Modelling NPC perception using supervised learning

Martin Estgren
<martin.estgren.2093@student.uu.se>
Uppsala University, Sweden

October 23, 2021

1 Introduction

Many games today employ visual perception models for non-playable characters (NPCs) as a way to improve player immersion, where the NPC gives of the illusion of being able to visually perceive the player in the environment. This is usually done through techniques such as geometrically defined vision zones and line of sight estimations using ray-tracing (*M. Walsh 2014* [1]).

Games like *Splinter Cell: Blacklist* (*M. Walsh 2014* [1]) employs sophisticated perception models in order to give of the illusion of NPC cognition and awareness of the player, and it's actions.

In this project, I examine the possibility of using a supervised learning model to approximate these vision models by creating a dataset of situations where a given NPC should be able to perceive the player, and situations where it should be unaware. This dataset will then be used to train a *feed forward neural network*, which the NPC will query during play-time to get an estimate of how well it's supposed to be able to perceive the player.

2 Implementation

The game is implemented in the Godot engine with scripting and game-logic written C#. The perception data is gathered from the render and physics API of the Godot engine, while the perception estimation model is implemented in *Pytorch*. All source code is included with this report.

2.1 Perception model

For the sake of simplicity, the NPC perception will be modelled as three variables:

2.1.1 Distance from NPC to the player

The distance is estimated by taking the *euclidean distance* between the center points of the NPC and the center point of the player, and clamping to a maximum length of *200* units. This distance is then divided by 200 to get a normalized distance value in [0,1]. The full expression can be seen in Equation: 1.

$$\max\left(\min\left(\frac{\left|\vec{x}_{\text{player}} - \vec{x}_{\text{NPC}}\right|}{200}, 1\right), 0\right) \tag{1}$$

, where $\vec{x}_{\text{player}}$ is the player's position, and $\vec{x}_{\text{NPC}}$ is the position of the NPC.

2.1.2 Relative bearing between the NPC and the player

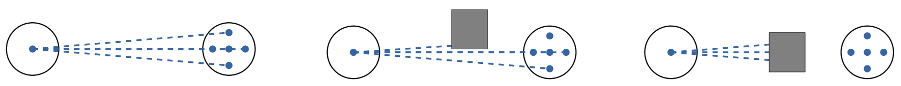
The relative bearing is estimated by taking the previous distance vector's angle in relation to the heading unit vector the NPC (the Up vector rotated to the same angle as the heading of the NPC). The full expression can be seen in Equation: 2.

$$\arccos\left(\left\|\vec{x}_{\text{player}} - \vec{x}_{\text{NPC}}\right\| \cdot \vec{r}_{\text{NPC}}\right) \tag{2}$$

, where $\vec{r}_{\text{NPC}}$ is the normalized heading of the NPC.

2.1.3 Ray-traced line of sight estimation

Line-of-sight is the most complex vision component, but also very important. A NPC that can detect an occluded player will feel very unrealistic and break the player's immersion. It's calculated by ray-casting from the NPC's position to a set of pre-defined points on the player model. Figure 1 shows how the pre-defined points are placed on the player. The final line-of-sight estimation is the ratio of rays that hit the player given the number of points that we tried to raytrace to.



(a) Fully visible (= 1) (b) Partial occlusion($\sim$ 0.5) (c) Fully occluded (= 0)

Figure 1: Different levels of player occlusion. The player is represented the rightmost circle and the NPC is the leftmost one. The blue points on the player represents pre-defined points that will help in determining the player's occlusion relative to the NPC.

2.2 Approximate the perception model

With these perception factors combined, we can model many perception models. For the purpose of this project, I model a rudimentary detection model, where these measures are acts as input to a *feed forward neural network*, which outputs a value $p \in [0,]$. This value p indicates how fast the NPC should detect the player. 0 means that the NPC is unaware of the player's position, while 1 indicates that ht the NPC is fully aware of the player.

2.3 The neural network model

The neural network model is implemented in *Pytorch* and communicates with the Godot engine through an HTTP socket. The model is fairly simple, given we only have 3 input features. The model contains of 3 layers with *ReLU* activation functions, and the output layer outputs a single value using a *sigmoid* activation function. The output value corresponds to an approximation of p. The model can be viewed in Listing: 1.

Listing 1 Pytorch model

```python
import torch
import torch.nn as nn

in_shape = 3
model = nn.Sequential(
    nn.Linear(in_shape,16),
    nn.ReLU(),
    nn.Linear(16,32),
    nn.ReLU(),
    nn.Linear(32,64),
    nn.ReLU(),
    nn.Linear(64,1),
    nn.Sigmoid()
)
```

2.4 Final NPC awareness model

To provide a more interesting and temporally consistent model, I do not model the NPC's perception directly from p_t instead, I create a new value P_t which uses the awareness value p_t as a velocity together with a decay value $d = 1$ on the form represented in Equation: 3 and 4.

$$ds_t = \begin{cases} 5p_t & \text{if } p_t > 0.2 \\ -d & \text{else} \end{cases} \tag{3}$$

$$P_{t+1} = \max\left(\min\left(P_t + ds_t, 1\right), 0\right) \tag{4}$$

This way, we get more consistency over consecutive frames. This also helps smooth out the sampling, as we only query for a new perception measure p every 20ms.

