for model in batch:
        mm = []
        for cdr in model:
            tt = 1
            if not 1 in cdr:
                tt = 0
            for i in range(0,4):
                mm.append(tt)
        mask.append(mm)
    return np.array(mask,dtype=np.float32)

def create_length(batch):
    ''' return the actual lengths of our CDR here. Taken from
    https://danijar.com/variable-sequence-lengths-in-tensorflow/
↪   '''
    used = tf.sign(tf.reduce_max(tf.abs(batch), 2))
    length = tf.reduce_sum(used, 1)
    length = tf.cast(length, tf.int32)
    return length
```

```
def cost(goutput, gtest):
  ''' Our error function which we will try to minimise'''
  # We find the absolute difference between the output angles and
  ↪  the training angles
  # Can't use cross entropy because thats all to do with
  ↪  probabilities and the like
  # Basic error of sum squares diverges to NaN due to gradient so
  ↪  I go with reduce mean
  # Values of -3.0 are the ones we ignore
  # This could go wrong as adding 3.0 to -3.0 is not numerically
  ↪  stable
  mask = tf.sign(tf.add(gtest,3.0))
  basic_error = tf.square(gtest-goutput) * mask

  # reduce mean doesnt work here as we just want the numbers
  ↪  where mask is 1
  # We work out the mean ourselves
  basic_error = tf.reduce_sum(basic_error)
  basic_error /= tf.reduce_sum(mask)
  return basic_error

def run_session(graph, datasets):
  ''' Run the session once we have a graph, training methodology
  ↪  and a dataset '''
  with tf.device('/gpu:0'):
    with tf.Session(graph=graph) as sess:

      training_input, training_output, validate_input,
      ↪  validate_output, test_input, test_output = datasets
      # Pull out the bits of the graph we need
      ginput = graph.get_tensor_by_name("train_input:0")
      gtest = graph.get_tensor_by_name("train_test:0")
      goutput = graph.get_tensor_by_name("output:0")
      gmask = graph.get_tensor_by_name("dmask:0")
      gprob = graph.get_tensor_by_name("keepprob:0")

      # Working out the accuracy
      basic_error = cost(goutput, gtest)
      # Setup all the logging for tensorboard
      variable_summaries(basic_error, "Error")
      merged = tf.summary.merge_all()
      train_writer =
      ↪  tf.summary.FileWriter('./summaries/train',graph)

      #train_step =
      ↪  tf.train.GradientDescentOptimizer(FLAGS.learning_rate).minimize(basic_error)
      optimizer = tf.train.AdagradOptimizer(FLAGS.learning_rate)

      gvs = optimizer.compute_gradients(basic_error)
```

```
capped_gvs = [(tf.clip_by_value(grad, -1., 1.), var) for
↪  grad, var in gvs]
train_step = optimizer.apply_gradients(capped_gvs)

#train_step =
↪  tf.train.AdamOptimizer(1e-4).minimize(basic_error)
#train_step =
↪  tf.train.MomentumOptimizer(FLAGS.learning_rate,
↪  0.1).minimize(basic_error)

tf.global_variables_initializer().run()
print('Initialized')

for i in range(0,FLAGS.num_epochs):
  stepnum = 0
  FLAGS.next_batch = 0
  print("Epoch",i)

  while has_next_batch(training_input, FLAGS):
    batch_is, batch_os = next_batch(training_input,
    ↪  training_output, FLAGS)
    batch_iv, batch_ov = random_batch(validate_input,
    ↪  validate_output, FLAGS)

    # For some reason, if the batches are not ALL the same
    ↪  size, we get a crash
    # so I reject batches smaller than the one set
    if len(batch_is) != FLAGS.batch_size or len(batch_iv)
    ↪  != FLAGS.batch_size:
      continue

    mask = create_mask(batch_is)
    summary, _ = sess.run([merged, train_step],
        feed_dict={ginput: batch_is, gtest: batch_os,
        ↪  gmask: mask, gprob: 0.8})

    # Find the accuracy at every step, but only print every
    ↪  100
    # We have to batch here too for some reason? LSTM or
    ↪  something?
    mask = create_mask(batch_iv)
    train_accuracy = basic_error.eval(
        feed_dict={ginput: batch_iv, gtest: batch_ov,
        ↪  gmask: mask, gprob: 1.0})

    if stepnum % 10 == 0:
      print('step %d, training accuracy %g' % (stepnum,
      ↪  train_accuracy))

    train_writer.add_summary(summary, stepnum)
```

```
            stepnum += 1

        # save our trained net
        saver = tf.train.Saver()
        saver.save(sess, 'saved/nn06')

def run_saved(datasets):
    ''' Load the saved version and then test it against the
    ↪  validation set '''
    with tf.Session() as sess:
        graph = sess.graph
        saver = tf.train.import_meta_graph('saved/nn06.meta')
        saver.restore(sess, 'saved/nn06')
        training_input, training_output, validate_input,
        ↪  validate_output, test_input, test_output = datasets
        goutput = graph.get_tensor_by_name("output:0")
        ginput = graph.get_tensor_by_name("train_input:0")
        gmask = graph.get_tensor_by_name("dmask:0")
        gprob = graph.get_tensor_by_name("keepprob:0")
        mask = create_mask(validate_input)
        res = sess.run([goutput], feed_dict={ginput: validate_input,
        ↪  gmask: mask, gprob: 1.0})

        # Now lets output a random example and see how close it is,
        ↪  as well as working out the
        # the difference in mean values. Don't adjust the weights
        ↪  though
        r = random.randint(0, len(validate_input)-1)

        print("Actual              Predicted")
        for i in range(0,len(validate_input[r])):
            sys.stdout.write(bitmask_to_acid(FLAGS,
            ↪  validate_input[r][i]))
            phi = math.degrees(math.atan2(validate_output[r][i*4],
            ↪  validate_output[r][i*4+1]))
            psi = math.degrees(math.atan2(validate_output[r][i*4+2],
            ↪  validate_output[r][i*4+3]))
            sys.stdout.write(": " +
            ↪  "{0:<8}".format("{0:.3f}".format(phi)) + " ")
            sys.stdout.write("{0:<8}".format("{0:.3f}".format(psi)) + "
            ↪  ")
            phi = math.degrees(math.atan2(res[0][r][i*4],
            ↪  res[0][r][i*4+1]))
            psi = math.degrees(math.atan2(res[0][r][i*4+2],
            ↪  res[0][r][i*4+3]))
            sys.stdout.write(" | " +
            ↪  "{0:<8}".format("{0:.3f}".format(phi)) + " ")
            sys.stdout.write("{0:<8}".format("{0:.3f}".format(psi)))
            print("")
```

```
def print_error(datasets):
  with tf.Session() as sess:
    graph = sess.graph
    saver = tf.train.import_meta_graph('saved/nn06.meta')
    saver.restore(sess, 'saved/nn06')
    training_input, training_output, validate_input,
    ↪  validate_output, test_input, test_output = datasets
    goutput = graph.get_tensor_by_name("output:0")
    ginput = graph.get_tensor_by_name("train_input:0")
    gprob = graph.get_tensor_by_name("keepprob:0")
    gmask = graph.get_tensor_by_name("dmask:0")
    gtest = graph.get_tensor_by_name("train_test:0")
    mask = create_mask(test_input)
    basic_error = cost(goutput, gtest)

    test_input = test_input[:FLAGS.batch_size]
    test_output = test_output[:FLAGS.batch_size]

    test_accuracy = basic_error.eval(
        feed_dict={ginput: test_input, gtest: test_output, gmask
        ↪  : mask, gprob: 1.0})

    print ("Error on test set:", test_accuracy)

def generate_pdbs(datasets):
  ''' Load the saved version and write a set of PDBs of both the
  ↪  predicted
  and actual models.'''
  with tf.Session() as sess:
    graph = sess.graph
    saver = tf.train.import_meta_graph('saved/nn06.meta')
    saver.restore(sess, 'saved/nn06')
    training_input, training_output, validate_input,
    ↪  validate_output, test_input, test_output = datasets
    goutput = graph.get_tensor_by_name("output:0")
    ginput = graph.get_tensor_by_name("train_input:0")
    gmask = graph.get_tensor_by_name("dmask:0")
    gprob = graph.get_tensor_by_name("keepprob:0")
    mask = create_mask(test_input)

    midx = 0

    for k in range(0, len(test_input), FLAGS.batch_size):
      test_input_batch = test_input[k:k+FLAGS.batch_size]
      test_output_batch = test_output[k:k+FLAGS.batch_size]

      if len(test_input_batch) != FLAGS.batch_size:
        break
```

```
res = sess.run([goutput], feed_dict={ginput:
↪  test_input_batch, gmask: mask, gprob: 1.0 })

for j in range(0,len(test_input_batch)):
  torsions_real = []
  torsions_pred = []
  residues = []

