Final Project Write-up

Project Background

Our team is a subset of a group of developers working on a social media-tangential platform. The goal of the platform is to maximize the likelihood of **meeting new people** that you will enjoy spending time with. This centralizes around **connecting** users based on the basic information that they give us. The goal of our project was to construct a **Neural Network** that could calculate affinities for our users, so we would know which users were most **affinitive** to a **central user**. For the scope of our project, we assumed that users would have:

1. Hobbies

   Users have interests - Soccer, Bowling, Hiking, and Knitting. If users share common interests, that makes them more compatible as potential friends!

2. Friends

   Users can mark other users as friends of theirs. If users have common friends or are friends, they are more affinitive.

3. Buddies

   Users can mark one other user as their "buddy," who meets new people with them. If a user is another user buddy, they are presumed to be 100% compatible and are given a very high-affinity score.

4. Groups

   Users on the platform are part of groups, and since the platform's goal is to maximize meeting new people, users are less affinitive if they've already shared groups.

5. Demographic information

   For our current scope, we limited ourselves to age - this is an important factor when meeting new people, as most people would like to meet others of a similar age range. Therefore, age difference was a key factor in the affinity calculation.

Neural Network Construction

The goal of the network is to take our central user and learn to calculate affinities from other users (u_i) on our platform to that central user (u_c). We therefore had to construct some set of features about a user that could be fed into the first layer of neurons.

Features of a User

We used the following list of features:

1. Age difference

$$|\text{age}(u_c) - \text{age}(u_i)|$$

2. Are they buddies?

$$1 \text{ if yes, } 0 \text{ otherwise}$$

3. Are they friends?

$$1 \text{ if yes, } 0 \text{ otherwise}$$

4. How many shared friends do they have?

   Counts distinct users that are friends of both u_c and u_i.

5. How many shared hobbies do they have?

   Counts distinct hobbies that are interests of both u_c and u_i.

6. How many groups have they been in together?

   Counts distinct groups that both u_c and u_i have been a member of.

This was our initial formulation of features - each one seemed to contribute information to the context of a user's relationship to u_c and thus we wanted it to factor into the network's affinity calculation.

Network Construction

Our initial construction of the network consisted of three layers: an input layer, an intermediary layer, and an output layer.

Since we have 6 features, our first layer of neurons takes in 6 features. Generally, as our features list adapts, our network will have its first layer take into account the "shape" of the input (this is built into the library we use - `keras`). We gave it an arbitrary 128 neurons, to begin with for the first runs. The reason we chose 128 was to give the network a chance to extract new patterns and features that take combinations of the features list.

> *Patterns that it could find might be things like the combination of two features such as being a buddy and being in a bunch of groups together (which is not a negative thing) vs. being a friend and being in a bunch of groups together (negative thing).*

The second layer consisted of 64 neurons, since we want to condense down our inputs into a single output ultimately, we want the network to cut down to the output shape before the output layer.

The final layer consists of a single neuron, to output a single numeric value (the affinity).

Other notable design decisions:

1. Optimizer function: adam

   This is standard for many simple neural networks.

2. Loss function: mean squared loss

   We chose this because our goal was to output a single number, so the loss on that number can be easily calculated via the mean squared error function.

3. Activation functions: relu for hidden layer(s), softmax for the output layer.

   This too is standard for many simple neural networks.

4. Epochs: 100

   This was a default recommended by our brief google searches.

Dataset Construction

Data ain't cheap - we don't have a running platform to pull genuine user data from, so we had to find ways to construct "realistic" data that we could use to train our model. This required two main pieces:

1. Fake User/Hobbies/Friends/Buddies/Groups data
2. Fake affinities

(2) is a lot more work than (1), because we ended up having to manually assign affinities. We constructed 1000 users, and assigned affinities manually between our central user and the 999 others. This took a lot of manual grinding, but we attempted to weight things based on the data given to us and score affinities in a way that would inform the algorithm of the priorities that stem from the features of users. We did this by printing out the features of each user and then assigning a subjective affinity to that user based purely on those features. It's a watered-down version of the interactions that will occur on the platform and therefore should serve as a functional imitation of the data the algorithm will get in practice.