3 Results

To show the generalizability of the model, the model was evaluated using two different datasets, one where the player should be detected *while not occluded and in-front of the NPC* (regular detection model), and one dataset where the player should only be perceived *while occluded and not in-front of the NPC* (Inverse detection model).

These models are trained on their respective datasets, and compared using both in-game testing (the attached video), and diagnostics graphs, presented in this section.

3.1 Regular detection model

Looking at figure 2 and 3, we can observe in both the *accuracy* score of the trained model, and the *loss* score over training iterations, that the model is learning an approximation of our desired reception model.

Performing a grid-search over *line of sight* and *relative bearing*, We can also observe in Figure 4 how it didn't quite predict 0 for all cases of *line of sight* being equal to 0 but have a lower value for those cases compared to higher *line of sight*.

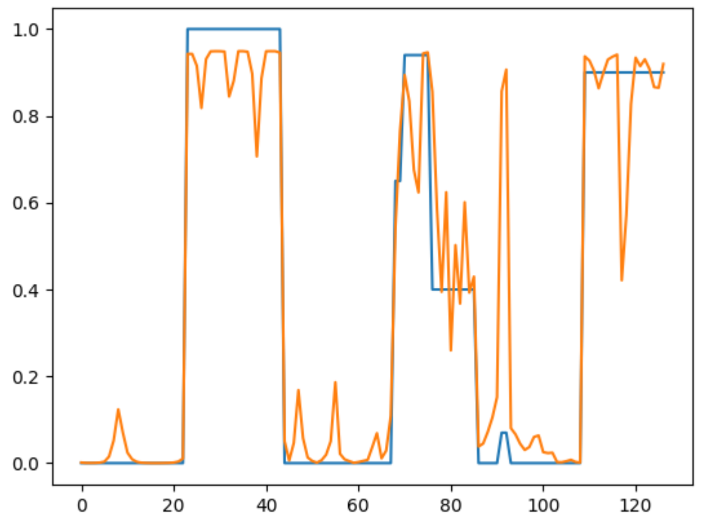


Figure 2: Predicted result (blue) over all cases in the training dataset compared to the expected result (orange).

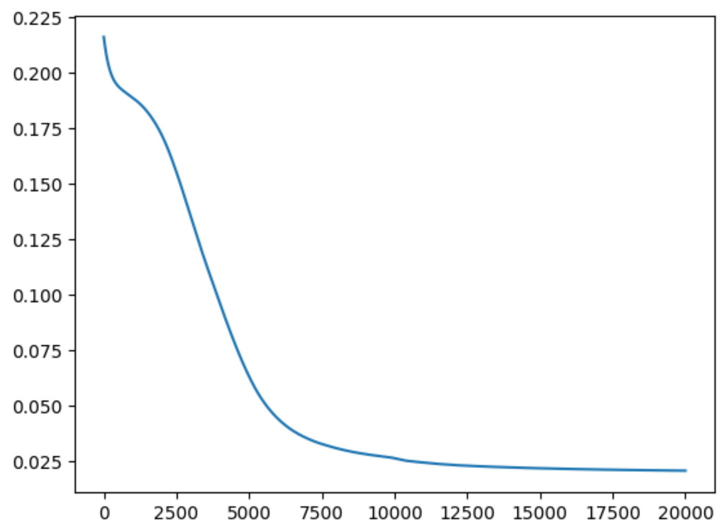


Figure 3: Training loss over training iterations. Loss is calculated using *mean square error*.

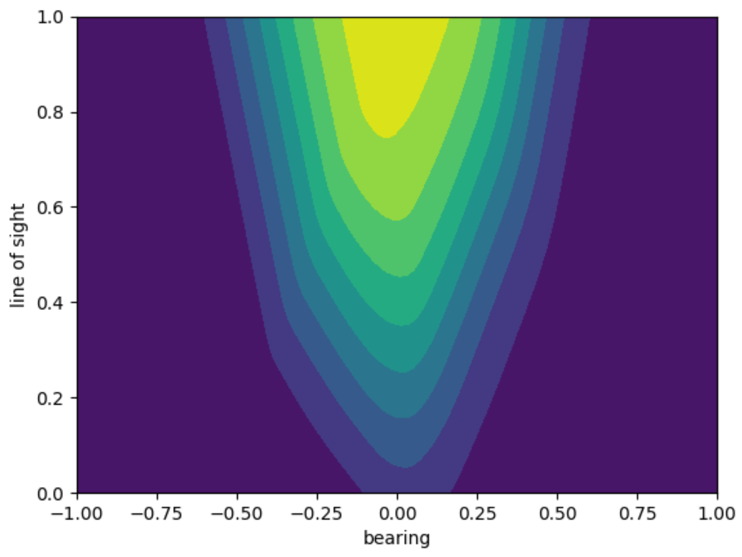


Figure 4: Predicted player awareness over *line of sight ratio* and *relative bearing* grid. Blue indicates low values while yellow indicates values close to 1

3.2 Inverse detection model

As we could see for the regular detection dataset, the model is able to approximate our desired perception model. Figure 5 and 6

Figure 7 shows that the model has generalized the idea that the player only should be perceived when not occluded and in front of the NPC.

