Assessing the Reliability of AlphaFold3 Predictions forProtein-Ligand Affinity Prediction via Sfcnn

Guo Yu (2022533074) * **Linzheng Tang (2022533087)** *
Junlin Chen (2022533009) *

* *ShanghaiTech University (email: yuguo2022@shanghaitech.edu.cn, tanglzh2022@shanghaitech.edu.cn, chenjl2022@shanghaitech.edu.cn)*

Abstract: This project investigates the reliability of using AlphaFold3 (AF3)-predicted structures as an alternative. Sfcnn, a 3D-CNN based protein-ligand affinity prediction model, is reproduced using PyTorch, and its performance is validated on the PDBbind v2019 refined set for training and the CASF-2016 core set for testing. The AF3-derived protein structures of CASF-2016 core set is then evaluated and compared against the groundtruth and Sfcnn scores on the core set to assess the viability of AF3 predictions in PLA tasks.

Keywords: AlphaFold3, protein-ligand affinity, CNN scoring function, CASF-2016

1. INTRODUCTION

1.1 Sfcnn Background

Sfcnn is a 3D convolutional neural network based scoring function model proposed in 2022, which aims to provide accurate and reliable scores for binding affinities of protein-ligand complexes.

1.2 Data Methods

Dataset The Sfcnn network was trained with protein–ligand complexes from the refined set of the PDBbind database version 2019, which contains protein—ligand complexes and their corresponding binding affinities expressed with pKa values, the trained network is later tested on the CASF-2016 core set, which has 285 protein–ligand complexes.

Note that the overlaps between train set and test set (266 protein complexes) are excluded, leaving 4852 train complexes in total.

Augumentation To scale up the training set, each protein-ligand complex is rotated randomly for 9 times using random rotation matrices, those 10 complexes should bear the same PLA (protein-ligand affinity) score, resulting in total 48520 complexes for training

Featurization To capture the features of a protein-ligand complex, Sfcnn uses the method of grid mapping and one-hot encoding. Each complex is mapped to a 3D grid with resolution 20 × 20 × 20, which is later transformed into a 4D tensor. Each cell within the grid is a formed by an encoding list of length 28, consists of 14 protein atom(isotope)[1] and 14 corresponding ligands, mapped with one-hot encoding method. The final training tensor size is therefore (48520, 28, 20, 20, 20).

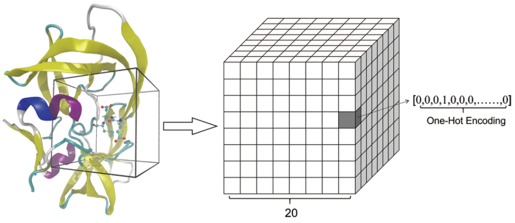

Fig. 1. Featurization of the protein–ligand complexes. PDB ID 1a30 is shown as an example. In the default case, the resolution of 20×20×20 and 28 categories of atomic types were used

1.3 Network

The original paper presents 4 different network structures along with 3 ways of featurization. The network shown in the figure, combing with the featurization method above achieved best performance on validation set.

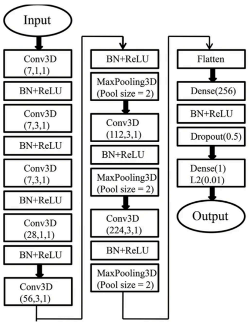

Fig. 2. Final CNN structure for Sfcnn network

[1] Please refer to the original Sfcnn paper for those atom types: https://bmcbioinformatics.biomedcentral.com/articles/10.1186/s12859-022-04762-3#availability-of-data-and-materials

This network features 3D convolution layers with batch normalization and ReLU activation.L2 regularization was applied on the output layer to reduce the probability of overfitting and improve generalization.

2. ENVIRONMENT

The training environment is built based on Gymnasium, which is a python reinforcement learning framework maintained by Farama Foundation. The reinforcement learning agents are applied in the form of a game mod, named "Combat Agent".