Dataset Design Choices

Our central user being the only central user of the algorithm, we stunted the dataset intentionally towards more connection around the central user. This means that every group in question is a group that the central user has been a part of. Not all users on the platform are in these groups, so there are still affinities to lots of users that have not been in groups with the central user, but since our algorithm does not take into account second-degree connections via groups, we didn't bother to generate any such data. Further, the only hobbies we generated in the database are hobby categories of our central user. This is because we only take into account the hobbies in common with any other given user, so it isn't necessary to include hobbies that the central user won't have.

These shouldn't have a stunting effect on the learning of the neural network, since the types of data being omitted are not data that would factor into the current features list anyway. This may represent features that should (or will) be added later in this project, but for now, they are not there and do not seem necessary to a good affinity algorithm.

Affinity Construction

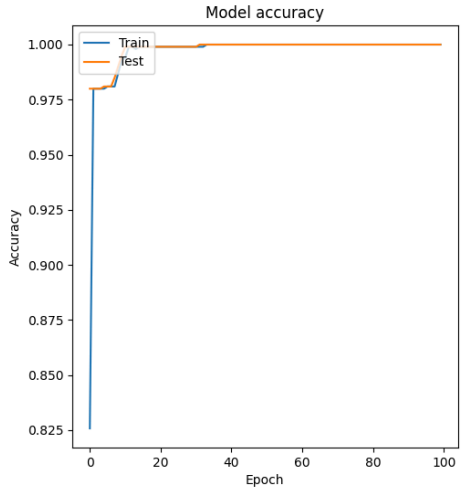
We went through several affinity generation methods. We began by testing the model on random affinities - mainly to confirm that it would run without error. Results of this are omitted, as this data in no way mimics real life. We then moved on to function-based generated affinities. Specifically, we used the following function:

$$f(u_i) = \begin{cases} 0 & \text{if } u_i \text{ is neither buddy nor friend} \\ 50 & \text{if } u_i \text{ is a friend but not a buddy} \\ 100 & \text{if } u_i \text{ is their buddy} \end{cases}$$

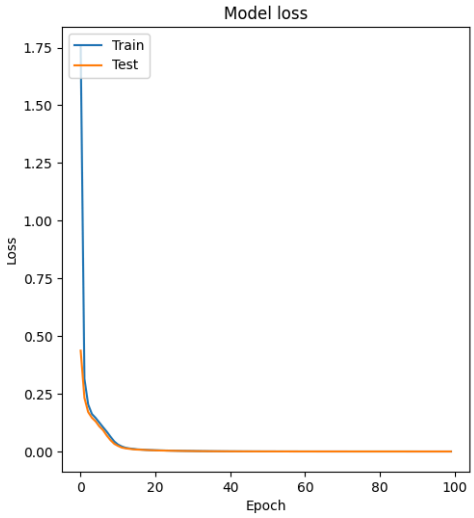
The results of this run are shown in the section below.

Finally, we turned to manual input of the affinities. We examined each user's data and gave them an affinity $a_i \in \mathbb{Z}$, $0 \leq a_i \leq 100$.

Results of the First Runs



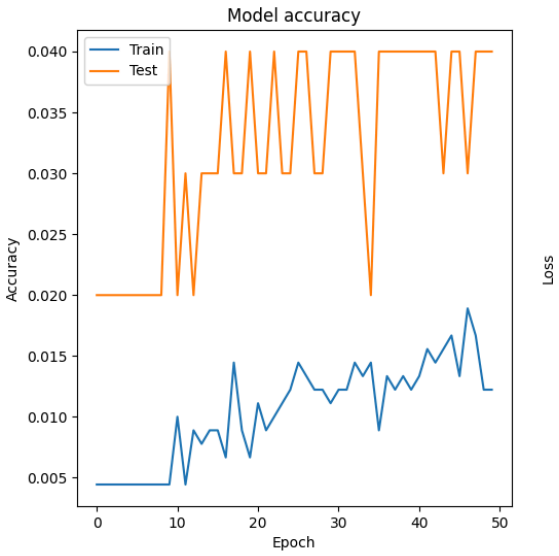
Test Accuracy: 100% (1.000)