  # Put the data in the correct arrays for PDB printing
  for i in range(0,len(test_input_batch[j])):
    tres = bitmask_to_acid(FLAGS, test_input_batch[j][i])
    if tres == "***": break
    residues.append((tres,i))
    phi = math.atan2(test_output_batch[j][i*4],
    ↪  test_output_batch[j][i*4+1])
    psi = math.atan2(test_output_batch[j][i*4+2],
    ↪  test_output_batch[j][i*4+3])
    torsions_real.append([phi,psi])
    phi = math.atan2(res[0][j][i*4], res[0][j][i*4+1])
    psi = math.atan2(res[0][j][i*4+2], res[0][j][i*4+3])
    torsions_pred.append([phi,psi])

  torsions_pred[0][0] = 0.0
  torsions_real[0][0] = 0.0
  torsions_pred[len(torsions_pred)-1][1] = 0.0
  torsions_real[len(torsions_real)-1][1] = 0.0

  from common import torsion_to_coord as tc

  mname = str(midx).zfill(3) + "_real.pdb"
  with open(mname,'w') as f:
    pf = {}
    pf["angles"] = torsions_real
    pf["residues"] = residues
    entries = tc.process(pf)
    f.write(tc.printpdb(mname, entries, residues))

  mname = str(midx).zfill(3) + "_pred.pdb"
  with open(mname,'w') as f:
    pf = {}
    pf["angles"] = torsions_pred
    pf["residues"] = residues
    entries = tc.process(pf)
    f.write(tc.printpdb(mname, entries, residues))

  mname = str(midx).zfill(3) + "_real.txt"
  with open(mname,'w') as f:
    for i in range(0, len(residues)):
```

```
            f.write(residues[i][0] + ": " +
            ↪  str(torsions_real[i][0]) + ", " +
            ↪  str(torsions_real[i][1]) + "\n")

        mname = str(midx).zfill(3) + "_pred.txt"
        with open(mname,'w') as f:
          for i in range(0, len(residues)):
            f.write(residues[i][0] + ": " +
            ↪  str(torsions_pred[i][0]) + ", " +
            ↪  str(torsions_pred[i][1]) + "\n")

        midx += 1

if __name__ == "__main__":
  from common import gen_data
  # If we just want to run the trained net
  if len(sys.argv) > 1:
    if sys.argv[1] == "-r":
      datasets = init_data_sets(FLAGS, gen_data)
      run_saved(datasets)
      sys.exit()
    if sys.argv[1] == "-e":
      datasets = init_data_sets(FLAGS, gen_data)
      print_error(datasets)
      sys.exit()
    if sys.argv[1] == "-g":
      datasets = init_data_sets(FLAGS, gen_data)
      generate_pdbs(datasets)
      sys.exit()

  datasets = init_data_sets(FLAGS, gen_data)
  graph = create_graph()
  run_session(graph, datasets)
  run_saved(datasets)
```

B.5 nn13 - LSTM with 5D encoding

```
"""
nn13.py - A bidirectional LSTM attempt with 5D encoding
author : Benjamin Blundell
email : me@benjamin.computer

We pad out the data to the maximum, but for each input
we find the real length and both stop the LSTM unrolls
at that point (dynamic) and mask out the output layers
so the cost function doesn't take these padded values
into account.

"""
```

```
import sys, os, math, random

import tensorflow as tf
import numpy as np

# Import our shared util
parentdir =
↪  os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
os.sys.path.insert(0,parentdir)
from common.util import *
from common.batch_real_3d import *
from common import gen_data

FLAGS = NNGlobals()
# A higher learning rate seems good as we have few examples in
↪  this data set.
# Would that be correct do we think?
FLAGS.learning_rate = 0.45
FLAGS.pickle_filename = 'pdb_martin_13.pickle'
FLAGS.lstm_size = 256   # number of neurons per LSTM cell do we
↪  think?
FLAGS.num_epochs = 2000 # number of loops around the training set
FLAGS.batch_size = 20

def weight_variable(shape, name ):
  ''' For now I use truncated normals with stdddev of 0.1.'''
  initial = tf.truncated_normal(shape, stddev=0.1, name=name)
  return tf.Variable(initial)

def bias_variable(shape, name):
  initial = tf.constant(1.0, shape=shape, name=name)
  return tf.Variable(initial)

def lstm_cell(size, kprob):
  ''' Return an LSTM Cell or other RNN type cell. We
  have a few choices. We can even throw in a bit of
  dropout if we want.'''

  cell= tf.nn.rnn_cell.BasicLSTMCell(size)
  #cell = tf.nn.rnn_cell.GRUCell(size)
  #cell = tf.nn.rnn_cell.BasicRNNCell(size)
  cell = tf.nn.rnn_cell.DropoutWrapper(cell=cell,
  ↪  output_keep_prob=kprob)
  return cell

def create_graph() :
  graph = tf.Graph()

  with tf.device('/gpu:0'):
```

```
with graph.as_default():
  # Input data. We take in padded CDRs but feed in a length /
  ↪  mask as well
  # Apparently the dynamic RNN thingy can cope with variable
  ↪  lengths
  # Input has to be [batch_size, max_time, ...]
  tf_train_dataset = tf.placeholder(tf.float32, [None,
  ↪  FLAGS.max_cdr_length, 5],name="train_input")
  output_size = FLAGS.max_cdr_length * 4
  dmask = tf.placeholder(tf.float32, [None, output_size],
  ↪  name="dmask")
  x = tf.cast(tf_train_dataset, dtype=tf.float32)

  # Since we are using dropout, we need to have a
  ↪  placeholder, so we dont set
  # dropout at validation time
  keep_prob = tf.placeholder(tf.float32, name="keepprob")

  # This is the number of unrolls I think - sequential cells
  # In this example, I'm going for max_cdr_length as we want
  ↪  all the history
  # This will take a while and it is dynamically sized based
  ↪  on the inputs.

  single_rnn_cell = lstm_cell(FLAGS.lstm_size, keep_prob)
  # 'outputs' is a tensor of shape [batch_size,
  ↪  max_cdr_length, lstm_size]
  # 'state' is a N-tuple where N is the number of LSTMCells
  ↪  containing a
  # tf.contrib.rnn.LSTMStateTuple for each cell
  length = create_length(x)
  initial_state =
  ↪  single_rnn_cell.zero_state(FLAGS.batch_size,
  ↪  dtype=tf.float32)
  outputs, states =
  ↪  tf.nn.bidirectional_dynamic_rnn(cell_fw=single_rnn_cell,
  ↪  cell_bw=single_rnn_cell, inputs=x, dtype=tf.float32,
  ↪  sequence_length = length)

  output_fw, output_bw = outputs
  states_fw, states_bw = states

  # We flatten out the outputs so it just looks like a big
  ↪  batch to our weight matrix
  # apparently this gives us weights across the entire set of
  ↪  steps

  output_fw = tf.reshape(output_fw, [-1, FLAGS.lstm_size],
  ↪  name="flattened_fw")
```

```
        output_bw = tf.reshape(output_bw, [-1, FLAGS.lstm_size],
        ↪  name="flattened_bw")

        output = tf.add(output_fw, output_bw)

        test = tf.placeholder(tf.float32, [None, output_size],
        ↪  name="train_test")

        W_i = weight_variable([FLAGS.lstm_size, 4],
        ↪  "weight_intermediate")
        b_i = bias_variable([4],"bias_intermediate")
        y_i = tf.tanh( ( tf.matmul( output, W_i) + b_i),
        ↪  name="intermediate")

        # Now reshape it back and run the mask against it
        y_b = tf.reshape(y_i, [-1, output_size], name="output")
        y_b = y_b * dmask

    return graph

def create_mask(batch):
    ''' create a mask for our fully connected layer, which
    is a [1] shape that is max_cdr * 4 long.'''
    mask = []
    for model in batch:
        mm = []
        for cdr in model:
            tt = 1
            if sum(cdr) == 0:
                tt = 0
            for i in range(0,4):
                mm.append(tt)
        mask.append(mm)
    return np.array(mask,dtype=np.float32)

def create_length(batch):
    ''' return the actual lengths of our CDR here. Taken from
    https://danijar.com/variable-sequence-lengths-in-tensorflow/
↪   '''
    used = tf.sign(tf.reduce_max(tf.abs(batch), 2))
    length = tf.reduce_sum(used, 1)
    length = tf.cast(length, tf.int32)
    return length

def cost(goutput, gtest):
    ''' Our error function which we will try to minimise'''
    mask = tf.sign(tf.add(gtest,3.0))
    basic_error = tf.square(gtest-goutput) * mask
    basic_error = tf.reduce_sum(basic_error)
    basic_error /= tf.reduce_sum(mask)
```

```
  return basic_error

def run_session(graph, datasets):
  ''' Run the session once we have a graph, training methodology
  ↪  and a dataset '''
  with tf.device('/gpu:0'):
    with tf.Session(graph=graph) as sess:

      training_input, training_output, validate_input,
      ↪  validate_output, test_input, test_output = datasets
      # Pull out the bits of the graph we need
      ginput = graph.get_tensor_by_name("train_input:0")
      gtest = graph.get_tensor_by_name("train_test:0")
      goutput = graph.get_tensor_by_name("output:0")
      gmask = graph.get_tensor_by_name("dmask:0")
      gprob = graph.get_tensor_by_name("keepprob:0")

      # Working out the accuracy
      basic_error = cost(goutput, gtest)
      # Setup all the logging for tensorboard
      variable_summaries(basic_error, "Error")
      merged = tf.summary.merge_all()
      train_writer =
      ↪  tf.summary.FileWriter('./summaries/train',graph)

      # So far, I have found Gradient Descent still wins out at
      ↪  the moment

      #train_step =
      ↪  tf.train.GradientDescentOptimizer(FLAGS.learning_rate).minimize(basic_error)
      optimizer = tf.train.AdagradOptimizer(FLAGS.learning_rate)

      gvs = optimizer.compute_gradients(basic_error)
      capped_gvs = [(tf.clip_by_value(grad, -1., 1.), var) for
      ↪  grad, var in gvs]
      train_step = optimizer.apply_gradients(capped_gvs)

      #train_step =
      ↪  tf.train.AdamOptimizer(1e-4).minimize(basic_error)
      #train_step =
      ↪  tf.train.MomentumOptimizer(FLAGS.learning_rate,
      ↪  0.1).minimize(basic_error)

      tf.global_variables_initializer().run()
      print('Initialized')

      for i in range(0,FLAGS.num_epochs):
        stepnum = 0
        FLAGS.next_batch = 0
        print("Epoch",i)
```

```
        while has_next_batch(training_input, FLAGS):
          batch_is, batch_os = next_batch(training_input,
          ↪  training_output, FLAGS)
          batch_iv, batch_ov = random_batch(validate_input,
          ↪  validate_output, FLAGS)

          # For some reason, if the batches are not ALL the same
          ↪  size, we get a crash
          # so I reject batches smaller than the one set
          if len(batch_is) != FLAGS.batch_size or len(batch_iv)
          ↪  != FLAGS.batch_size:
            continue

          mask = create_mask(batch_is)
          summary, _ = sess.run([merged, train_step],
              feed_dict={ginput: batch_is, gtest: batch_os,
              ↪  gmask: mask, gprob: 0.8})

          if stepnum % 10 == 0:
            mask = create_mask(batch_iv)
            train_accuracy = basic_error.eval(
              feed_dict={ginput: batch_iv, gtest: batch_ov,
              ↪  gmask: mask, gprob: 1.0})

            print('step %d, training accuracy %g' % (stepnum,
            ↪  train_accuracy))

          train_writer.add_summary(summary, stepnum)
          stepnum += 1

      # save our trained net
      saver = tf.train.Saver()
      saver.save(sess, 'saved/nn13')

def run_saved(datasets):
  ''' Load the saved version and then test it against the
  ↪  validation set '''
  with tf.Session() as sess:
    graph = sess.graph
    saver = tf.train.import_meta_graph('saved/nn13.meta')
    saver.restore(sess, 'saved/nn13')
    training_input, training_output, validate_input,
    ↪  validate_output, test_input, test_output = datasets
    goutput = graph.get_tensor_by_name("output:0")
    ginput = graph.get_tensor_by_name("train_input:0")
    gmask = graph.get_tensor_by_name("dmask:0")
    gprob = graph.get_tensor_by_name("keepprob:0")
    mask = create_mask(validate_input)
```

```
    res = sess.run([goutput], feed_dict={ginput: validate_input,
    ↪  gmask: mask, gprob: 1.0})

    # Now lets output a random example and see how close it is,
    ↪  as well as working out the
    # the difference in mean values. Don't adjust the weights
    ↪  though
    r = random.randint(0, len(validate_input)-1)

    print("Actual              Predicted")
    for i in range(0,len(validate_input[r])):
      sys.stdout.write(vector_to_acid(FLAGS,
      ↪  validate_input[r][i]))
      phi = math.degrees(math.atan2(validate_output[r][i*4],
      ↪  validate_output[r][i*4+1]))
      psi = math.degrees(math.atan2(validate_output[r][i*4+2],
      ↪  validate_output[r][i*4+3]))
      sys.stdout.write(": " +
      ↪  "{0:<8}".format("{0:.3f}".format(phi)) + " ")
      sys.stdout.write("{0:<8}".format("{0:.3f}".format(psi)) + "
      ↪  ")
      phi = math.degrees(math.atan2(res[0][r][i*4],
      ↪  res[0][r][i*4+1]))
      psi = math.degrees(math.atan2(res[0][r][i*4+2],
      ↪  res[0][r][i*4+3]))
      sys.stdout.write(" | " +
      ↪  "{0:<8}".format("{0:.3f}".format(phi)) + " ")
      sys.stdout.write("{0:<8}".format("{0:.3f}".format(psi)))
      print("")

def print_error(datasets):
  with tf.Session() as sess:
    graph = sess.graph
    saver = tf.train.import_meta_graph('saved/nn13.meta')
    saver.restore(sess, 'saved/nn13')
    training_input, training_output, validate_input,
    ↪  validate_output, test_input, test_output = datasets
    goutput = graph.get_tensor_by_name("output:0")
    ginput = graph.get_tensor_by_name("train_input:0")
    gprob = graph.get_tensor_by_name("keepprob:0")
    gmask = graph.get_tensor_by_name("dmask:0")
    gtest = graph.get_tensor_by_name("train_test:0")
    mask = create_mask(test_input)
    basic_error = cost(goutput, gtest)

    test_input = test_input[:FLAGS.batch_size]
    test_output = test_output[:FLAGS.batch_size]

    test_accuracy = basic_error.eval(
```

```
        feed_dict={ginput: test_input, gtest: test_output, gmask
        ↪  : mask, gprob: 1.0})

    print ("Error on test set:", test_accuracy)

def generate_pdbs(datasets):
  ''' Load the saved version and write a set of PDBs of both the
  ↪  predicted
  and actual models.'''
  with tf.Session() as sess:
    graph = sess.graph
    saver = tf.train.import_meta_graph('saved/nn13.meta')
    saver.restore(sess, 'saved/nn13')
    training_input, training_output, validate_input,
    ↪  validate_output, test_input, test_output = datasets
    goutput = graph.get_tensor_by_name("output:0")
    ginput = graph.get_tensor_by_name("train_input:0")
    gmask = graph.get_tensor_by_name("dmask:0")
    gprob = graph.get_tensor_by_name("keepprob:0")
    mask = create_mask(test_input)

    midx = 0

    for k in range(0, len(test_input), FLAGS.batch_size):
      test_input_batch = test_input[k:k+FLAGS.batch_size]
      test_output_batch = test_output[k:k+FLAGS.batch_size]

      if len(test_input_batch) != FLAGS.batch_size:
        break

      res = sess.run([goutput], feed_dict={ginput:
      ↪  test_input_batch, gmask: mask, gprob: 1.0 })

      for j in range(0,len(test_input_batch)):
        torsions_real = []
        torsions_pred = []
        residues = []

        # Put the data in the correct arrays for PDB printing
        for i in range(0,len(test_input_batch[j])):
          tres = vector_to_acid(FLAGS, test_input_batch[j][i])
          if tres == "***": break
          residues.append((tres,i))
          phi = math.atan2(test_output_batch[j][i*4],
          ↪  test_output_batch[j][i*4+1])
          psi = math.atan2(test_output_batch[j][i*4+2],
          ↪  test_output_batch[j][i*4+3])
          torsions_real.append([phi,psi])
          phi = math.atan2(res[0][j][i*4], res[0][j][i*4+1])
          psi = math.atan2(res[0][j][i*4+2], res[0][j][i*4+3])
```