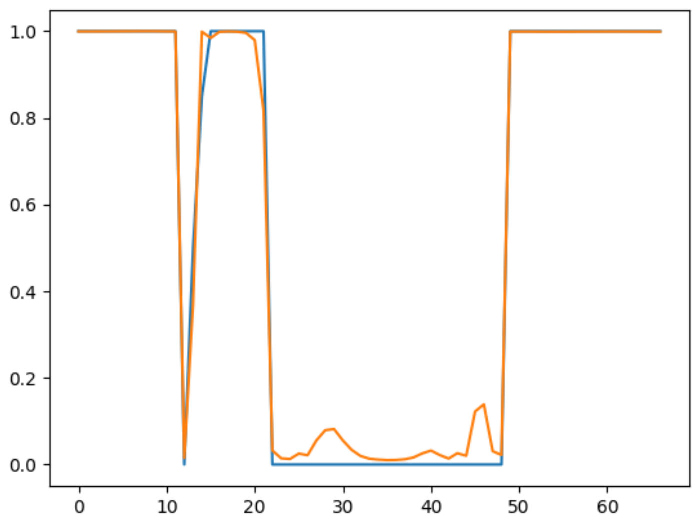


Figure 5: Predicted result (blue) over all cases in the training dataset compared to the expected result (orange).

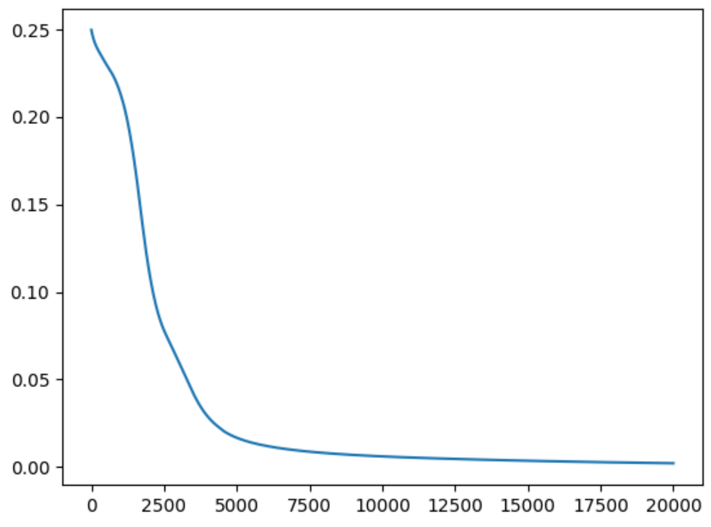


Figure 6: Training loss over training iterations. Loss is calculated using *mean square error*.

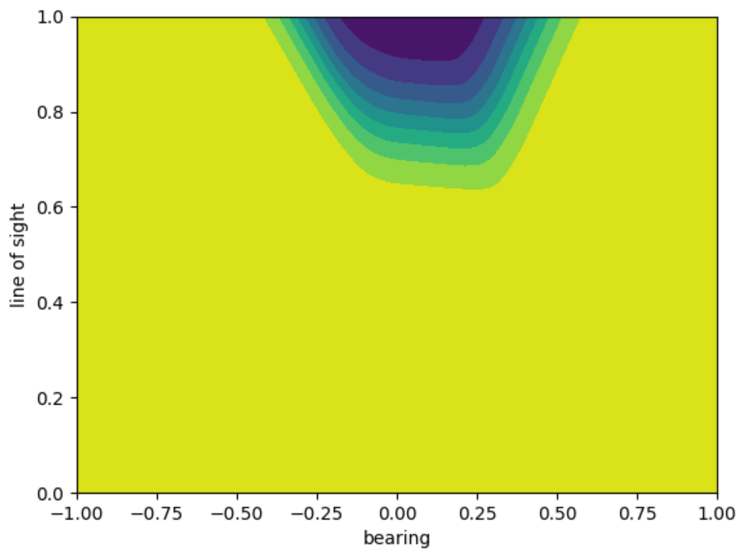


Figure 7: Predicted player awareness over *line of sight ratio* and *relative bearing* grid. Blue indicates low values while yellow indicates values close to 1

4 Conclusions

In conclusions, we can see that my approximation model is able to both approximate wildly different perception models compared to what is commonly used in games, and begin able to provide a reliable and more traditional perception model.

In total, the project resulted in:

- Around 300 lines of Python code with a bespoke neural network model written in Pytorch served over HTTP.
- A Godot project using the built-in scene editor, renderer, physics system, pathfinder, and ray-tracer.
- Around 700 lines of C# code for all game logic and marshalling of data between the game and the Pytorch model.
- More time wasted on debugging the interface between Godot and the Flask server than I care to admit.

References

[1] M. Walsh. Modeling ai perception and awareness in splinter cell: Blacklist. In *Proceedings of the Game Developers Conference (GDC)*, 2014.