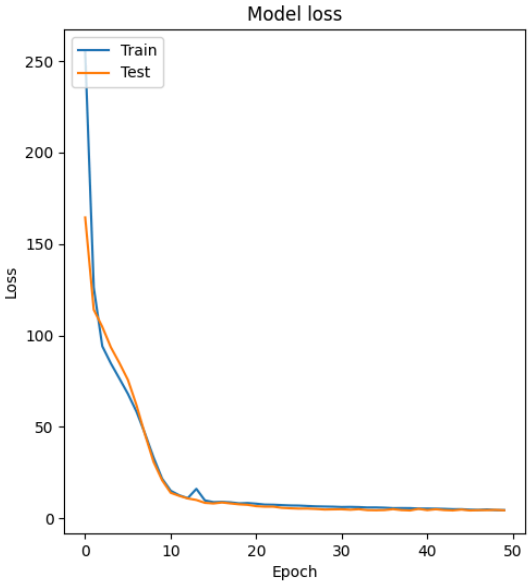
Test loss: 3.533090784912929e-05

In the above, we see the results of running our model against the function-generated affinities described in the above section. As we can see, the model almost immediately has near-perfect accuracy and eventually reaches exact accuracy to 3 decimal places. We attribute this to the fact that our function can be learned and mimicked precisely by the model. For this reason, we moved away from function-based affinity creation, into manual generation.

Second run: manual affinities



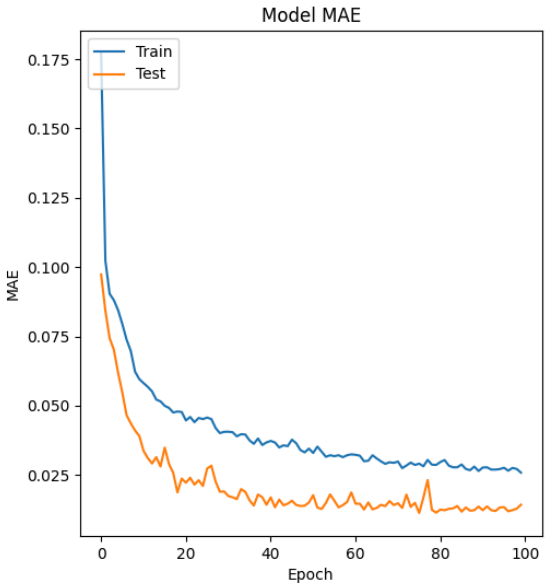
Test accuracy: 0.03999999910593033



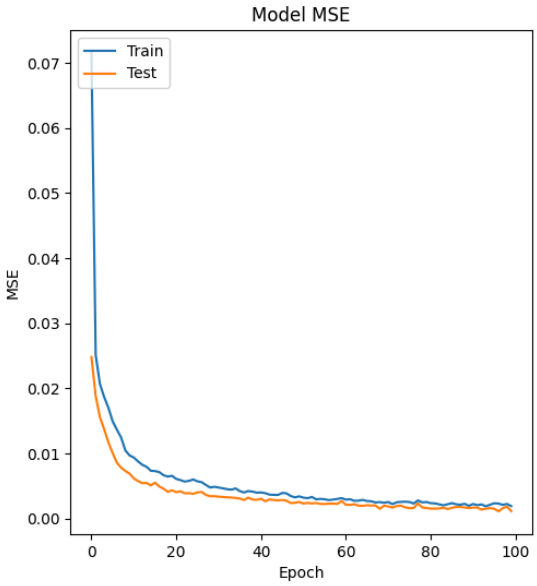
Test loss: 4.469648838043213

As we can see from the above, our loss looks good but our accuracy is all over the place. We did some research and discovered that accuracy is not a useful metric for mean squared error. We then adjusted our network to use **mean absolute error** as our loss metric.