```
            torsions_pred.append([phi,psi])

        torsions_pred[0][0] = 0.0
        torsions_real[0][0] = 0.0
        torsions_pred[len(torsions_pred)-1][1] = 0.0
        torsions_real[len(torsions_real)-1][1] = 0.0

        from common import torsion_to_coord as tc

        mname = str(midx).zfill(3) + "_real.pdb"
        with open(mname,'w') as f:
          pf = {}
          pf["angles"] = torsions_real
          pf["residues"] = residues
          entries = tc.process(pf)
          f.write(tc.printpdb(mname, entries, residues))

        mname = str(midx).zfill(3) + "_pred.pdb"
        with open(mname,'w') as f:
          pf = {}
          pf["angles"] = torsions_pred
          pf["residues"] = residues
          entries = tc.process(pf)
          f.write(tc.printpdb(mname, entries, residues))

        mname = str(midx).zfill(3) + "_real.txt"
        with open(mname,'w') as f:
          for i in range(0, len(residues)):
            f.write(residues[i][0] + ": " +
            ↪  str(torsions_real[i][0]) + ", " +
            ↪  str(torsions_real[i][1]) + "\n")

        mname = str(midx).zfill(3) + "_pred.txt"
        with open(mname,'w') as f:
          for i in range(0, len(residues)):
            f.write(residues[i][0] + ": " +
            ↪  str(torsions_pred[i][0]) + ", " +
            ↪  str(torsions_pred[i][1]) + "\n")

        midx += 1

if __name__ == "__main__":
  from common import gen_data
  datasets = init_data_sets(FLAGS, gen_data)
  # If we just want to run the trained net
  if len(sys.argv) > 1:
    if sys.argv[1] == "-r":
      run_saved(datasets)
      sys.exit()
    if sys.argv[1] == "-e":
```

```
      print_error(datasets)
      sys.exit()
    if sys.argv[1] == "-g":
      generate_pdbs(datasets)
      sys.exit()

  graph = create_graph()
  run_session(graph, datasets)
  run_saved(datasets)
```

B.6 nn23 - LSTM with last relevant selection

```
"""
nn23.py - A bidirectional LSTM tided up
author : Benjamin Blundell
email : me@benjamin.computer

We pad out the data to the maximum, but for each input
we find the real length and both stop the LSTM unrolls
at that point (dynamic) and mask out the output layers
so the cost function doesn't take these padded values
into account.

"""

import sys, os, math, random

import tensorflow as tf
import numpy as np

# Import our shared util
parentdir =
↪  os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
os.sys.path.insert(0,parentdir)
from common.util import *
from common.batch_real import *

FLAGS = NNGlobals()
# A higher learning rate seems good as we have few examples in
↪  this data set.
# Would that be correct do we think?
FLAGS.learning_rate = 0.45
FLAGS.pickle_filename = 'pdb_martin_23.pickle'
FLAGS.lstm_size = 256   # number of neurons per LSTM cell do we
↪  think?
FLAGS.num_epochs = 2000 # number of loops around the training set
FLAGS.batch_size = 20

def weight_variable(shape, name ):
```

```
  ''' For now I use truncated normals with stdddev of 0.1.'''
  initial = tf.truncated_normal(shape, stddev=0.1, name=name)
  return tf.Variable(initial)

def bias_variable(shape, name):
  initial = tf.constant(1.0, shape=shape, name=name)
  return tf.Variable(initial)

def lstm_cell(size, kprob):
  ''' Return an LSTM Cell or other RNN type cell. We
  have a few choices. We can even throw in a bit of
  dropout if we want.'''

  cell= tf.nn.rnn_cell.BasicLSTMCell(size)
  #cell = tf.nn.rnn_cell.GRUCell(size)
  #cell = tf.nn.rnn_cell.BasicRNNCell(size)
  cell = tf.nn.rnn_cell.DropoutWrapper(cell=cell,
  ↪  output_keep_prob=kprob)
  return cell

def last_relevant(output, length):
  ''' Taken from
  ↪  https://danijar.com/variable-sequence-lengths-in-tensorflow/
  Essentially, we want the last output after the total CDR has
↪  been computed.
  That output is then converted to our 4 * max_cdr output. '''
  batch_size = tf.shape(output)[0]
  max_length = tf.shape(output)[1]
  out_size = int(output.get_shape()[2])
  index = tf.range(0, batch_size) * FLAGS.max_cdr_length +
  ↪  (length - 1)
  flat = tf.reshape(output, [-1, out_size])
  relevant = tf.gather(flat, index)
  return relevant

def create_graph() :
  graph = tf.Graph()

  with tf.device('/gpu:0'):
    with graph.as_default():
      # Input data. We take in padded CDRs but feed in a length /
      ↪  mask as well
      # Apparently the dynamic RNN thingy can cope with variable
      ↪  lengths
      # Input has to be [batch_size, max_time, ...]
      tf_train_dataset = tf.placeholder(tf.int32, [None,
      ↪  FLAGS.max_cdr_length,
      ↪  FLAGS.num_acids],name="train_input")
      output_size = FLAGS.max_cdr_length * 4
```

```
        dmask = tf.placeholder(tf.float32, [None, output_size],
        ↪  name="dmask")
        x = tf.cast(tf_train_dataset, dtype=tf.float32)

        # Since we are using dropout, we need to have a
        ↪  placeholder, so we dont set
        # dropout at validation time
        keep_prob = tf.placeholder(tf.float32, name="keepprob")

        single_rnn_cell_fw = lstm_cell(FLAGS.lstm_size, keep_prob)
        single_rnn_cell_bw = lstm_cell(FLAGS.lstm_size, keep_prob)
        # 'outputs' is a tensor of shape [batch_size,
        ↪  max_cdr_length, lstm_size]
        # 'state' is a N-tuple where N is the number of LSTMCells
        ↪  containing a
        # tf.contrib.rnn.LSTMStateTuple for each cell
        length = create_length(x)
        initial_state =
        ↪  single_rnn_cell_fw.zero_state(FLAGS.batch_size,
        ↪  dtype=tf.float32)
        initial_state =
        ↪  single_rnn_cell_bw.zero_state(FLAGS.batch_size,
        ↪  dtype=tf.float32)

        outputs, states =
        ↪  tf.nn.bidirectional_dynamic_rnn(cell_fw=single_rnn_cell_fw,
        ↪  cell_bw=single_rnn_cell_bw, inputs=x, dtype=tf.float32,
        ↪  sequence_length = length)

        output_fw, output_bw = outputs
        states_fw, states_bw = states

        output_fw = last_relevant(output_fw, length)
        output_bw = last_relevant(output_bw, length)
        output = tf.add(output_fw, output_bw)

        test = tf.placeholder(tf.float32, [None, output_size],
        ↪  name="train_test")

        # Output layer converts our LSTM to 4 outputs (4 angles)
        W_o = weight_variable([FLAGS.lstm_size, output_size],
        ↪  "weight_output")
        b_o = bias_variable([output_size],"bias_output")

        # I use tanh to bound the results between -1 and 1
        y_conv = tf.tanh( ( tf.matmul(output, W_o) + b_o) * dmask,
        ↪  name="output")
        variable_summaries(y_conv, "y_conv")

    return graph
```

```
def create_mask(batch):
  ''' create a mask for our fully connected layer, which
  is a [1] shape that is max_cdr * 4 long.'''
  mask = []
  for model in batch:
    mm = []
    for cdr in model:
      tt = 1
      if not 1 in cdr:
        tt = 0
      for i in range(0,4):
        mm.append(tt)
    mask.append(mm)
  return np.array(mask,dtype=np.float32)

def create_length(batch):
  ''' return the actual lengths of our CDR here. Taken from
  https://danijar.com/variable-sequence-lengths-in-tensorflow/
↪   '''
  used = tf.sign(tf.reduce_max(tf.abs(batch), 2))
  length = tf.reduce_sum(used, 1)
  length = tf.cast(length, tf.int32)
  return length

def cost(goutput, gtest):
  ''' Our error function which we will try to minimise'''
  # We find the absolute difference between the output angles and
  ↪  the training angles
  # Can't use cross entropy because thats all to do with
  ↪  probabilities and the like
  # Basic error of sum squares diverges to NaN due to gradient so
  ↪  I go with reduce mean
  # Values of -3.0 are the ones we ignore
  # This could go wrong as adding 3.0 to -3.0 is not numerically
  ↪  stable
  mask = tf.sign(tf.add(gtest,3.0))
  basic_error = tf.square(gtest-goutput) * mask

  # reduce mean doesnt work here as we just want the numbers
  ↪  where mask is 1
  # We work out the mean ourselves
  basic_error = tf.reduce_sum(basic_error)
  basic_error /= tf.reduce_sum(mask)
  return basic_error

def run_session(graph, datasets):
  ''' Run the session once we have a graph, training methodology
  ↪  and a dataset '''
  with tf.device('/gpu:0'):
```

```
with tf.Session(graph=graph) as sess:

  training_input, training_output, validate_input,
  ↪  validate_output, test_input, test_output = datasets
  # Pull out the bits of the graph we need
  ginput = graph.get_tensor_by_name("train_input:0")
  gtest = graph.get_tensor_by_name("train_test:0")
  goutput = graph.get_tensor_by_name("output:0")
  gmask = graph.get_tensor_by_name("dmask:0")
  gprob = graph.get_tensor_by_name("keepprob:0")

  # Working out the accuracy
  basic_error = cost(goutput, gtest)
  # Setup all the logging for tensorboard
  variable_summaries(basic_error, "Error")
  merged = tf.summary.merge_all()
  train_writer =
  ↪  tf.summary.FileWriter('./summaries/train',graph)


  #train_step =
  ↪  tf.train.GradientDescentOptimizer(FLAGS.learning_rate).minimize(basic_error)
  optimizer = tf.train.AdagradOptimizer(FLAGS.learning_rate)

  gvs = optimizer.compute_gradients(basic_error)
  capped_gvs = [(tf.clip_by_value(grad, -1., 1.), var) for
  ↪  grad, var in gvs]
  train_step = optimizer.apply_gradients(capped_gvs)

  #train_step =
  ↪  tf.train.AdamOptimizer(1e-4).minimize(basic_error)
  #train_step =
  ↪  tf.train.MomentumOptimizer(FLAGS.learning_rate,
  ↪  0.1).minimize(basic_error)

  tf.global_variables_initializer().run()
  print('Initialized')

  for i in range(0,FLAGS.num_epochs):
    stepnum = 0
    FLAGS.next_batch = 0
    print("Epoch",i)

    while has_next_batch(training_input, FLAGS):
      batch_is, batch_os = next_batch(training_input,
      ↪  training_output, FLAGS)
      batch_iv, batch_ov = random_batch(validate_input,
      ↪  validate_output, FLAGS)
```

```
            # For some reason, if the batches are not ALL the same
            ↪  size, we get a crash
            # so I reject batches smaller than the one set
            if len(batch_is) != FLAGS.batch_size or len(batch_iv)
            ↪  != FLAGS.batch_size:
              continue

            mask = create_mask(batch_is)
            summary, _ = sess.run([merged, train_step],
                feed_dict={ginput: batch_is, gtest: batch_os,
                ↪  gmask: mask, gprob: 0.8})

            if stepnum % 10 == 0:
              mask = create_mask(batch_iv)
              train_accuracy = basic_error.eval(
                feed_dict={ginput: batch_iv, gtest: batch_ov,
                ↪  gmask: mask, gprob: 1.0})

              print('step %d, training accuracy %g' % (stepnum,
              ↪  train_accuracy))

            train_writer.add_summary(summary, stepnum)
            stepnum += 1

        # save our trained net
        saver = tf.train.Saver()
        saver.save(sess, 'saved/nn23')

def run_saved(datasets):
  ''' Load the saved version and then test it against the
  ↪  validation set '''
  with tf.Session() as sess:
    graph = sess.graph
    saver = tf.train.import_meta_graph('saved/nn23.meta')
    saver.restore(sess, 'saved/nn23')
    training_input, training_output, validate_input,
    ↪  validate_output, test_input, test_output = datasets
    goutput = graph.get_tensor_by_name("output:0")
    ginput = graph.get_tensor_by_name("train_input:0")
    gmask = graph.get_tensor_by_name("dmask:0")
    gprob = graph.get_tensor_by_name("keepprob:0")
    mask = create_mask(validate_input)
    res = sess.run([goutput], feed_dict={ginput: validate_input,
    ↪  gmask: mask, gprob: 1.0})

    # Now lets output a random example and see how close it is,
    ↪  as well as working out the
    # the difference in mean values. Don't adjust the weights
    ↪  though
    r = random.randint(0, len(validate_input)-1)
```

```
    print("Actual              Predicted")
    for i in range(0,len(validate_input[r])):
      sys.stdout.write(bitmask_to_acid(FLAGS,
      ↪  validate_input[r][i]))
      phi = math.degrees(math.atan2(validate_output[r][i*4],
      ↪  validate_output[r][i*4+1]))
      psi = math.degrees(math.atan2(validate_output[r][i*4+2],
      ↪  validate_output[r][i*4+3]))
      sys.stdout.write(": " +
      ↪  "{0:<8}".format("{0:.3f}".format(phi)) + " ")
      sys.stdout.write("{0:<8}".format("{0:.3f}".format(psi)) + "
      ↪  ")
      phi = math.degrees(math.atan2(res[0][r][i*4],
      ↪  res[0][r][i*4+1]))
      psi = math.degrees(math.atan2(res[0][r][i*4+2],
      ↪  res[0][r][i*4+3]))
      sys.stdout.write(" | " +
      ↪  "{0:<8}".format("{0:.3f}".format(phi)) + " ")
      sys.stdout.write("{0:<8}".format("{0:.3f}".format(psi)))
      print("")

def print_error(datasets):
  with tf.Session() as sess:
    graph = sess.graph
    saver = tf.train.import_meta_graph('saved/nn23.meta')
    saver.restore(sess, 'saved/nn23')
    training_input, training_output, validate_input,
    ↪  validate_output, test_input, test_output = datasets
    goutput = graph.get_tensor_by_name("output:0")
    ginput = graph.get_tensor_by_name("train_input:0")
    gprob = graph.get_tensor_by_name("keepprob:0")
    gmask = graph.get_tensor_by_name("dmask:0")
    gtest = graph.get_tensor_by_name("train_test:0")

    basic_error = cost(goutput, gtest)

    test_input = test_input[:FLAGS.batch_size]
    test_output = test_output[:FLAGS.batch_size]

    mask = create_mask(test_input)
    test_accuracy = basic_error.eval(
        feed_dict={ginput: test_input, gtest: test_output, gmask
        ↪  : mask, gprob: 1.0})

    print ("Error on test set:", test_accuracy)

def generate_pdbs(datasets):
  ''' Load the saved version and write a set of PDBs of both the
  ↪  predicted
```

```
and actual models.'''
with tf.Session() as sess:
  graph = sess.graph
  saver = tf.train.import_meta_graph('saved/nn23.meta')
  saver.restore(sess, 'saved/nn23')
  training_input, training_output, validate_input,
  ↪  validate_output, test_input, test_output = datasets
  goutput = graph.get_tensor_by_name("output:0")
  ginput = graph.get_tensor_by_name("train_input:0")
  gmask = graph.get_tensor_by_name("dmask:0")
  gprob = graph.get_tensor_by_name("keepprob:0")

  midx = 0

  for k in range(0, len(test_input), FLAGS.batch_size):
    test_input_batch = test_input[k:k+FLAGS.batch_size]
    test_output_batch = test_output[k:k+FLAGS.batch_size]

    mask = create_mask(test_input_batch)
    if len(test_input_batch) != FLAGS.batch_size:
      break

    res = sess.run([goutput], feed_dict={ginput:
    ↪  test_input_batch, gmask: mask, gprob: 1.0 })

    for j in range(0,len(test_input_batch)):
      torsions_real = []
      torsions_pred = []
      residues = []

      # Put the data in the correct arrays for PDB printing
      for i in range(0,len(test_input_batch[j])):
        tres = bitmask_to_acid(FLAGS, test_input_batch[j][i])
        if tres == "***": break
        residues.append((tres,i))
        phi = math.atan2( test_output_batch[j][i*4],
        ↪  test_output_batch[j][i*4+1])
        psi = math.atan2( test_output_batch[j][i*4+2],
        ↪  test_output_batch[j][i*4+3])
        torsions_real.append([phi,psi])
        phi = math.atan2(res[0][j][i*4], res[0][j][i*4+1])
        psi = math.atan2(res[0][j][i*4+2], res[0][j][i*4+3])
        torsions_pred.append([phi,psi])

      torsions_pred[0][0] = 0.0
      torsions_real[0][0] = 0.0
      torsions_pred[len(torsions_pred)-1][1] = 0.0
      torsions_real[len(torsions_real)-1][1] = 0.0

      from common import torsion_to_coord as tc
```

```
            mname = str(midx).zfill(3) + "_real.pdb"
            with open(mname,'w') as f:
              pf = {}
              pf["angles"] = torsions_real
              pf["residues"] = residues
              entries = tc.process(pf)
              f.write(tc.printpdb(mname, entries, residues))

            mname = str(midx).zfill(3) + "_pred.pdb"
            with open(mname,'w') as f:
              pf = {}
              pf["angles"] = torsions_pred
              pf["residues"] = residues
              entries = tc.process(pf)
              f.write(tc.printpdb(mname, entries, residues))

            mname = str(midx).zfill(3) + "_real.txt"
            with open(mname,'w') as f:
              for i in range(0, len(residues)):
                f.write(residues[i][0] + ": " +
                ↪  str(torsions_real[i][0]) + ", " +
                ↪  str(torsions_real[i][1]) + "\n")

            mname = str(midx).zfill(3) + "_pred.txt"
            with open(mname,'w') as f:
              for i in range(0, len(residues)):
                f.write(residues[i][0] + ": " +
                ↪  str(torsions_pred[i][0]) + ", " +
                ↪  str(torsions_pred[i][1]) + "\n")

            midx += 1

if __name__ == "__main__":
  from common import gen_data
  # If we just want to run the trained net
  if len(sys.argv) > 1:
    if sys.argv[1] == "-r":
      datasets = init_data_sets(FLAGS, gen_data)
      run_saved(datasets)
      sys.exit()
    if sys.argv[1] == "-e":
      datasets = init_data_sets(FLAGS, gen_data)
      print_error(datasets)
      sys.exit()
    if sys.argv[1] == "-g":
      datasets = init_data_sets(FLAGS, gen_data)
      generate_pdbs(datasets)
      sys.exit()
```

