Aaron Lok Hin Chan
A0117352M

CS5242: Neural Networks and Deep Learning

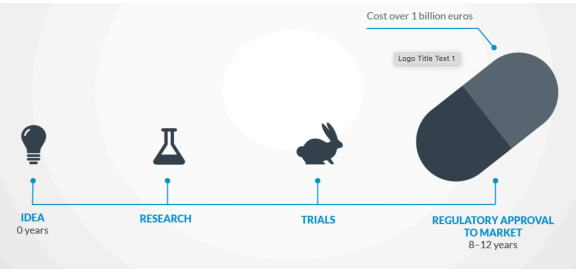
Drug Discovery Project Report

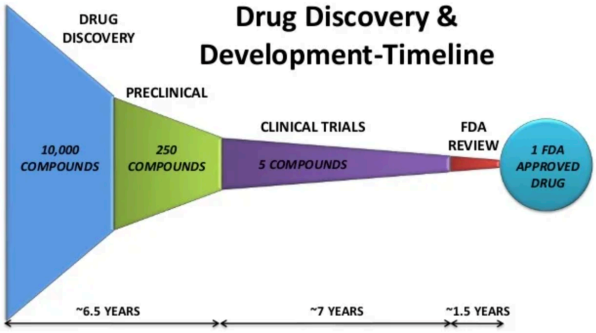
Table of Contents

Aaron Lok Hin Chan
A0117352M

Section #1: Background, Introduction, Motivation

The motivation for this research project stems from the challenges faced today in the bio-medicine and pharmaceutical field. The process of finding new drugs that cure a wide range of diseases has always been a key focus area in sciences and humanity. To give a brief sense of what the status quo looks like in the current drug discovery pipeline, drugs traditionally go through a pipeline from research inception to FDA approval and production over a span of around 12 years, with an average cost of $2.6 billion (Raval, 2018). A primary reason for this is attributed to the immense time requirements needed to test out all possible protein-ligand combinations in order to find a suitable candidate for an approved drug. After going through the entire process, it is said that "90% of all clinical trials still fail in humans trials, even after the drugs have been successfully tested in animals" (Raval, 2018).





These traditional processes can be described as knowledge-based, which not only require an immense amount of time in itself, but also generally rely on an exclusive set of knowledge that only a small circle of highly-trained medical professionals are privy to, in order to form new insights or make meaningful contributions. Even with new knowledge gained in bio-medicine, the rate at which we discover new drugs may only improve incrementally, simply due to the realistic limitations detailed. All in all, the amount of research work, money, and time that goes into drug discovery is astronomical.

Potentials of Deep Learning:

With these existing challenges and obstacles in mind, deep learning offers a fundamentally new approach in tackling this age-old problem. Although, domain knowledge will no doubt still be critical in this field, artificial intelligence and neural networks, in particular, offer to at least take the grunt work of mindless trial-and-error out of the equation, and narrow down the list of candidates of drugs in much less time than previously deemed possible.

With recent advances in the deep learning field and also computational power, neural networks have the potential ability to recognize and predict the binding affinities of a particular protein-ligand pair. In other words, neural networks offer to be able to predict, at a relatively high accuracy, a subset of ligands that can act as potential drugs to common diseases - all at a significantly lower amount of human effort as compared to traditional knowledge-based approaches.

This is one of the advantages of using neural networks' 'black-box' approach. Due to the mathematical underpinnings involved – forward propagation, backward propagation, and gradient descent – the practitioner simply needs to feed in a relevant set of features into a model, let the model calculate the loss according to a defined loss function, and allow it to iteratively learn its parameters (weights and biases) through the process of backpropagation. This process repeats itself until the model reaches a local or global minimum point in the loss curve through gradient descent.

In the context of medicine, enormous amounts of time spent on research and testing, as well as technical domain knowledge in biomedicine and pharmacology, are no longer a baseline requirement to contribute to the field of drug discovery. All that is required is someone with the knowledge of deep learning techniques and access to large amounts of pharmaceutical data to input a set of features into a neural network and train it over time to recognize what ligands can bind effectively to which type of proteins.

Aaron Lok Hin Chan
A0117352M

Report Structure:

In this report, we will explore in detail how we can use deep learning techniques to train a network with available data to identify possible ligand-protein pairings that can potentially go further into the drug discovery pipeline. More specifically, we will look at data that is presented to us, which includes protein data in the form of Protein Data Bank files (PDB), which is a textual file format describing 3d structures of molecules; ligand data in the form of Simplified molecular-input line-entry ("SMILES"), which is a line notation that represents the chemical make-up of a ligand; and lastly, the centroid position in the form of x, y, z coordinates that describe the precise docking location on a protein that a ligand must bind into. The report will also go through the general pipeline used to clean and process the mentioned data, and specific methods, techniques, and hyper-parameters chosen in constructing the network model. Lastly, we will look at the training results, assumptions made, and lessons learned throughout the project.