The server side(agent) and client side(game) has a communication interval of 0.5 seconds, agents are trained in parallel in the vectorized environment with the following specifications:

2.1 Action Space

Due to the short communication interval, the action space is restricted by the pawn's walking speed. The action space is therefore defined as a unit length nine-square grid, centering the ally pawn, whose relative coordinates is set to (0, 0).

An action is defined as the coordinate within the relative nine-square grid.

2.2 Observation Space

The observation space consists of 6 layers in total, each layer represents certain information of the current game state and is shaped 15x15. Those layers are viewed as 6 channels of the observation space.

Ally position layer A binary map, where the value is 1 if the ally pawn is at the corresponding position, otherwise 0.

Enemy position layer A binary map, where the value is 1 if the enemy pawn is at the corresponding position, otherwise 0.

Cover positions layer A discrete map, where 1 represents the tree position and 2 represents the wall position.

Aiming layer A binary map, where the value is 1 if the ally pawn or the enemy pawn is aiming, otherwise 0. This layer helps the agent determine whether to interrupt the aiming process of both parties.

Status layer A binary map, where the value is 1 if the ally or enemy pawn is paralyzed, otherwise 0.

Danger layer A discrete map, where the value is 0-100, representing the danger level of both parties. The specific calculation of danger value will be elaborated in the reward function section.

Frame Stack The 6 layers present the current game state, however, in actual combating, the agent needs information through the timeline to make a better decision. Therefore, the observation space is stacked per 8 frames, making the observation space 6x15x15x8, those frames are viewed as the depth dimension.

2.3 Episode Termination

The episode is terminated if one of the following conditions is met:

1. A winner is declared.
2. The maximum step length 800 is reached. In which case, episode reward will be preserved with no change.

2.4 Reward Function

After each episode is terminated, the episode reward will be calculated and recorded based on the following reward table:

Table 1. Reward Settings

Reward Type	Enemy	Ally
Original	/	0
Win	-50	$+50$
Danger	$+200$	-200
Invalid Action	/	-0.25
Distance	/	± 0.1
Cover	/	$+0.15$

Notice that only the currently in use rewards are given in the table, other rewards such as ally remain still penalty are designed but deprecated, their coefficients are set to 0.

Win If a winner is declared and it is the ally, the ally will be rewarded 50 points, otherwise it will be penalized 50 points.

Danger The original paralyzing mechanism in the game is calculated through a linear piecewise function regarding the pawn's consciousness, pain and blood loss stage.

In actual calculations, to unify the computing system, the paralyzing danger degree is defined as follows:

$$d = \frac{1}{1 + \exp(5 + c - p - b)} \tag{1}$$

where c is the pawn's consciousness proportion, p is the pawn's pain proportion and b is the pawn's blood loss proportion, they are floats ranging from 0 to 1.

The danger value is mapped to (0, 1) with 2 decimal places by sigmoid function. When the value is greater than 0.75, the pawn will be paralyzed. The corresponding reward is calculated by the difference equation:

$$r_d = c_d \cdot (d_{n+1} - d_n) \tag{2}$$

where r_d is the danger reward , c_d is the danger reward coefficient, d_{n+1} is the danger value of the current step, d_n is the danger value of the previous step.

Notice that the paralyzing danger degree is calculated separately from the game's internal paralyzing mechanism, and when passed to the Danger layer, it is mapped to an integer within (0, 100).

Invalid Action When the ally pawn tries to move to a position that is blocked by walls or outside its action space, the action is deemed invalid.

Invalid coordinates will be clipped to the action space and reward will be penalized c_i points, where c_i is the invalid action coefficient.

Distance The ally pawn tends to perform better when it keeps certain distance from the enemy pawn. In practice, the optimal distance is set to 5 with a slack range of 1 grid. Once the ally is out of the optimal distance range, the reward will be penalized c_d points, otherwise the reward will be added c_d points.

Cover The ally is encouraged to take cover behind walls or trees and hide from the enemy's straight shot. The action space of ally is divided into north-east, north-west, south-east and south-west quadrants, the quadrant closest to the enemy is activated, every cover within such quadrant will be rewarded c_c points, where c_c is the cover reward coefficient.