```
  datasets = init_data_sets(FLAGS, gen_data)
  graph = create_graph()
  run_session(graph, datasets)
  run_saved(datasets)
```

B.7 Final version of network 23

This final version of network 23 spans several files, including many ancillary functions for generating statistics, testing various options and different training functions. We reproduce the important graph and training functions here. The entire program can be found at `https://github.com/OniDaito/MRes` and `https://doi.org/10.5281/zenodo.1319787`

```
"""
graph.py - A bidirectional LSTM graph
author : Benjamin Blundell
email : me@benjamin.computer

"""

import sys, os, math, random
import tensorflow as tf
import numpy as np

# Import our shared util - bit hacky but allows testing with
↪  __main__
if __name__ != "__main__":
  parentdir =
  ↪  os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
  os.sys.path.insert(0,parentdir)
  import common.acids as acids
  import common.batcher as batcher
  import common.settings
  from common.util_neural import *

def weight_variable(shape, name ):
  ''' For now I use truncated normals with stdddev of 0.1.'''
  initial = tf.truncated_normal(shape, stddev=0.01, name=name)
  return tf.Variable(initial)

def bias_variable(shape, name):
  initial = tf.constant(0.0, shape=shape, name=name)
  return tf.Variable(initial)

def lstm_cell(size, kprob, name):
  ''' Return an LSTM Cell or other RNN type cell. We
  have a few choices. We can even throw in a bit of
  dropout if we want.'''
```

```
  #cell = tf.nn.rnn_cell.BasicLSTMCell(size, name = name)
  #cell = tf.nn.rnn_cell.LSTMCell(size, use_peepholes = True,
  ↪  name = name)
  #wrapper = tf.contrib.rnn.LSTMBlockWrapper
  cell = tf.nn.rnn_cell.GRUCell(size) # Removed name for the CSD3
  ↪  machine
  #cell = tf.nn.rnn_cell.BasicRNNCell(size)
  cell = tf.nn.rnn_cell.DropoutWrapper(cell=cell,
  ↪  input_keep_prob=kprob, output_keep_prob=kprob)
  return cell

def last_relevant(FLAGS, output, length, name="last"):
  ''' Taken from
  ↪  https://danijar.com/variable-sequence-lengths-in-tensorflow/
  Essentially, we want the last output after the total CDR has
↪  been computed.'''
  batch_size = tf.shape(output)[0]
  max_length = tf.shape(output)[1]
  out_size = int(output.get_shape()[2])
  index = tf.range(0, batch_size) * FLAGS.max_cdr_length +
  ↪  (length - 1)
  flat = tf.reshape(output, [-1, out_size])
  relevant = tf.gather(flat, index, name=name)
  return relevant

def create_length(batch):
  ''' return the actual lengths of our CDR here. Taken from
  https://danijar.com/variable-sequence-lengths-in-tensorflow/
↪  '''
  used = tf.sign(tf.reduce_max(tf.abs(batch), 2))
  length = tf.reduce_sum(used, 1)
  length = tf.cast(length, tf.int32)
  return length

def create_graph(FLAGS) :
  graph = tf.Graph()
  with tf.device(FLAGS.device):
    with graph.as_default():
      x = None
      ww = FLAGS.num_acids

      if FLAGS.type_in == batcher.BatchTypeIn.BITFIELD:
        x = tf.placeholder(tf.float32, [None,
        ↪  FLAGS.max_cdr_length,
        ↪  FLAGS.num_acids],name="train_input")
      elif FLAGS.type_in == batcher.BatchTypeIn.FIVED or
      ↪  FLAGS.type_in == batcher.BatchTypeIn.FIVEDADD:
        ww = 5
        x = tf.placeholder(tf.float32, [None,
        ↪  FLAGS.max_cdr_length, ww],name="train_input")
```

```
elif FLAGS.type_in == batcher.BatchTypeIn.FIVEDTRIPLE:
  ww = 15
  x = tf.placeholder(tf.float32, [None,
  ↪  FLAGS.max_cdr_length, ww],name="train_input")
elif FLAGS.type_in == batcher.BatchTypeIn.BITFIELDTRIPLE:
  ww = FLAGS.num_acids * 3
  x = tf.placeholder(tf.float32, [None,
  ↪  FLAGS.max_cdr_length, ww],name="train_input")

dmask = tf.placeholder(tf.float32, [None,
↪  FLAGS.max_cdr_length, 4], name="dmask")

# Since we are using dropout, we need to have a
↪  placeholder, so we dont set
# dropout at validation time
keep_prob = tf.placeholder(tf.float32, name="keepprob")

sizes = [FLAGS.lstm_size, FLAGS.lstm_size ]

single_rnn_cell_fw = tf.contrib.rnn.MultiRNNCell(
↪  [lstm_cell(sizes[i], keep_prob, "cell_fw" + str(i)) for
↪  i in range(len(sizes))])
single_rnn_cell_bw = tf.contrib.rnn.MultiRNNCell(
↪  [lstm_cell(sizes[i], keep_prob, "cell_bw" + str(i)) for
↪  i in range(len(sizes))])

# 'outputs' is a tensor of shape [batch_size,
↪  max_cdr_length, lstm_size]
# 'state' is a N-tuple where N is the number of LSTMCells
↪  containing a
# tf.contrib.rnn.LSTMStateTuple for each cell
length = create_length(x)
initial_state =
↪  single_rnn_cell_fw.zero_state(FLAGS.batch_size,
↪  dtype=tf.float32)
initial_state =
↪  single_rnn_cell_bw.zero_state(FLAGS.batch_size,
↪  dtype=tf.float32)

outputs, states =
↪  tf.nn.bidirectional_dynamic_rnn(cell_fw=single_rnn_cell_fw,
↪  cell_bw=single_rnn_cell_bw, inputs=x, dtype=tf.float32,
↪  sequence_length = length)

output_fw, output_bw = outputs
states_fw, states_bw = states

# We can avoid the costly gather operation here by using
↪  the state
# Seems to only apply for single layer LSTMS potentially?
```

```
      output_fw = last_relevant(FLAGS, output_fw, length,
      ↪  "last_fw")
      output_bw = last_relevant(FLAGS, output_bw, length,
      ↪  "last_bw")

      output_fw = states_fw[-1]
      output_bw = states_bw[-1]

      output = tf.add(output_fw, output_bw)
      output /= 2.0

      dim = sizes[-1]

      W_f = weight_variable([dim, FLAGS.max_cdr_length * 4],
      ↪  "weight_output")
      b_f = bias_variable([FLAGS.max_cdr_length * 4],
      ↪  "bias_output")

      #y_conv = tf.nn.relu( (tf.matmul(h_drop, W_f) + b_f),
      ↪  name="output") * dmask
      output = tf.nn.tanh( (tf.matmul(output, W_f) + b_f))
      output = tf.reshape(output, [-1, FLAGS.max_cdr_length, 4],
      ↪  name="output") * dmask
      test = tf.placeholder(tf.float32, [None,
      ↪  FLAGS.max_cdr_length, 4], name="train_test")

      variable_summaries(output, "output")
      variable_summaries(output, "output_layer")

  return graph

"""
train.py - train our neural network
author : Benjamin Blundell
email : me@benjamin.computer

"""
import os, sys, math, random
import tensorflow as tf
import numpy as np

if __name__ != "__main__":
  parentdir = os.path.dirname( os.path.dirname(
  ↪  os.path.abspath(__file__)))
  os.sys.path.insert(0,parentdir)
  from common.util_neural import *
  from common import acids
  from common import batcher
  from lstm import test
```