Third run: mean absolute error

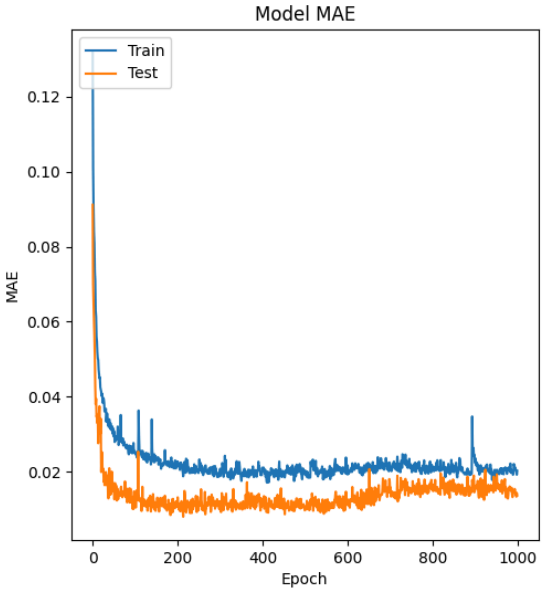


Training MAE: 0.0259, Validation MAE: 0.0143

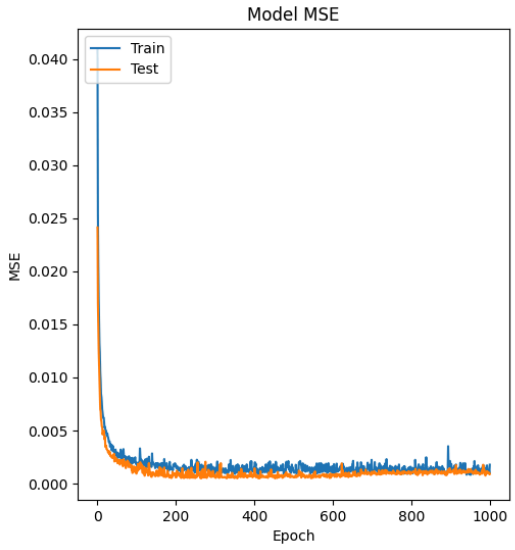


Training Loss: 0.0019, Validation Loss: 0.0011

As we can see from the above, our loss and MAE now are much more in line with each other. We have very good loss rates with only 100 epochs, but the incline had not yet plateaued. We therefore re-ran it with 1000 epochs.



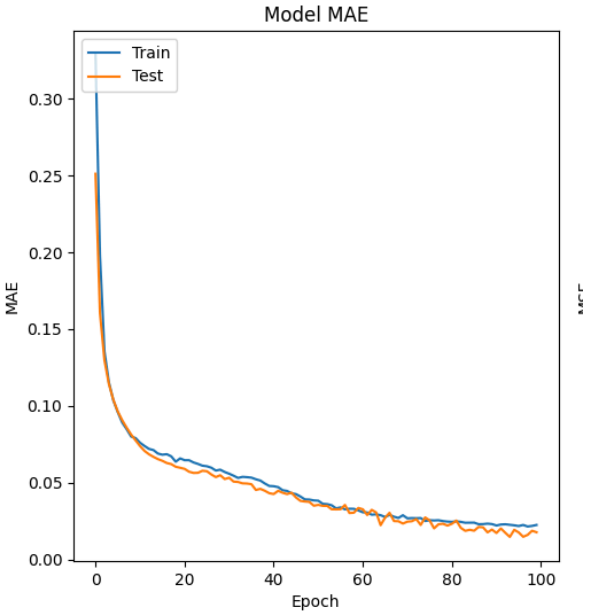
Training MAE: 0.0202, Validation MAE: 0.0136



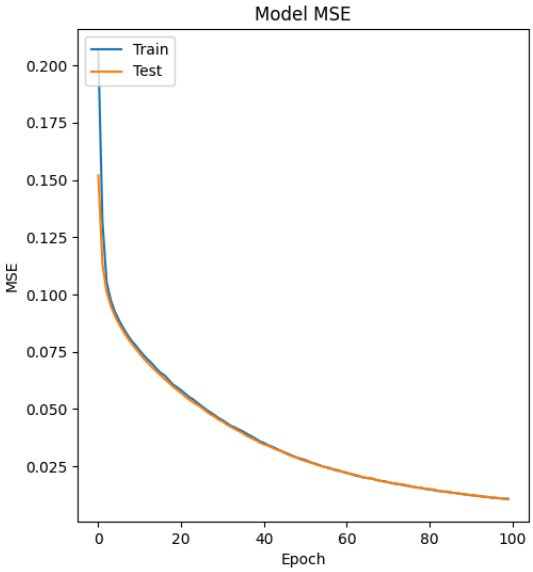
Training Loss: 0.0018, Validation Loss: 0.0009

Neural Network Tweaking

Based on more external research, we discovered that there were significant improvements to make to the structure of our network. We normalized our data to be between 0 and 1, the introduction of an additional dense layer, and a modified dropout rate to counteract overfitting. The learning rate has been fine-tuned to 0.0001, and an L2-regularizer has been applied to the first hidden layer. Finally, we replaced our softmax activation function for the output layer with a sigmoid function, which research indicates should function better.



Training MAE: 0.0225, Validation MAE: 0.0177



Training Loss: 0.0109, Validation Loss: 0.0107

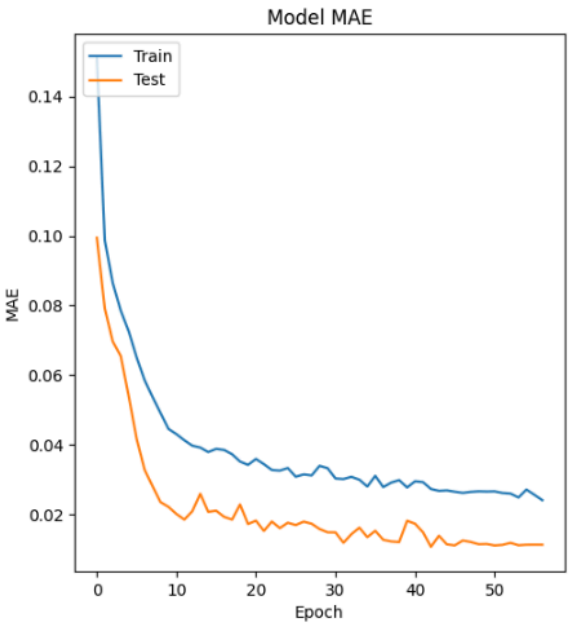
From the above diagrams, we can see much smoother paths to low losses, which is what we wanted. However, we had some other methods to try in order to improve our network's ability to learn.

Final Neural Network Tweaks

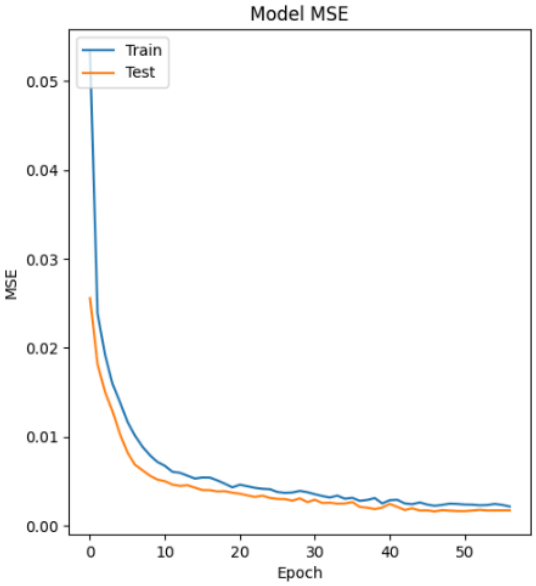
Our model was already performing quite well. However, we had some ideas for modifications to refine our model's training process:

1. **Implement Early Stopping**: To prevent overfitting and save time, we implemented an EarlyStopping callback to end training once the model plateaus on the validation set.
2. **Reduce Learning Rate on Plateau**: Adjust the learning rate when a plateau in model performance is detected. This allows for finer tuning of the model weights.
3. **Fine-Tune Dropout Rates**: We added a new dropout layer, as well as lowering the dropout rate of the first one from $0.5 \rightarrow 0.3$.
4. **Model Checkpoint**: Add a model checkpoint to save the model at it's best validation value.

We implemented these changes (visible in the final code output at the bottom of this write-up) and ran it again with 200 epochs.



Training MAE: 0.0241, Validation MAE: 0.0112



Training Loss: 0.0022, Validation Loss: 0.0017

As we can see from the above, our model checkpoint cut us off around 57 epochs, rather than going the full 200. We can further see slight improvements in our MAE and MSE values (0.0112 vs. 0.0177), indicating improvements to the model's ability to predict our data!

Manual Testing

To determine the accuracy of our model in real life, we wanted to construct fake users and see how the model predicts those users' affinity with u_c. Our goal was to confirm that the model's predictions matched our understanding of what was a "good" or "bad" feature in a user.

Manual Inputs and Results

(ID) **Affinity**	Age difference	Buddies	Friends	Common Friends	Common Hobbies	Common Groups
(1) 30	0	0	0	2	3	0
(2) 10	10	0	0	2	3	0
(3) 37	1	1	0	10	5	10
(4) 17	1	1	0	10	5	40
(5) 26	10	1	0	10	5	10
(6) 10	20	1	0	10	5	10
(7) 48	1	1	1	10	5	10
(8) 20	2	0	0	1	1	0
(9) 27	2	0	0	1	4	0
(10) 35	2	0	0	1	10	0

(ID) **Affinity**	Age difference	Buddies	Friends	Common Friends	Common Hobbies	Common Groups
(11) 25	1	0	0	1	2	0
(12) 30	1	0	0	4	2	0
(13) 36	1	0	0	10	2	0
(14) 30	1	0	0	4	2	0
(15) 27	1	0	0	4	2	2
(16) 25	1	0	0	4	2	4
(17) 27	1	0	0	4	2	2
(18) 47	1	0	1	4	2	4

Test 1: Age difference (rows $1-2$)

As we can see from the first two rows, an increase in the age difference results in a lower affinity scoring. This matches the expected outcome, since we believe people closer in age to be higher in affinity for friendship.