3. DEEP-Q-LEARNING

3.1 Neural Network

Double Q-learning The DQN agent uses double Q-learning to stabilize the training process. The agent consists of two identical Q-networks: the policy network and the target network.

The policy network is used to estimate the Q-value of the current state, while the target network is used to estimate the Q-value of the next state. This separation helps mitigate the overestimation bias inherent in traditional Q-learning. The Q-value of the current state is calculated using the Bellman equation:

$$Q(s_t, a_t) = r_t + \gamma \cdot \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}), \tag{3}$$

where s_t is the current state, a_t is the action taken, r_t is the reward received, γ is the discount factor, and s_{t+1} is the next state. The target network is periodically updated to match the policy network, ensuring stable and consistent Q-value estimates.

Distributional Q-learning Distributional Q-learning extends traditional Q-learning by modeling the distribution of possible returns rather than just their expected values. Instead of estimating a single Q-value for each state-action pair, the agent predicts a probability distribution over a discrete set of possible returns (atoms).

This approach captures the inherent uncertainty in the environment and provides richer information for decision-making. The distributional Q-value is computed as:

$$Z(s_t, a_t) = \sum_{i=1}^{N} p_i \cdot \delta_{z_i}, \tag{4}$$

where p_i is the probability of the i-th atom z_i, and δ_{z_i} is the Dirac delta function centered at z_i. The expected Q-value can then be derived by taking the weighted sum of the atoms:

$$Q(s_t, a_t) = \sum_{i=1}^{N} p_i \cdot z_i. \tag{5}$$

This method improves the agent's ability to handle stochastic environments and leads to more robust learning.

ResNet Block The first block of the model is based on the ResNet architecture, specifically the R3D-18 variant, which is adapted for 3D input data. The ResNet block consists of a series of convolutional layers with residual connections, allowing the network to learn deeper representations without suffering from vanishing gradients.

The input is passed through a stem convolutional layer, followed by multiple residual blocks. Each residual block includes a combination of 3D convolutions, batch normalization (replaced with group normalization for stability), and ReLU activation functions. The output of the ResNet block is a high-dimensional feature representation of the input state, which is then fed into the subsequent layers of the network.

Noisy Linear Layer The noisy linear layer is a variant of the standard linear layer that introduces noise to the weights and biases during training. This noise encourages exploration by adding uncertainty to the network's predictions, which is particularly useful in reinforcement learning tasks where exploration is critical. The noisy linear layer is defined as:

$$y = \left(W_\mu + W_\sigma \cdot \epsilon_W\right) \cdot x + \left(b_\mu + b_\sigma \cdot \epsilon_b\right), \tag{6}$$

where W_μ and b_μ are the mean weights and biases, W_σ and b_σ are the standard deviations of the noise, and ϵ_W and ϵ_b are random noise terms sampled from a standard normal distribution.

During evaluation, the noise is disabled, and the layer behaves like a standard linear layer. This approach allows the agent to balance exploration and exploitation effectively.

Dueling Network Block The dueling network architecture separates the estimation of state values and action advantages, which are then combined to produce the final Q-value estimates. This separation allows the network to learn which states are valuable without needing to learn the effect of each action in every state.

The dueling network consists of two streams: the value stream and the advantage stream. The value stream estimates the value of the state $V(s)$, while the advantage stream estimates the advantage of each action $A(s, a)$. The two streams are combined using the following formula:

$$Q(s, a) = V(s) + \left(A(s, a) - \frac{1}{|A|} \sum_{a'} A(s, a')\right), \tag{7}$$

where $|A|$ is the number of actions. This ensures that the Q-values are centered around the state value, improving the stability of the learning process.

Both streams consist of multiple noisy linear layers with ReLU activations, followed by a final fully connected layer that outputs the value and advantage estimates. The final Q-value distribution is obtained by applying a softmax function to the combined output.