Aaron Lok Hin Chan
A0117352M

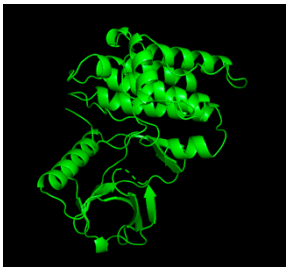
Section #2: Methods

Raw Data:

The raw data for our project includes:

- Protein data on 3000 proteins: in the form of its Protein ID (PID), and its associated Protein Data Bank file (PDB), which is a textual file format describing 3d structures of molecules. The file contains data on an atomic level, including type of atom, positional coordinates of the atom (x, y, z), and the amino acid group in which the atom is a part of.
- Ligand data of 3431 ligands: in the form of its Ligand ID (LID), and its simplified molecular-input line-entry ("SMILE's"), which is a line notation that represents the chemical make-up of a ligand.
- Centroid position that describes the precise docking location on a protein that a ligand must bind to, in the form of x, y, z coordinates.
- 3000 Protein-Ligand Pairs, which acts as the ground-truth label, in which we will feed into the network as 'correct' labels in order for it to learn its weights and biases.

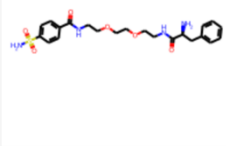
Initial Research and Exploratory Data Analysis:



We made use of PyMol to visualize the proteins in 3D and interactive format. A 3D representation of protein with PID "5WNK" in PyMol shown below:

The RDKit library had a wealth of information that could be extracted on proteins. Some possible features we could have used include:

- Number of atoms: Using "one_mol_object.GetNumAtoms()" can retrieve a count of atoms inside a SMILE string.
- Visual representation of protein and ligand:

LID	Smiles	ROMol
1	S(=O)(=O)(N)c1ccc(cc1)C(=O)NCCOCCOCCNC(=O)[C@@...	

DeepDTAF's Smile Encoder offered a simple label-encoding dictionary for the SMILE strings, which encodes each character in the SMILE string into a number from 1 to 64:

```
CHAR_SMI_SET = {"(": 1, ".": 2, "0": 3, "2": 4, "4": 5, "6": 6, "8": 7, "@": 8,
                "B": 9, "D": 10, "F": 11, "H": 12, "L": 13, "N": 14, "P": 15, "R": 16,
                "T": 17, "V": 18, "Z": 19, "\\": 20, "b": 21, "d": 22, "f": 23, "h": 24,
                "l": 25, "n": 26, "r": 27, "t": 28, "#": 29, "%": 30, ")": 31, "+": 32,
                "-": 33, "/": 34, "1": 35, "3": 36, "5": 37, "7": 38, "9": 39, "=": 40,
                "A": 41, "C": 42, "E": 43, "G": 44, "I": 45, "K": 46, "M": 47, "O": 48,
                "S": 49, "U": 50, "W": 51, "Y": 52, "[": 53, "]": 54, "a": 55, "c": 56,
                "e": 57, "g": 58, "i": 59, "m": 60, "o": 61, "s": 62, "u": 63, "y": 64}
```

Data Set Refinement Steps:

Protein PDB Files:

Pre-processing: first we read in each of the PDB files from the PDB directory folder into a Python dictionary, where a key in the dictionary represents each protein, and inside each protein value is a list of its associated elements.

1. PDB Dictionary Object ("pdb_dict"):
    a. X List, Y List Z List, Atom Type List, Amino Acid List, Amino Acid Name List

```
pdb_dict['1DIA'].keys()
```

```
dict_keys(['X_list', 'Y_list', 'Z_list', 'atomtype_list', 'aminoacid_list', 'aminoacidname_list'])
```

Feature extraction:

We use some basic domain knowledge to make a few assumptions, in order to extract useful features from the processed PDB data, and convert into a Pandas DataFrame object that contains the protein information. The Pandas DataFrame format will make the process easier for analysis and feature extraction.

Atom → Polar / Hydrophobic:

The first assumption involved classifying the atom based on their properties. Instead of inputting the atoms at face value – eg. Nitrogen, Oxygen, Hydrogen, Sulphur. We classified them into 2 groups – Hydrophobic and Polar.

List of distinct atoms from all proteins:

```
all_atoms = ['C', 'D', 'N', 'O', 'P', 'S']
```

By doing so, we can abstract out the lower-level details of the atom itself, and put more emphasis on its polar/hydrophobic attribute. This assumes that an atom's polar/hydrophobic nature has an impact on the protein-ligand binding affinity, and that one polar atom has a similar effect on the result as another polar atom, for example. The network should thus be able to formulate patterns more quickly than if each atom is represented at its atomic level.

Atoms → Amino Acid Grouping:

The second assumption also required a little bit of domain knowledge, as we make the assumption that grouping our protein inputs into amino acids would yield better performance as individually feeding in data on each atom. Again, the logic follows that the network would not be able to formulate patterns as easily if each row of our data contained a single atom. Grouping them into amino acids also makes sense because each protein contains thousands of atoms, which would create a significantly larger array for each input sample, and would cause issues when feeding in the full set of data. Grouping atoms into amino acids reduces the number of rows needed from thousands down to hundreds, per protein on average. For the positional coordinates, we will only take the coordinates of the first atom per amino acid. Afterwards, we will simply add a count of hydrophobic and polar atoms separately to each amino acid. By doing so, we assume that the specific ordering of atoms inside the amino acid will not play a large role in determining protein-ligand binding, and that the count of each type of atom is sufficient in forming patterns.

List of all distinct amino acids present in protein data (19):

```
: array(['PRO', 'ILE', 'ALA', 'THR', 'TYR', 'VAL', 'LYS', 'GLU', 'TRP',
         'LEU', 'GLN', 'ARG', 'HIS', 'GLY', 'ASP', 'PHE', 'SER', 'ASN',
         'CYS'], dtype=object)
```

One-Hot-Encoding:

Lastly, instead of simply having the name of protein, or label-encoded protein, inside the dataset, we should one-hot-encode the data instead. This approach is preferred over label-encoding since numerical labels will introduce an unwanted ordinal effect on the network. In other words, our model may form a correlation between an amino acid with a larger numerical label and one with a smaller numerical value, which is unwanted. Since there is no numerical correlation between each amino acid, the preferred approach is to one-hot-encode the data.

Final Protein DataFrame:

	X	Y	Z	hydrophobic	polar	PID	ALA	ARG	ASN	ASP	CYS	GLN	GLU	GLY	HIS	ILE	LEU	LYS	PHE	PRO	SER	THR	TRP	TYR	VA
1	3.271	14.781	5.265	3	3	1O4J	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.(
2	4.175	17.075	7.466	6	2	1O4J	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.(
3	2.916	19.336	8.256	5	4	1O4J	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.(
4	0.377	18.587	8.674	3	2	1O4J	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.(

Aaron Lok Hin Chan
A0117352M

Ligands SMILE's data:

Data Set Refinement Steps:

First, we read in all the remaining data sets and merge into one DataFrame object, which includes 'Pairs', 'Ligands', and 'Centroids' files.

	PID	LID	x	y	z	Smiles
0	102D	494	9.819391	24.178348	71.561739	c1(ccc(cc1)C(=N)N)OCCCOc1ccc(cc1)C(=N)N
1	110M	1797	35.189667	6.802667	12.175667	C=NC
2	112M	732	34.892200	7.174000	12.498400	C=NCCC
3	11BA	1313	-14.688256	14.944487	0.193744	n1(c(=O)[nH]c(=O)cc1)[C@@H]1O[C@H](CO)[C@@H](O...

Next, we used DeepTAF's Smile Encoder dictionary to create a SMILE-encoded field, which we then used one-hot-encoding and a count function to aggregate the number of times each SMILE-encoded character appeared in the ligand's SMILE string.

Negative Data:

If we only show data with positive labels to our model, it would have an inherit bias towards only predicting positive values, since it will have never seen negative data during training. Hence we need to add data samples with a negative label, or "wrong" pairing.

In order to generate sufficient negative data to properly train our model, we build a random PID-generator function that randomly selects a "PID" that is **not** the actual paired "PID" from the full list, and then add a column for the "wrong PID" for each row in the DataFrame. We will simply have a 1:1 ratio of positive to negative labelled data, which results in a total dataset of 6000 data samples.

Merge Protein and Ligand Data:

Lastly, we merge the two DataFrame objects (Ligands and Proteins) into one, and impute any 'nan' values with zeros. The resulting DataFrame has the following 84 columns, which will act as the final features in our input dataset.

```
1 final_df.columns
```

```
Index(['PID', 'LID', 'x', 'y', 'z', 55, 34, 0, 30, 41, 39, 13, 47, 25, 52, 11, 53, 7, 14, 3, 48, 33, 60, 10, 44, 19,
35, 24, 50, 4, 36, 46, 57, 1, 56, 5, 61, 8, 26, 17, 62, 20, 28, 16, 58, 40, 32, 15, 18, 64, 31, 'X', 'Y', 'Z', 'hydro
phobic', 'polar', 'ALA', 'ARG', 'ASN', 'ASP', 'CYS', 'GLN', 'GLU', 'GLY', 'HIS', 'ILE', 'LEU', 'LYS', 'PHE', 'PRO',
'SER', 'THR', 'TRP', 'TYR', 'VAL', 'MET', 'DA', 'DC', 'DG', 'DT', 'DU', 'A', 'C', 'G', 'U', 'UNK'], dtype='object')
```

Length of Input Sample DataFrame

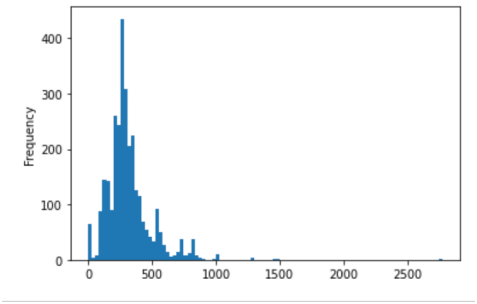
However, the challenge that remained involved the need to set a fixed length per input sample. The model required a defined shape for the input array, which was not currently the case, since the DataFrame (and later array object) has a varying number of rows that depended on the number of amino acids the associated protein contained. To illustrate this point, we can see below that the range of protein length ranged from 1 to 2774 rows, with an average of 315.

```
1 print("Min: ",final_df.groupby(['PID','LID']).size().min())
2 print("Max: ", final_df.groupby(['PID','LID']).size().max())
3 print("Mean: ", final_df.groupby(['PID','LID']).size().mean())
```

```
Min:  1
Max:  2774
Mean:  315.34991652754593
```

There are several ways to remedy this problem. We can simply choose the max amount, 2774 in this case, and simply zero-pad all other protein objects to 2774 rows in order to match the shape. However, this may not work well since most of our data will have zero values that represent useless information being fed in.

Another way is to choose a smaller number, such as the average at 315, but we will need to cut out a large portion of data from larger protein molecules. This is also not ideal since a lot of information will be lost. In deciding the optimal length, we chose 500 since it looks to be large enough to cover most (~90%) of the proteins in our dataset. The distribution of protein length is shown below.



Merge Positive and Negative Data:

After splitting up and converting each protein DataFrame into individual arrays with 500 rows each, we convert them into a 3D Numpy Array, with the final shape of (5990, 500, 84), including both positive and negative data. A few data samples were discarded initially as they contained some invalid values, such as empty coordinate values etc.

Standardization and Labels Creation:

Next, I used Sci-kit-learn's StandardScaler() to standardize all features to have a mean of 0 and a standard deviation of 1, which ensures consistency among features and that no features disproportionately carry more weight than the others. Lastly, I created a separate labels array using "np.zeroes()" and "np.ones()" that associate with each sample. The resulting shapes of the data and labels array are shown below:

```
complete_array.shape:  (5990, 500, 84)
complete_labels.shape:  (5990,)
```

All the pre-processing steps outlined above were then cleaned up and packaged into the following custom functions, which are present in the "utilities_student_1.py" file:

1. "batch_process_SMILE(ligands,centroid,PID)"
2. "process_PDB(PID,PDBS_DIR,pdb_dict)"
3. "combine_df(main_protein_df,ligand_df_merge)"

The arrays were then converted to Torch tensors, and were now ready for inputting into the neural network model.

Aaron Lok Hin Chan
A0117352M

Section #3: Presentation of Experiments

Model Type

Initially, I explored a basic multilayer perceptron model (MLP), as it could act as a model baseline for further experiments comparisons and results interpretation.

```
class MLP(nn.Module):
    def __init__(self, input_size=500*84, hidden_size=10,device='cpu'):
        super(Model, self).__init__()
        self.device = device
        self.input_size = input_size
        self.hidden_size  = hidden_size
        self.fc1 = torch.nn.Linear(self.input_size, self.hidden_size)
        self.fc2 = torch.nn.Linear(self.hidden_size, self.hidden_size)
        self.relu = torch.nn.ReLU()
        self.fc3 = torch.nn.Linear(self.hidden_size, 2)

    def forward(self, x):
        x = x.to(self.device)
        x = x.reshape(-1,500*84)
        hidden = self.relu(self.fc1(x))
        hidden = self.relu(self.fc2(hidden))
        hidden = self.relu(self.fc2(hidden))
        output = self.fc3(hidden)
        return output

    def inference(self, PID, PDBS_DIR, centroid, ligands):
        ligand_df_merge = batch_process_SMILE(ligands,centroid,PID)
        main_protein_df = process_PDB(PID,PDBS_DIR,pdb_dict)
        test_data = combine_df(main_protein_df,ligand_df_merge)
        return self.forward(test_data.float())[:,1].reshape((-1,1))
```

Multilayer Perceptron (MLP):

As seen in the code above, the MLP model I implemented contained 3 fully-connected hidden layers, with 10 neurons per layer. The activation function between each layer was ReLU; the optimizers used included SGD and Adam; and the loss function chosen was Cross-Entropy Loss.

Stochastic Gradient Descent:

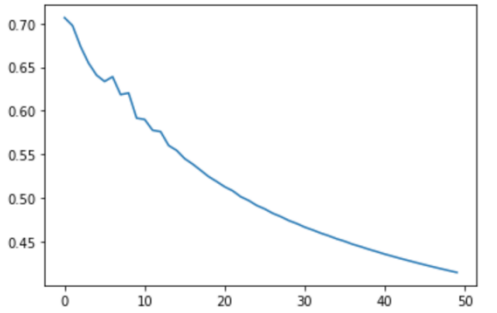
My first run of this model involved the following variables and hyperparameters:

- Optimizer: Stochastic Gradient Descent (SGD)
- Learning Rate: 0.001
- Epochs: 50
- 10 hidden neurons per layer

I plotted the train loss curve:

```
Epoch 49: train loss: 0.4147413372993469
Test loss after Training 0.40595191717147827
```

```
1 plt.plot(loss_log)
2 plt.show()
```



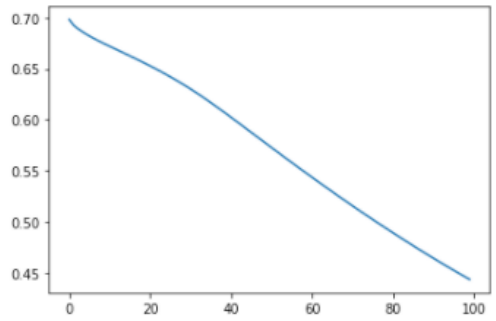
The train loss at the end of 50 epochs was 0.415, while the test loss was 0.406. The train loss seemed to be continuing to go down even after 50 epochs.

I then tried increasing the number of epochs to 100, and also increased the number of neurons per hidden layer to 20, to see what effect it would have on the train / test loss numbers.

Aaron Lok Hin Chan
A0117352M

```
Epoch 99: train loss: 0.44364213943481445
Test loss after Training 0.5431983470916748
```

```
1 import matplotlib.pyplot as plt
2 loss_log
3 plt.plot(loss_log)
4 plt.show()
```



Surprisingly, the curve became more straight, and seemingly going down at a sharper rate than at 50 epochs.

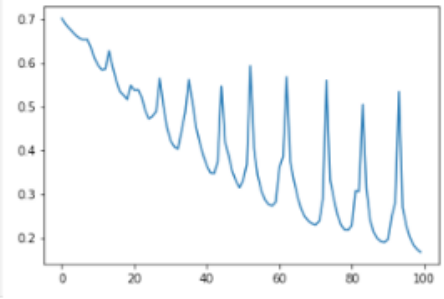
However, test performance seemed to have suffered slightly, with a test loss of 0.543.