Test 2: Buddy variance (rows $3-7$)

These rows test variance in affinity for a buddy given variance in other characteristics. We want no adjustments in affinities - with the hope of near-100 affinities for all, since a buddy should always be included in the groups that u_c is invited to. However, the results we got were not the expected ones: the variance was high based on other features, and the buddy value was not weighted nearly high enough. We attribute this to the dataset only having 1 buddy sample, which is not enough to demonstrate a pattern to our neural network. In the future, we would have to increase the number of buddy samples (or reuse the same one quite a few times) to indicate a clear pattern to the network. We considered this beyond the scope of the project, since we can always hardcode buddy affinities.

Test 3: Hobby overlap (rows $8-10$)

Test 4: Common Friends overlap (rows $11-13$)

Test 5: Groups overlap (rows $14-16$)

Test 6: Friends (rows $17-18$)

Further Feature Ideas

1. Per-hobby overlap

   This would mean a feature for each user being "do we share this hobby?" - and it will be either 0 for neither u_c nor u_i have this hobby, -1 for u_c has this hobby (but u_i doesn't), or 1 for both of u_c and u_i have this hobby.

   This could be useful because users may weigh certain hobbies above others in how important they are to have in friends. The neural network should be able to pick up and account for this variance.

   > *Adapting the network to allow for this would be interesting, as far more personalized weights would be constructed. At the moment, our neural network is training on data that isn't super personalized to the individual's preference of friends. Different users will weigh their hobbies differently, and will also sometimes consider hobbies they don't have to be a positive. For this reason, more separated features around the overlap in hobbies could allow for more precise and personalized affinities.*

2. Location proximity

In the real platform, the goal would be to find friends **in person**. This would make physical proximity an extremely important aspect of affinity construction and therefore should be taken into account. We did not yet include that in our initial dataset, as this project mainly focuses on the basic construction and training of the neural network.

3. Availability/schedule overlap

   Since the goal was to give people opportunities to meet each other, we want u_i and u_c to have overlapping times when they can meet up, so our platform can recommend times to them. Therefore, matching users based on compatibility of schedule would be a useful feature.

4. Improving friends/buddy features list

   There are a lot of ways to improve the current features of buddies and friends that we have - if u_i is friends with u_c's buddy, it should weigh higher than if they are friends with one of u_c's friends. By the same token, if u_c is friends with u_i's buddy, it should weigh higher as well (than if u_c is just friends with one of u_i's friends).

   This type of tiered social proximity is useful in a purely friend-based context as well - if u_i and u_c share a lot of third-degree friends, that should be indicative of having no social overlap at all. Including features for third and fourth-degree friends could be useful.

   Further, once affinities are stored between every pair of users on the platform, we can take into account the average affinity between the given u_i and the common friends they have with u_c.

Code Segments

Below is the code for the construction and training of the neural network.

```
# Normalize y to be between 0 and 1
y_train = y_train / 100.0
y_test = y_test / 100.0

self.model.add(Dense(128, input_shape=(6,), activation='relu', kernel_regularizer=l2
(0.001)))
self.model.add(Dropout(0.3))  # Slightly reduce dropout
self.model.add(Dense(64, activation='relu'))
self.model.add(Dropout(0.3))  # Slightly reduce dropout
# Output layer with sigmoid activation for output between 0 and 1
self.model.add(Dense(1, activation='sigmoid'))

self.model.compile(optimizer=Adam(learning_rate=0.001), loss='mse', metrics=['mae'])

# Callbacks
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights
=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=1e-
5)
model_checkpoint = ModelCheckpoint('best_model.h5', monitor='val_loss', save_best_on
ly=True)
```

```
history = self.model.fit(
    x_train, y_train, epochs=200,  # Reduced the number of epochs
    validation_data=(x_test, y_test),
    batch_size=32,
    verbose=1,
    callbacks=[early_stopping, reduce_lr, model_checkpoint]  # Added callbacks
)
```

Full Github Project