3.2 Action

Epsilon Greedy The DQN agent uses epsilon greedy strategy. Specifically, the epsilon decay follows the following formula:

$$\varepsilon = \max(\varepsilon_{\text{final}}, \varepsilon_0 \times (1 - \exp(-5 \times d^s))) \tag{8}$$

where d is the decay rate, ε_0 is the initial epsilon, $\varepsilon_{\text{final}}$ is the final epsilon threshold, s is the step number. In practice, ε_0 is set to 1.0, $\varepsilon_{\text{final}}$ is set to 0.01, d is set to 0.9999935.

The agent will take the action with the highest Q value with probability $1 - \varepsilon$ and a random action sampled from the action space with probability ε.

3.3 Update

Prioritized Experience Replay The DQN agent employs prioritized experience replay to improve sample efficiency and learning stability. Instead of uniformly sampling experiences from the replay buffer, the agent prioritizes experiences based on their temporal difference (TD) errors.

Experiences with higher TD errors are more likely to be sampled, as they provide more significant learning signals. The priority of each experience is computed as:

$$p_i = |\delta_i| + \varepsilon_p, \tag{9}$$

where δ_i is the TD error for the i-th experience, and ε_p is a small constant to ensure that all experiences have a non-zero probability of being sampled. The sampling probability for each experience is then given by:

$$P(i) = \frac{p_i^\alpha}{\sum_j} p_j^\alpha, \tag{10}$$

where α controls the degree of prioritization. When $\alpha = 0$, the sampling becomes uniform, and when $\alpha = 1$, the sampling is fully prioritized. To correct for the bias introduced by prioritized sampling, importance sampling weights are applied during the update:

$$w_i = (N \cdot P(i))^{-\beta}, \tag{11}$$

where N is the size of the replay buffer, and β controls the degree of importance sampling correction. The weights are normalized to ensure stability during training.

Policy Network Update The policy network is updated using the sampled experiences and their corresponding importance weights. The loss function for the policy network is based on the KL divergence between the predicted Q-value distribution and the target distribution. The target distribution is computed using the target network and the Bellman equation:

$$T(s_t, a_t) = r_t + \gamma \cdot Z(s_{t+1}, a_{t+1}), \tag{12}$$

where $Z(s_{t+1}, a_{t+1})$ is the distributional Q-value predicted by the target network for the next state-action pair. The loss function is then given by:

$$L = \sum_i w_i \cdot \text{KL}\ (Q(s_t^i, a_t^i), T(s_t^i, a_t^i)), \tag{13}$$

where $Q(s_t^i, a_t^i)$ is the predicted Q-value distribution for the i-th experience, and KL is the Kullback-Leibler divergence. The policy network is updated by minimizing this loss using gradient descent.

Target Network Update The target network is periodically updated to match the policy network, ensuring stable and consistent Q-value estimates. This update is performed by copying the weights of the policy network to the target network at regular intervals.

The update frequency is controlled by a hyperparameter, typically set to a few thousand steps. This periodic update helps mitigate the overestimation bias inherent in traditional Q-learning and stabilizes the training process.

Priority Update After each policy network update, the priorities of the sampled experiences are updated based on the new TD errors. The updated priorities are computed as:

$$p_i^{\text{new}} = |\delta_i^{\text{new}}| + \varepsilon_p, \tag{14}$$

where δ_i^{new} is the new TD error for the i-th experience. These updated priorities are then stored in the replay buffer, ensuring that the sampling probabilities reflect the most recent learning signals.

This dynamic prioritization allows the agent to focus on the most informative experiences, leading to more efficient learning.

3.4 Training Results

The DQN agent is trained with 1,000,000 steps, the recorded inner parameter curve is shown in the following figure:

The detailed training reward and actual tactical development of the agent will be shown in the result comparison section.

4. PROXIMAL POLICY OPTIMIZATION

The general workflow of PPO is the same as DQN, the difference is that the update of PPO is performed with a fixed interval with temporal memory, which makes it generally faster in convergence than DQN.