Since it looked like the loss is still converging after 100 epochs, I decided to increase the learning rate in my model to 0.05, as a way to speed up learning.

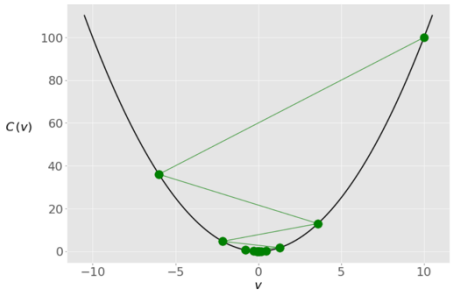
As expected, this change had a big impact on the resulting loss curve (shown on the bottom left).

```
Test loss after Training 0.36769458651542664
```

```
1 import matplotlib.pyplot as plt
2 loss_log
3 plt.plot(loss_log)
4 plt.show()
```



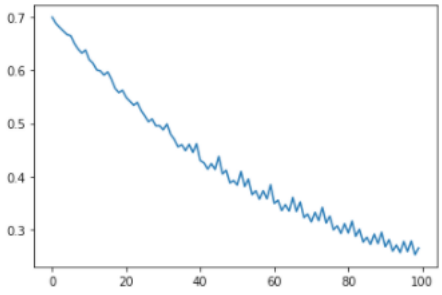
At 0.05 learning rate, the model is, in effect, jumping over the minimum point at each epoch, causing the loss to oscillate back and forth, similar to the representation below.



Reduce the learning to a middle ground of 0.03 smoothened the curve to a more acceptable shape:

```
Epoch 99: train loss: 0.2648065686225891
Test loss after Training 0.4294326603412628
```

```
1 import matplotlib.pyplot as plt
2 loss_log
3 plt.plot(loss_log)
4 plt.show()
```



As expected, the loss curve in Stochastic Gradient Descent has quite a bit of noise, since it is updating its weights and biases after each and every test sample (batch size of 1).

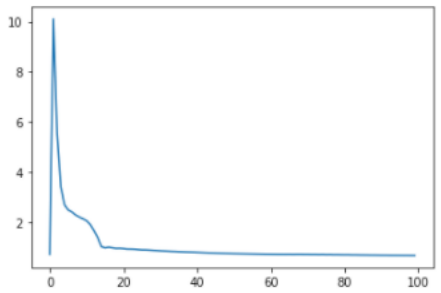
Adam:

Next, I changed the optimizer to Adam to see its effects. The below is the train loss chart using 10 hidden neurons, 2 hidden layers, 100 epochs, and a learning rate of 0.01 under Adam.

Aaron Lok Hin Chan
A0117352M

```
Epoch 99: train loss: 0.6680249571800232
Test loss after Training 2.352956771850586
```

```
1 import matplotlib.pyplot as plt
2 loss_log
3 plt.plot(loss_log)
4 plt.show()
```



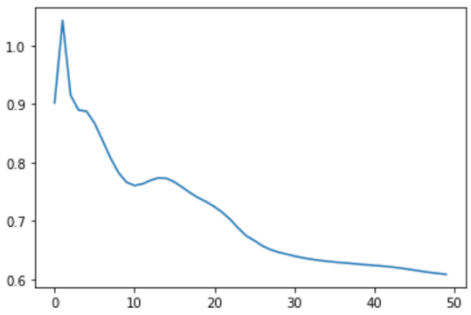
We can see the curve is much smoother than it was for SGD, perhaps due to the effects of its momentum coefficients, which take into account the values of the previous gradients in calculating the updated one. The loss also evidently converged much faster than in SGD. However, it is also important to note that the test loss after training was much higher, at 2.35, compared to 0.67 in train loss. This suggests there was a lot of overfitting in the model training phase.

To combat this, I decided to reduce the number of training epochs to 50. Surprisingly, this did improve the test loss to 0.645, significantly better than before.

```
Epoch 48: train loss: 0.6099766492843628
Epoch 49: train loss: 0.6085339188575745
Test loss after Training 0.6454832553863525
```

9]:

```
1 import matplotlib.pyplot as plt
2 loss_log
3 plt.plot(loss_log)
4 plt.show()
```

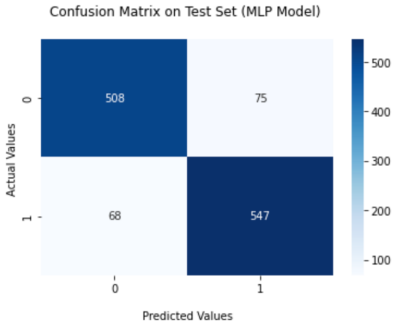


As the numbers from the train and test loss can sometimes be misleading, I looked at more direct metrics to evaluate model accuracy, such as test accuracy, and confusion matrix, before running the final project grading script in the next section.

```
1 correct = 0
2 total = 0
3 outputs = model(x_test.float())
4 labels = y_test
5 _, predicted = torch.max(outputs.data, 1)
6 total += labels.size(0)
7 correct += (predicted == labels).sum().item()
8 print(f'Accuracy of the network on the test samples: {100 * correct // total} %')
```

```
Accuracy of the network on the test samples: 88 %
```

To my surprise, the model had a test accuracy of 88%. However, on multiple runs, this number did fluctuate between 50% and the 80%. The confusion matrix gave a more detailed view of the distribution between false negatives and false positives.



Convolutional Neural Network (CNN):

Next, I decided to feed in my data into a more complex network, to compare with our baseline model.

Aaron Lok Hin Chan
A0117352M

```python
class CNN(nn.Module):
    def __init__(self, num_classes=2,device='cpu'):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=16, kernel_size=3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.conv4 = nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(16)
        self.bn2 = nn.BatchNorm2d(32)
        self.aap = nn.AdaptiveAvgPool2d((1, 1))
        self.flatten =  nn.Flatten(start_dim=1, end_dim=-1)
        self.fc = nn.Linear(32, num_classes)
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.bn1(self.relu(self.conv1(x)))
        x = self.bn1(self.relu(self.conv2(x)))
        x = self.maxpool(x)
        x = self.bn2(self.relu(self.conv3(x)))
        x = self.bn2(self.relu(self.conv4(x)))
        x = self.aap(x)
        x = self.flatten(x)
        out = self.fc(x)
        return out
```

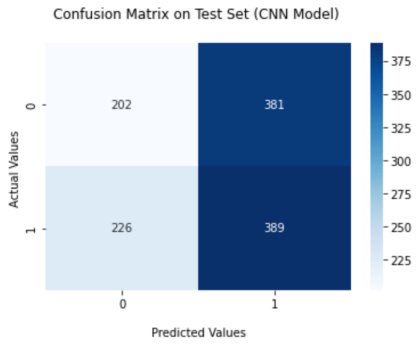
Conceptually, the immediate advantages of using a CNN are attributed to its ability to take spatial information into account, which is present in our dataset in the form of the X, Y, Z coordinates of both the amino acid location inside the protein, and the docking location for the ligand. However, I was not sure if these benefits would manifest in my data, since the X, Y, Z coordinates were not structured in the same systematic way as pixels are inside an image, which is the usual use case of a CNN. Additionally, only 6 of the total features were spatial data (X, Y, Z), whereas the remaining 78 features were either one-hot-encoded values, or sums of elements inside the protein or ligand. Nonetheless, I believed the CNN could make better inferences on the data due to the effects of layers such as batch norm, and max pool, which should theoretically make calculations faster as it reduces the tensor size.
For the optimizer, I used SGD, with a learning rate of 0.33. However, I was surprised to see that even at 1 epoch, my computer failed to run the model, and often crashed as a result. It became evident that the convolution layers were actually much more resource-intensive than the fully-connected layers in the MLP. In response, I took out half of the convolution layers that may have been causing the trouble, namely the ones that had input and output layers of 64 and 128.

At last, I was able to run the model, which outputted a train loss that hovered around 0.69 per batch. The accuracy on the test set turned out to be 48%. Although this would normally sound decent in other contexts, 48% in our model is equivalent to a random guess (~50%).

```python
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        inputs, labels = data[0].to(device), data[1].to(device)
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the test samples: {100 * correct // total} %')
```

```
Accuracy of the network on the test samples: 48 %
```



I was surprised to see a significantly lower accuracy than the simple MLP model I had tested earlier. To confirm the results, I needed to run the official “project_grading_script.py” file, which would test my model with more robust methods, including comparing the binding score of the protein with all ligands in our dataset, a much tougher test than choosing between 0 and 1.

Aaron Lok Hin Chan
A0117352M

Section #4: Discussion and Analysis

Results

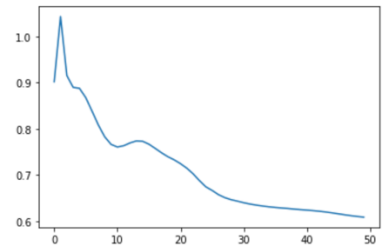
Multilayer Perceptron (MLP):

In tuning my MLP model, both Stochastic Gradient Descent and Adam algorithms performed better than I expected during training, even with a simple 3-layer model.

Under the Adam optimizer, and a learning rate of 0.01, I obtained a smooth loss curve on my training data, and a test loss of 0.645.

```
Epoch 48: train loss: 0.6099766492843628
Epoch 49: train loss: 0.6085339188575745
Test loss after Training 0.6454832553863525
```

```
9]: 1 import matplotlib.pyplot as plt
    2 loss_log
    3 plt.plot(loss_log)
    4 plt.show()
```

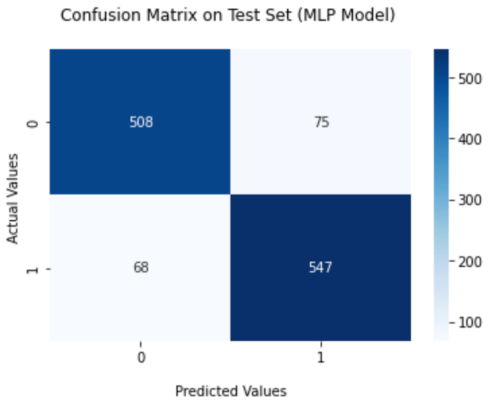


By printing out the test accuracy, I found the model produced an 88% test accuracy, which was much better than I expected. However, this number did fluctuate between 50% to 80%.

```
correct = 0
total = 0
outputs = model(x_test.float())
labels = y_test
_, predicted = torch.max(outputs.data, 1)
total += labels.size(0)
correct += (predicted == labels).sum().item()
print(f'Accuracy of the network on the test samples: {100 * correct // total} %')
```

```
Accuracy of the network on the test samples: 88 %
```

The confusion matrix showed a more detailed view of the distribution of false positive and false negative predictions.



However, I learned that these preliminary results could be misleading, as they may differ significantly from the other results, such as the metrics used in the project grading file for this project.

In short, the script takes each protein and outputs a value as a “binding score” for each of the 3000 ligands, and the result will only be considered correct if the correct ligand is amongst the top 10 protein-ligand binding scores. This presented a much tougher metric to evaluate the model on, since it was no longer simply predicting between 0 and 1.

After coding the inference method, I ran the project grading script on a handful of proteins (~20), and returned a disappointing Inference Prediction Score of 0.00000.

MLP test with 20 protein-ligand pairs that were excluded in the train set:

```
torch.Size([100, 1])
torch.Size([100, 1])
torch.Size([24, 1])
Inference Prediction Score: 0.00000.
```

The results did not change when I ran the model using different proteins or with slightly modified and trained models.

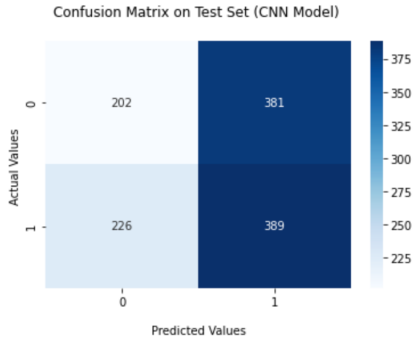
Convolutional Neural Network (CNN):

As for my CNN model, I wanted to see whether having a more complex model would garner better results. However, as seen in the last section, I did not have high expectations since the train / test accuracy scores were already low (~48% accuracy on test set).

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        inputs, labels = data[0].to(device), data[1].to(device)
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the test samples: {100 * correct // total} %')
```

```
Accuracy of the network on the test samples: 48 %
```



Hence the chances of performing well in the project grading script would be much lower. Because of the intensive amounts of CPU processing running the script took, I was only able to run the script for a few protein-ligand pairings each time. As expected, the first run did not fare very well, outputting another disappointing Inference Prediction Score of 0.00000.

CNN model inference score on 3 unseen proteins:

```
110M
torch.Size([24, 1, 500, 84])
Inference Prediction Score: 0.00000.
```

To test whether the model was working at all, I decided to try running the inference method on “seen” proteins, which means that the model should technically be able to predict it. Running the model on 5 seen proteins, I was able to get a score of 0.20000, which meant the model guessed 1 out of 5 correctly.

CNN model inference score on 5 seen proteins:

```
torch.Size([24, 1, 500, 84])
Inference Prediction Score: 0.20000.
Traceback (most recent call last):
```

Although the results were not the ones I hoped for, the experiments allowed me to see first-hand the effects on using different model types, such as MLP and CNN; optimizer algorithms such as SGD and Adam; as well as tuning of hyperparameters, such as learning rates, epochs, batch sizes, number of neurons, and number of hidden layers.

Aaron Lok Hin Chan
A0117352M

Interpretations of Results:

Initially, I was surprised that the positive results (especially in my MLP model) didn't translate in the project grading script. But upon further thought, this may not have been too much of a surprise. Predicting the correct ligand within the top-10 binding scores out of a pool of 3000 total ligands requires the model to be very sure of its results. For example, random guessing would only be correct 0.33% of the time (10/3000).

To explain the difference in results between my initial tests and the project script, the "accuracy" in my initial tests only required the model to guess between 2 values – 0 and 1. Hence even a random guess would have had a 50% accuracy rate.

```
correct = 0
total = 0
outputs = model(x_test.float())
labels = y_test
_, predicted = torch.max(outputs.data, 1)
total += labels.size(0)
correct += (predicted == labels).sum().item()
print(f'Accuracy of the network on the test samples: {100 * correct // total} %')
```

```
Accuracy of the network on the test samples: 88 %
```

Even if my model produced high accuracy of 88%, it would need to produce a very high score for the correct protein-ligand pair, while producing a much lower one for all others. In the project grading script, it is possible that even if my model predicted a good score for the correct pair, all other incorrect pairs may have also received similarly high scores, making it difficult to place the correct pair in the top 10.

Lastly, although the initial results looked promising, it could still be the case that the data was suffering from some overfitting, which resulted in the 0.0000 scores during inference.

Challenges: Domain Knowledge and Feature Engineering

Although the use of neural networks is predicated on the practitioner not needing to know the exact inner workings of how the model came up with a prediction, it became evident that domain knowledge would have helped tremendously in this project. For example, domain knowledge would definitely impact the way I interpret the raw data and thus how I choose to represent elements of the protein and ligands in the input format. Lacking this, the project often relied more on trial-and-error and guesses in the dark to try to tune out a better performance. Albeit on a smaller scale, I can see how this project reflects much of the real-life challenges in building a neural network that could churn out useful insights in long-standing industries like bio-medicine.

Ordering of Atoms and SMILEs Characters

For future iterations, we could try a more detailed representations of the protein and ligand, as I may have abstracted the structure too much in this project. For example, the ordering of the atoms inside amino acids may have mattered. Same with the ordering of SMILE's characters. By summing out these elements, we remove the knowledge of how subcomponents are connected.

Atom Attributes

Although grouping atoms into hydrophobic and polar classes seems to be a generally accepted way of classification, it also may not have had the intended effect. There may be other ways to group them that are more meaningful in the context of biomedicine, but that would require more research and domain knowledge to come to a conclusion.

Atom Location

Another approach may have involved focusing on only the amino acids in close proximity of the known binding location. Since these are the molecules that would directly interact with the ligand, there is a higher chance of predicting success. This approach makes sense intuitively, but we would need to decide on factors like what is considered "close proximity" in molecular terms. I believe this may produce better results, since it removes much of the unneeded elements that may have acted as noise in our model.

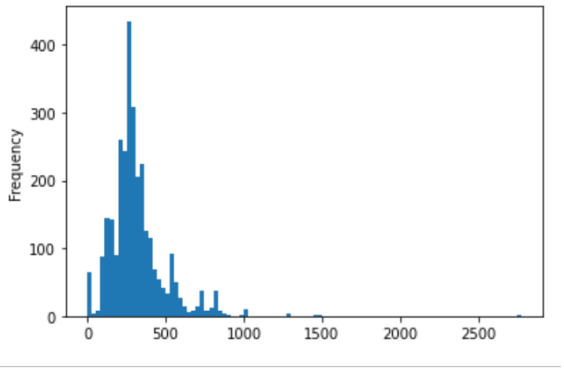
Aaron Lok Hin Chan
A0117352M

Data Shape

Deciding on the data shape was also a challenge, since we needed both the protein and ligand objects to have the same length. In our example, the length of each array followed the protein, which had each row representing an amino acid, whereas all of the ligand's array was represented in one row only. As a result, the ligand data needed to repeat for the number of rows in the protein, which ranged from 1 to 500.

The max limit of 500 was also set arbitrarily. It could be possible that 500 was already too high. Since the average row length was 315, most of the arrays needed to be zero-padded significantly to be consistent with the 500 row requirement. This may have negatively affected model performance since most of the data turned out to be zeros, making it difficult for the model to form interpretations.

Distribution of length of each individual protein array before processing:



Alternative Solutions

Molecular Fingerprint

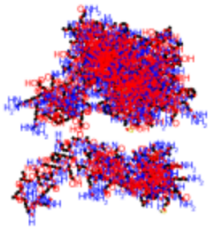
In our current model, we simply one-hot-encoded the label-encoded representation of our SMILEs string. Another possible representation may have been "molecular fingerprints", which could be converted from SMILE's using RDKit's Chem.RDKFingerprint library.

```
1.  Chem.RDKFingerprint(mol_object,maxPath=7,fpSize=512).ToBitString()
```

Moreover, there was an option to limit the size of the fingerprint, such as 512 bits. Doing this would ensure every molecule would have a consistent length, while still representing the molecule in its entirety (Sangsoo, 2021).

Image-based Approaches

Alternatively, we could try using pure image representations of proteins and ligands to feed into a CNN network, which may be more suitable than our current model. For example, this could be done using PyMol's visualization capabilities, or RDKit's functions. An image representation of a protein is shown below from RDKit:

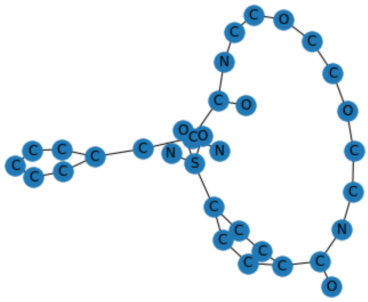


Matrix and Graph Representations:

Furthermore, there are matrix and graph representations of proteins and ligands, which may prove to be a much better approach to represent molecular makeup, since it contains the relational information of each subpart with each other, which was abstracted out in our dataset.

Visual graph representation of a ligand using PySmiles and NetworkX's "nx.to_numpy_matrix(mol)" function:

Aaron Lok Hin Chan
A0117352M



Lastly, there are many real-life nuances in the biomedical field that may either be out of our knowledge area, or simply not easily captured in input data. For example, the angles of which the ligand needs to bind to the protein may be a crucial data point, whereas we only have X, Y, Z coordinates. The binding affinity may also be a function of far more factors than the information we were given. For example, chemical factors such as the temperature or pH of the solvent used for reaction, presence of covalent bonds, hydrogen bonds, salt bridges, and Van der Walls forces may all well play a role in determining binding affinity (SciWris Life Sciences, 2020).

Conclusion

Although it is undisputed that deep learning offers a revolutionary way of approaching a problem like drug discovery, it's also evident that it is not easy to predict binding affinity purely based on X, Y, Z coordinates, and information present in ligands' SMILE's strings and proteins' PDB files. In order to garner more valuable results, we may need to extract more features, either externally or internally, and engineer and tune them in a more meaningful way, which may only be practical with more research and trial-and-error.

References

Ahmed A., Mam B., Sowdhamini R. (2017). DEELIG: A Deep Learning Approach to Predict Protein-Ligand Binding Affinity. Bioinform Biol Insights. Retrieved from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8274096/

Raval, S. (2018, April 21). AI in Medicine, Drug Discovery with GANs. Retrieved from URL https://github.com/llSourcell/AI_for_healthcare

Sangsoo L., Lu Y., Cho C.Y., Sung I., Kim J., Kim Y., Park S., Kim S. (2021). A review on compound-protein interaction prediction methods. Comput Struct Biotechnol, J. Retrieved from URL https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8008185/

SciWris Life Sciences (2020, Sept 23). Analysing Protein-Ligand Interactions. Retrieved from https://www.youtube.com/watch?v=UfU4pYvPD7w