```
def cost(goutput, gtest, FLAGS):
  ''' Our error function which we will try to minimise'''
  mask = tf.sign(tf.add(gtest,3.0))
  basic_error = tf.square(gtest-goutput) * mask

  rama_error = error_rama(goutput)
  basic_error += 0.1 * rama_error # scaling function

  basic_error = tf.reduce_sum(basic_error)
  basic_error /= tf.reduce_sum(mask)
  #basic_error += 0.001*tf.nn.l2_loss(gweights)
  return basic_error

def error_rama(goutput):
  # Take an estimate of the ramachandran plot
  (pre_phi,pre_psi)= tf.split(goutput,2,2)

  (sin_phi,cos_phi)= tf.split(pre_phi,2,2)
  (sin_psi,cos_psi)= tf.split(pre_psi,2,2)

  phi = tf.atan2(sin_phi, cos_phi)
  psi = tf.atan2(sin_psi, cos_psi)

  x = phi
  y = psi

  a = tf.maximum(tf.sin(x*1.8-45.9+y*0.26)*1.2 +
  ↪  tf.cos(y*1.8)*0.3 - (0.5 * x) -1.4 + (y * 0.1), 0)
  b = tf.maximum(tf.sin(y-0.35 + x*0.85)*0.8 + tf.cos(x-1.0)
  ↪  -1.5, 0)

  return 1.0 - tf.minimum(1.0, a+b)


def train_load(FLAGS, bt):
  with tf.control_dependencies( tf.get_collection(
  ↪  tf.GraphKeys.UPDATE_OPS)):
    #with tf.device(FLAGS.device):
    with tf.Session() as sess:
      saver = tf.train.import_meta_graph( FLAGS.save_path + "/" +
      ↪  FLAGS.save_name + '.meta')
      #saver.restore(sess, FLAGS.save_path + "/" +
      ↪  FLAGS.save_name)
      saver.restore(sess,
      ↪  tf.train.latest_checkpoint(FLAGS.save_path))
      graph = sess.graph
      _train(FLAGS, graph, bt, sess, reload=True)

def train(FLAGS, graph, bt):
```

```
  with tf.control_dependencies( tf.get_collection(
  ↪  tf.GraphKeys.UPDATE_OPS)):
    with tf.device(FLAGS.device):
      with tf.Session(graph=graph) as sess:
        _train(FLAGS, graph, bt, sess)

def _train(FLAGS, graph, bt, sess, reload=False):
  ''' Run the training session once we have a graph, training
  ↪  methodology and a dataset.
  FLAGS is the NeuralGlobals class. Graph is a tensorflow graph
↪  object and
  datasets is a tuple as received from gen_data.'''
  # Pull out the bits of the graph we need
  ginput = graph.get_tensor_by_name("train_input:0")
  gtest = graph.get_tensor_by_name("train_test:0")
  goutput = graph.get_tensor_by_name("output:0")
  gmask = graph.get_tensor_by_name("dmask:0")
  gprob = graph.get_tensor_by_name("keepprob:0")

  # Working out the accuracy
  basic_error = None
  glabel = None
  gpred = None
  basic_error = cost(goutput, gtest, FLAGS)

  l1_regularizer = tf.contrib.layers.l1_regularizer(scale=0.001,
  ↪  scope=None)
  weights = tf.trainable_variables() # all vars of your graph
  print(weights)
  #ll = [greg0, greg1, greg2, greg3]
  #regularization_penalty =
  ↪  tf.contrib.layers.apply_regularization(l1_regularizer,
  ↪  [gweights, gweights2, gweights3])
  #regularization_penalty =
  ↪  tf.contrib.layers.apply_regularization(l1_regularizer, ll)
  regularization_penalty =
  ↪  tf.contrib.layers.apply_regularization(l1_regularizer,
  ↪  weights)
  regularized_loss = basic_error + regularization_penalty # this
  ↪  loss needs to be minimized


  # Setup all the logging for tensorboard
  esum = tf.summary.scalar("TrainingError",basic_error)
  vsum = tf.summary.scalar("ValidationError", basic_error)
  train_writer = tf.summary.FileWriter(FLAGS.save_path +
  ↪  '/summaries/train',graph)
  merged = tf.summary.merge_all()

  train_step = None
```

```
if not reload:

  # There is a bug here. If we don't add optimizer.name this
  ↪  won't serialise
  optimizer = tf.train.AdamOptimizer(learning_rate =
  ↪  FLAGS.learning_rate, name="BUG")
  optimizer.name = "BUG"

  #train_step = optimizer.minimize(regularized_loss)
  #train_step = optimizer.minimize(basic_error,
  ↪  global_step=global_step)
  #train_step = optimizer.minimize(basic_error)

  # Gradient clipping - stops things from exploding
  gvs = optimizer.compute_gradients(basic_error)
  #gvs = optimizer.compute_gradients(regularized_loss)
  capped_gvs = [(tf.clip_by_value(grad, -1., 1.), var) for
  ↪  grad, var in gvs]
  train_step = optimizer.apply_gradients(capped_gvs)

  tf.global_variables_initializer().run()
  tf.add_to_collection("optimizer", optimizer)
else:
  train_step = tf.get_collection("optimizer")[0]

print('Initialized')

print("goutput shape", goutput.shape)

# The actual running
total_steps = 0
error_history = []
eval_limit = 50
lowest = 5.0

for epoch in range(0,FLAGS.num_epochs):
  stepnum = 0
  bt.reset()

  while bt.has_next_batch_random(batcher.SetType.TRAIN):
    (batch_is, batch_os, loop_t) =
    ↪  bt.next_batch(batcher.SetType.TRAIN, randset=True )
    (batch_iv, batch_ov, loop_v) =
    ↪  bt.random_batch(batcher.SetType.VALIDATE)

    # For some reason, if the batches are not ALL the same
    ↪  size, we get a crash
    # so I reject batches smaller than the one set
```

```
if len(batch_is) != FLAGS.batch_size or len(batch_iv) !=
↪  FLAGS.batch_size:
  continue

mask = bt.create_mask(batch_is)
feed_dict = {ginput: batch_is, gtest: batch_os, gmask:
↪  mask, gprob: FLAGS.dropout}
summary, _ = sess.run([merged, train_step], feed_dict=
↪  feed_dict)

# Evaluate and print the error, based on the validation set
# We also record the error and perform an early stop and
↪  save
if stepnum % eval_limit == 0:
  mask = bt.create_mask(batch_iv)
  feed_dict = {ginput: batch_iv, gtest: batch_ov, gmask:
  ↪  mask, gprob: 1.0}

  validation_accuracy, validation_sum =
  ↪  sess.run([basic_error, vsum], feed_dict=feed_dict)

  mask = bt.create_mask(batch_is)
  feed_dict = {ginput: batch_is, gtest: batch_os, gmask:
  ↪  mask, gprob: 1.0}

  train_accuracy, train_sum = sess.run([basic_error, esum
  ↪  ], feed_dict=feed_dict)

  train_writer.add_summary(validation_sum, total_steps)
  train_writer.add_summary(train_sum, total_steps)

  print('epoch %d, step %d, training accuracy %g,
  ↪  validation accuracy %g' % (epoch, stepnum,
  ↪  train_accuracy, validation_accuracy))

  if validation_accuracy < lowest:
    lowest = validation_accuracy
    saver = tf.train.Saver()
    saver.save(sess, FLAGS.save_path + "/" + str(epoch) +
    ↪  "_best_" + FLAGS.save_name)

  eval_limit = max(1, 5 *
  ↪  int(math.floor(validation_accuracy /
  ↪  FLAGS.absolute_error)))

  error_history.append(validation_accuracy)
  if len(error_history) > FLAGS.error_window:
    error_history = error_history[1:]

train_writer.add_summary(summary, total_steps)
```

```
      stepnum += 1
      total_steps += 1
      global_step = total_steps

      if stepnum % eval_limit == 0:
        # Run a quick test
        test.predict(FLAGS, sess, graph, bt)

        # Test for early quit
        if len(error_history) == FLAGS.error_window:
          diff = 0
          for i in range(1, FLAGS.error_window):
            diff += math.fabs(error_history[i] -
            ↪  error_history[i-1])
          diff /= FLAGS.error_window
          print("Diff", diff, "Err:", error_history[-1])
          if diff <= FLAGS.error_delta and error_history[-1] <
          ↪  FLAGS.absolute_error:
            print("Low error reached. Saving at epoch:", epoch)
            saver = tf.train.Saver()
            saver.save(sess, FLAGS.save_path + "/" + str(epoch) +
            ↪  "_" + FLAGS.save_name)
            sys.exit()

    # Save a version each epoch
    saver = tf.train.Saver()
    saver.save(sess, FLAGS.save_path + "/" + FLAGS.save_name)
```

B.8 Labelling network

The labelling network shares sections with the final network, so only the graph code is shown here. The entire program can be found at `https://github.com/OniDaito/MRes` and `https://doi.org/10.5281/zenodo.1319787`

```
"""
graph.py - Labelling network
author : Benjamin Blundell
email : me@benjamin.computer

"""

import sys, os, math, random
import tensorflow as tf
import numpy as np

# Import our shared util - bit hacky but allows testing with
↪  __main__
if __name__ != "__main__":
  parentdir =
  ↪  os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

```
  os.sys.path.insert(0,parentdir)
  import common.acids as acids
  import common.batcher as batcher
  import common.settings
  from common.util_neural import *

def weight_variable(shape, name ):
  ''' For now I use truncated normals with stdddev of 0.1.'''
  initial = tf.truncated_normal(shape, stddev=0.01, name=name)
  return tf.Variable(initial)

def bias_variable(shape, name):
  initial = tf.constant(0.0, shape=shape, name=name)
  return tf.Variable(initial)

def lstm_cell(size, kprob, name):
  ''' Return an LSTM Cell or other RNN type cell. We
  have a few choices. We can even throw in a bit of
  dropout if we want.'''

  #cell = tf.nn.rnn_cell.BasicLSTMCell(size, name = name)
  #cell = tf.nn.rnn_cell.LSTMCell(size, use_peepholes = True,
  ↪  name = name, activation=tf.nn.elu)
  #wrapper = tf.contrib.rnn.LSTMBlockWrapper
  cell = tf.nn.rnn_cell.GRUCell(size, name=name)
  #cell = tf.nn.rnn_cell.BasicRNNCell(size)
  cell = tf.nn.rnn_cell.DropoutWrapper(cell=cell,
  ↪  input_keep_prob=kprob, output_keep_prob=kprob)
  return cell

def last_relevant(FLAGS, output, length, name="last"):
  ''' Taken from
  ↪  https://danijar.com/variable-sequence-lengths-in-tensorflow/
  Essentially, we want the last output after the total CDR has
↪  been computed.'''
  batch_size = tf.shape(output)[0]
  max_length = tf.shape(output)[1]
  out_size = int(output.get_shape()[2])
  index = tf.range(0, batch_size) * FLAGS.max_cdr_length +
  ↪  (length - 1)
  flat = tf.reshape(output, [-1, out_size])
  relevant = tf.gather(flat, index, name=name)
  return relevant

def create_length(batch):
  ''' return the actual lengths of our CDR here. Taken from
  https://danijar.com/variable-sequence-lengths-in-tensorflow/
↪  '''
  used = tf.sign(tf.reduce_max(tf.abs(batch), 2))
  length = tf.reduce_sum(used, 1)
```

```
  length = tf.cast(length, tf.int32, name="length")
  return length

def create_graph(FLAGS) :
  graph = tf.Graph()
  with tf.device(FLAGS.device):
    with graph.as_default():

      x = None
      ww = FLAGS.num_acids

      if FLAGS.type_in == batcher.BatchTypeIn.BITFIELD:
        x = tf.placeholder(tf.float32, [None,
        ↪  FLAGS.max_cdr_length,
        ↪  FLAGS.num_acids],name="train_input")
      elif FLAGS.type_in == batcher.BatchTypeIn.FIVED or
      ↪  FLAGS.type_in == batcher.BatchTypeIn.FIVEDADD:
        ww = 5
        x = tf.placeholder(tf.float32, [None,
        ↪  FLAGS.max_cdr_length, ww],name="train_input")
      elif FLAGS.type_in == batcher.BatchTypeIn.FIVEDTRIPLE:
        ww = 15
        x = tf.placeholder(tf.float32, [None,
        ↪  FLAGS.max_cdr_length, ww],name="train_input")
      elif FLAGS.type_in == batcher.BatchTypeIn.BITFIELDTRIPLE:
        ww = FLAGS.num_acids * 3
        x = tf.placeholder(tf.float32, [None,
        ↪  FLAGS.max_cdr_length, ww],name="train_input")

      num_classes = 36 * 36 # 10 degree divisions
      dmask = tf.placeholder(tf.float32, [None,
      ↪  FLAGS.max_cdr_length, num_classes], name="dmask")
      keep_prob = tf.placeholder(tf.float32, name="keepprob")

      sizes = [FLAGS.lstm_size,
      ↪  int(math.floor(FLAGS.lstm_size/2)),
      ↪  int(math.floor(FLAGS.lstm_size/4))]

      single_rnn_cell_fw = tf.contrib.rnn.MultiRNNCell(
      ↪  [lstm_cell(sizes[i], keep_prob, "cell_fw" + str(i)) for
      ↪  i in range(len(sizes))])
      single_rnn_cell_bw = tf.contrib.rnn.MultiRNNCell(
      ↪  [lstm_cell(sizes[i], keep_prob, "cell_bw" + str(i)) for
      ↪  i in range(len(sizes))])

      # 'outputs' is a tensor of shape [batch_size,
      ↪  max_cdr_length, lstm_size]
      # 'state' is a N-tuple where N is the number of LSTMCells
      ↪  containing a
      # tf.contrib.rnn.LSTMStateTuple for each cell
```

```
length = create_length(x)
initial_state =
↪  single_rnn_cell_fw.zero_state(FLAGS.batch_size,
↪  dtype=tf.float32)
initial_state =
↪  single_rnn_cell_bw.zero_state(FLAGS.batch_size,
↪  dtype=tf.float32)

outputs, states =
↪  tf.nn.bidirectional_dynamic_rnn(cell_fw=single_rnn_cell_fw,
↪  cell_bw=single_rnn_cell_bw, inputs=x, dtype=tf.float32,
↪  sequence_length = length)

output_fw, output_bw = outputs
states_fw, states_bw = states

# We can avoid the costly gather operation here by using
↪  the state
# Seems to only apply for single layer LSTMS potentially?
#output_fw = last_relevant(FLAGS, output_fw, length,
↪  "last_fw")
#output_bw = last_relevant(FLAGS, output_bw, length,
↪  "last_bw")

#output_fw = states_fw[-1]
#output_bw = states_bw[-1]

output = tf.concat((output_fw, output_bw), axis=2,
↪  name='bidirectional_concat_outputs')

output = tf.nn.dropout(output, keep_prob)
dim = sizes[-1] * 2

W_f = weight_variable([dim, num_classes], "weight_output")
b_f = bias_variable([num_classes], "bias_output")

# Flatten to apply same weights to all time steps.

output = tf.reshape(output, [-1, dim])
logits = tf.add(tf.matmul(output, W_f), b_f, name="logits")

prediction = tf.reshape(tf.nn.softmax(logits), [-1,
↪  FLAGS.max_cdr_length, num_classes], name="prediction")
output = tf.reshape(logits, [-1, FLAGS.max_cdr_length,
↪  num_classes], name="output")

test = tf.placeholder(tf.float32, [None,
↪  FLAGS.max_cdr_length, num_classes], name="train_test")
labels = tf.placeholder(tf.int32, [None,
↪  FLAGS.max_cdr_length], name="labels")
```

```
        variable_summaries(prediction, "output")
        variable_summaries(output, "mid_output")
        variable_summaries(W_f, "weight_output")
        variable_summaries(b_f, "bias_output")

    return graph
```

Appendix C

5D amino acid vectors

Acid	0	1	2	3	4
ALA	0.189	-3.989	1.989	0.14	1.009
ARG	5.007	0.834	-2.709	-2.027	3.696
ASN	7.616	0.943	0.101	3.308	0.207
ASP	7.781	0.03	1.821	1.376	-3.442
CYS	-5.929	-4.837	6.206	2.884	5.365
GLN	5.48	1.293	-3.091	-2.348	1.628
GLU	7.444	1.005	-2.121	-1.307	-1.011
GLY	4.096	0.772	7.12	0.211	-1.744
HIS	3.488	6.754	-2.703	4.989	0.452
ILE	-7.883	-4.9	-2.23	0.99	-2.316
LEU	-7.582	-3.724	-2.74	-0.736	-0.208
LYS	5.665	-0.166	-2.643	-2.808	2.474
MET	-5.2	-2.547	-3.561	-1.73	0.859
PHE	-8.681	4.397	-0.732	1.883	-1.987
PRO	4.281	-2.932	2.319	-3.269	-4.451
SER	4.201	-1.948	1.453	1.226	1.014
THR	0.774	-3.192	0.666	0.07	0.407
TRP	-8.492	9.958	4.874	-5.288	0.672
TYR	-6.147	7.59	-2.065	2.413	-0.562
VAL	-6.108	-5.341	-1.95	0.025	-2.062

Table C.0.1: Five-dimensional values for each amino acid, from Li and Koehl[39]