4.1 Neural Network

Convolution block The structure of PPO convolution block is the same as the DQN convolution block mentioned in the previous section.

Actor block The actor block is responsible for the action sampling of the agent, which consists of two Linear ReLU layers and a full connected layer.

The output of the actor block is then scaled and used to calculate the action mean value and action standard deviation.

Critic block The critic block is responsible for the state value estimation, which consists of two Linear ReLU layers and a full connected layer.

The output state value will be used for advantage calculation in the update step.

4.2 Action

Sampling After forward propagation, the next action will be sampled from a certain distribution with action log probability.

In such process, the nine-square action space is mapped to a 9x1 action list, ranging from 0 to 8, where each number represents one of the nine positions in the action space.

For both x and y coordinates, the next x and y will be sampled and rounded separately from the following Categorical distribution, which is widely used in describing the discrete action space:

$$\text{Mapped}(x, y) \sim \text{Categorical}(l) \tag{15}$$

where l is the log probability of the action, x and y are coordinates of the action.

Evaluation In this part, the state tensor will go through forward propagation again, the action will be resampled from the same distribution. The log probability and distribution entropy of the action will be used for the calculation of loss in the update step.

4.3 Update

The update step is performed once every 500 steps in practice, key concept is using the temporal difference and clipped advantage to calculate total loss and use back propagation for parameter update.

GAE (Generalized Advantage Estimation) This method is used to calculate the advantage of the current state, the calculation flow is as follows:

Temporal difference calculation:

$$\delta_t = r_t + \gamma V(s_{t+1}) \cdot (1 - \text{done}_t) - V(s_t) \tag{16}$$

where r_t is the reward of the current state, γ is the discount factor, which is set to 0.97 in practice, $V(s_{t+1})$ is the state value of the next state, done_t is the termination flag of the current state.

Current state advantage calculation:

$$A_t = \delta_t + \gamma\lambda(1 - \text{done}_t)A_{t+1} \tag{17}$$

where λ is the GAE parameter, which is set to 0.95 in practice, A_{t+1} is the advantage of the next state.

Current state return calculation:

$$R_t = A_t + V(s_t) \tag{18}$$

where R_t is the return of the current state.

Normalized advantage calculation:

$$\hat{A_t} = A_t - \frac{\mu_A}{\sigma_A + \varepsilon} \tag{19}$$

where μ_A is the mean value of the advantage, σ_A is the standard deviation of the advantage, ε is a small number to prevent division by zero,

which is set to 10^{-8} in practice.

Actor Loss The actor loss is associated with the estimated advantage and entropy, the calculation follows equation:

Surrogate loss 1:

$$s_1 = \exp(l_t - l_{t-1}) \cdot \hat{A_t} \tag{20}$$

where l_t is the log probability of the current state, l_{t-1} is the log probability of the previous state, $\hat{A_t}$ is the normalized advantage of the current state.

Surrogate loss 2:

$$s_2 = \text{clamp}(\exp(l_t - l_{t-1}), 1 - c, 1 + c) \cdot \hat{A_t} \tag{21}$$

where c is the clip parameter, which is set to 0.1 in practice, clamp is a function that clips the value to the range of (1 - c, 1 + c).

Notice that the two surrogate losses are calculated using absolute values instead of exponential functions to prevent value explosion.

Entropy bonus:

$$e_b = e_c \times \text{entropy}(s_t) \tag{22}$$

where e_b is the entropy bonus, e_c is the entropy coefficient, $\text{entropy}(s_t)$ is the entropy of the current state, in practice, e_c is set to 0.05.

Total Actor loss:

$$L_a = -\min(s_1, s_2) - e_b \tag{23}$$

where L_a is the actor loss.

Critic Loss The critic loss is associated with the state value, the calculation follows equation:

$$L_c = (R_t - V(s_t))^2 \tag{24}$$

where L_c is the critic loss, R_t is the return of the current state, $V(s_t)$ is the state value of the current state.

Total Loss The total loss is the sum of the actor loss and the critic loss, the calculation follows equation:

$$L = L_a + L_c \times c_c \tag{25}$$

where L is the total loss, c_c is the critic coefficient.

4.4 Training Results

The PPO agent is trained with 1,000,000 steps, the recorded loss curve is shown in the following figure:

The detailed training reward and actual tactical development of the agent will be shown in the result comparison section.

5. POLICY GRADIENT METHOD

5.1 Neural Network

Convolution Block The structure of the convolution block is the same as the DQN convolution block mentioned in the previous section.

Actor block Similar to the PPO actor block mentioned in the previous section, the actor block is responsible for the action sampling of the agent, which consists of two Linear ReLU layers and a full connected layer.

5.2 Action

Sampling Similar to PPO, the exact action coordinates will be sampled from two separate Categorical distributions and mapped to the action space.

Evaluation The same as PPO except that the action would not be forwarded the second time, only log probability and entropy of the action will be recorded as the sum of the two coordinates' log probability and entropy.

Notice that the detailed action sampling and evaluation process is explained in the PPO section and will not be repeated here.

5.3 Update

The update step is performed once every 500 steps in practice, key concept is using return values calculated by reward function to update the policy network.

Return Value The return value is calculated and normalized by the following equation:

$$R_t = \sum(r_t * \gamma^{t-1}) \tag{26}$$

$$\hat{R_t} = R_t - \frac{\mu_R}{\sigma_R + \varepsilon} \tag{27}$$

where R_t is the return value of the current state, r_t is the reward of the current state, γ is the discount factor, which is set to 0.975 in practice.

μ_R is the mean value of the return value, σ_R is the standard deviation of the return value, ε is a small number to prevent division by zero, which is set to 10^{-8} in practice.

Policy Loss The policy loss is associated with the return value, the calculation follows equation:

$$L_p = \hat{R_t} \times \log p(a_t) \tag{28}$$

where L_p is the policy loss, $\hat{R_t}$ is the normalized return value of the current state, $p(a_t)$ is the probability of the action.

Total Loss The total loss is the sum of the policy loss and the entropy loss, the calculation follows equation:

$$L = L_p - e_c \times \text{entropy}(a_t) \tag{29}$$

where L is the total loss, e_c is the entropy coefficient.

Notice that in practice, the entropy coefficient is set to 0.05.

5.4 Training Results

The PGM agent is trained with 120,000 steps since it converged with extremely high speed. The recorded loss curve is shown in the following figure:

The detailed training reward and actual tactical development of the agent will be shown in the result comparison section.

6. DISCUSSION

Basically, only Deep-Q-learning agent performs intellectually in the training process, during the training process, the agent has noticed how the shooting system works and has developed the behaviour to actively attack the enemy pawn. Althgouh the agent then converges to stay still at a specific position, but this mis-behaviour is due to the reward setting and can be fixed by adjusting the reward function.

As for PPO and Policy Gradient Method, they failed to form strategies, which is probabiliy due to the unique feature of the game. In the game, only the moving is deterministic, any other non-deterministic states will drastically increase the actual state space the agent percepts, in others words, the hidden states space might actually much larger than we used to expect. This causes the agents needs tons of exploration to find the optimal strategy. Our DQN has a epsilon decay mechanism to directly control the exploration behaviour, while PPO and PGM doesn't have such mechanism, which might be the reason why they failed to form strategies.

Also, since many state transitions in the game are non-deterministic, the estimation of the return value is a huge challenge, which might significantly affect the training process of PPO and PGM, including DQN. Thus, many hyperparameters could not be tuned in a way that under other environments we could have done. For example, the discount factor is set to 0.97 in PPO and PGM, which is reasonable under many circumstances, but might be relatively high in this game.

7. ATTRIBUTION

The original code of the paper is open-sourced on github, the link and contributions are as follows:

Agents

Contribution History:

Paper https://github.com/zivmax/rimworld-combat-agent-paper/graphs/contributors

Contribution History:

RimWorld Mod https://github.com/zivmax/rimworld-combat-agent-client

Contribution History: