Declan Kelly - G00378925

A. Threads

There are five enemies in the game; each of them inherits the `GameCharacter` class, which implements the `Runnable` interface. In the `Runner` class, the enemies are spawned in and added to the `ExecutorService` pool.

This executes the `run` method in the `GameCharacter`; this runs a loop until the enemy is defeated. Enemies are spawned in random locations across the graph, and there can be multiple instances of an enemy. They want to find where the player is and will randomly traverse the graph until they find the player and loiter at that location. This all happens on separate threads.

B. Semantic Network and Grammar Set

The semantic network is implemented using a network of nodes, with each `Location` being a vertex and each direction being an edge. When the game starts, this graph is generated to a certain depth; if you continue to move in the same direction, you will find an exit, but you require a key to go through, this key can be retrieved by defeating an enemy. The graph is created in the `Location.setupLocationGraph`, and uses recursion to create the branches.
You begin in the middle of the graph, you can travel **west** using the `WEST`, and then back to your original location using the `EAST` command.

There are five different locations in the game, which are the following: **Cave**, **Dessert**, **Volcano**, **Castle** and **Laboratory**. Each location has a random number of items, you can view these using the `LOOK` command.

Retrieve an item or weapon using the `GET` command; for example, if there is a `GLOCK` at your location, use `GET GLOCK`. If you picked up an edible item, such as an apple, use the `EAT` command (`EAT APPLE`).

When all the enemies that use neural networks have been trained, you will be presented with a prompt; this is where you will enter text commands. Notice that you can see the enemies currently at your location, listed before the prompt `>`.

To fight an **Orc** at your location with a `GLOCK`, for example, use `FIGHT ORC WITH GLOCK`, the **Dragon** likes to drink alcohol, if the dragon is at your location, use the command: `TELL DRAGON POUR BRANDY` (all beverages are listed in the table below). Get the Dragon plastered and they will fight badly. Upon defeating an enemy, keep travelling in the same direction until you find the exit.

All commands are case-insensitive, parameters are delimitted using whitespace. The commands in `blue` below are used for moving in different directions, these commands don't expect any parameters.

Grammar parsing is implemented in the `Menu.java` file; it polls for an input from standard in, then lexes it using a regular expression to split by whitespace, and then parses it using switch statements.

Verb / Command	Parameter(s)	Description
`LOOK`	-	Will give you a description of the current location, listing all the enemies, items and weapons currently there.
`GET`	`ITEM_NAME`	Retrieve an item or weapon by name that is at the current location.
`FIGHT`	`ENEMY_NAME WITH WEAPON_NAME`	Fight an enemy that is at the current location, using a weapon that is in your inventory.
`EAT`	`ITEM_NAME`	Eat an item that is currently in your inventory, Example: `EAT APPLE`.
`TELL`	`DRAGON POUR BEVERAGE_NAME`	Request the dragon to pour a beverage, available beverages include: `ALE`, `BRANDY` and `WHISKEY`.
`INVENTORY`	-	List the items and weapons inside the players inventory.
`TRAIN`	`ENEMY_NAME`	Train the neural networks, use `TRAIN` to train all NNs, and `TRAIN ENEMY_NAME` to train a specific NN.
`VALIDATE`	`ENEMY_NAME`	Test the validation data on the neural networks, use `VALIDATE` to test all NNs, and `VALIDATE ENEMY_NAME` to validate a specific NN.
`NORTH`	-	Move the player to the location `NORTH` of the current location.
`SOUTH`	-	Move the player to the location `SOUTH` of the current location.
`EAST`	-	Move the player to the location `EAST` of the current location.
`WEST`	-	Move the player to the location `WEST` of the current location.

C. Fuzzy Logic

I have implemented 2 characters: `Dragon` and the `Orc` using Fuzzy Logic. Both of the characters statically encapsulate the `fis` object. Allowing multiple instances of the characters to share the same inference system.

The code for each of the characters can be found in the `Dragon.java` and `Orc.java`.
Like the characters that use the neural network, when you fight the character with a weapon, it calls the `fight` method of the enemy. Some parameters are passed to the `fis` object, and the output being the `attackResponse` is then deducted from the player's health.

1. Dragon

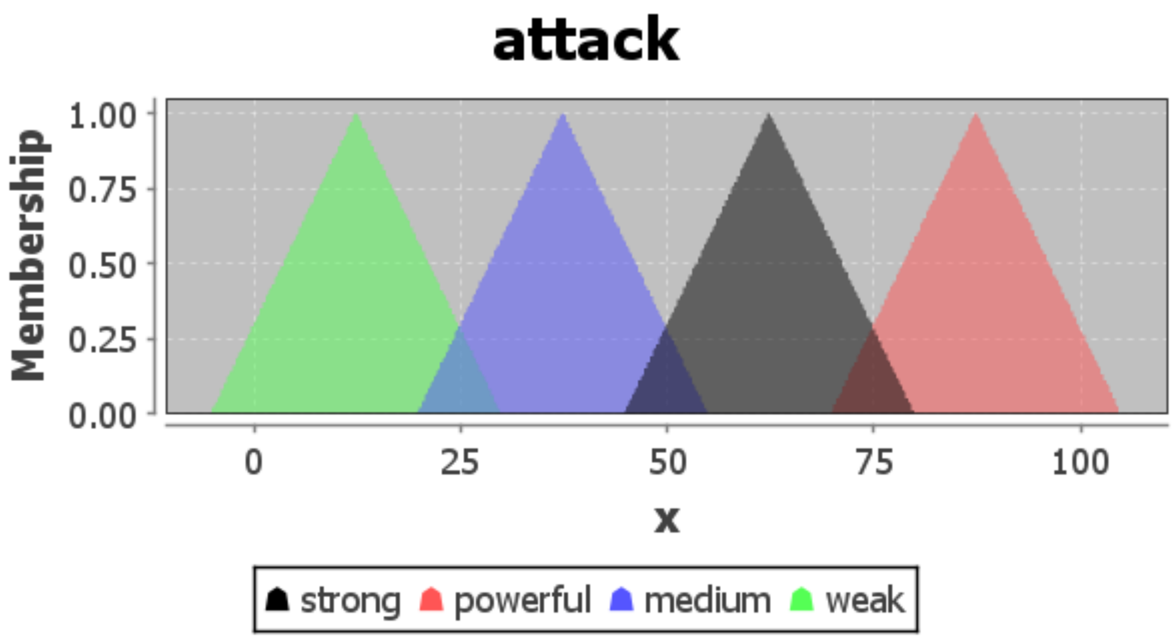
Type: **Mamdani**
FCL: `./resources/fuzzy/dragon.fcl`

For the **Dragon**, I'm using **Mamadani** inference, as seen in the FCL file, the defuzzification is using non-linear functions, being the triangular membership functions. It will be taking in three inputs and emmitting one output.

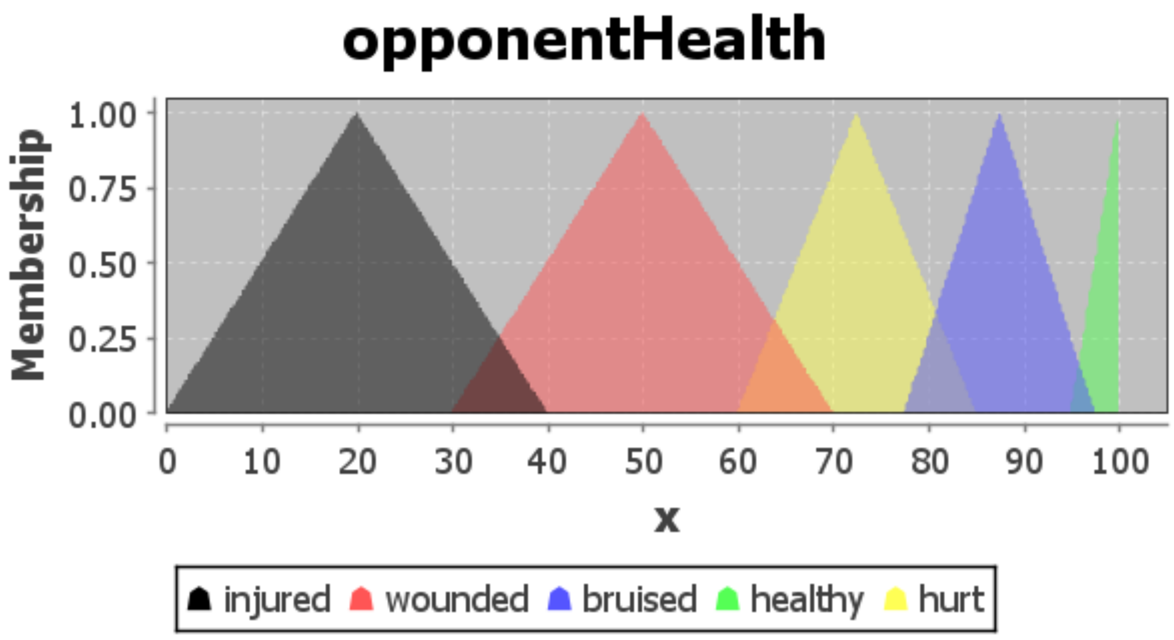
The inputs are the following: `attack`, `opponentHealth` and `bloodAlcoholLevel`.
And the output being the attack response of the Dragon (`attackResponse`).

I am using the **Center of Gravity** (COG) defuzzifier, because its good at modelling nuanced relationships, between the membership functions.

Fuzzy sets and membership functions



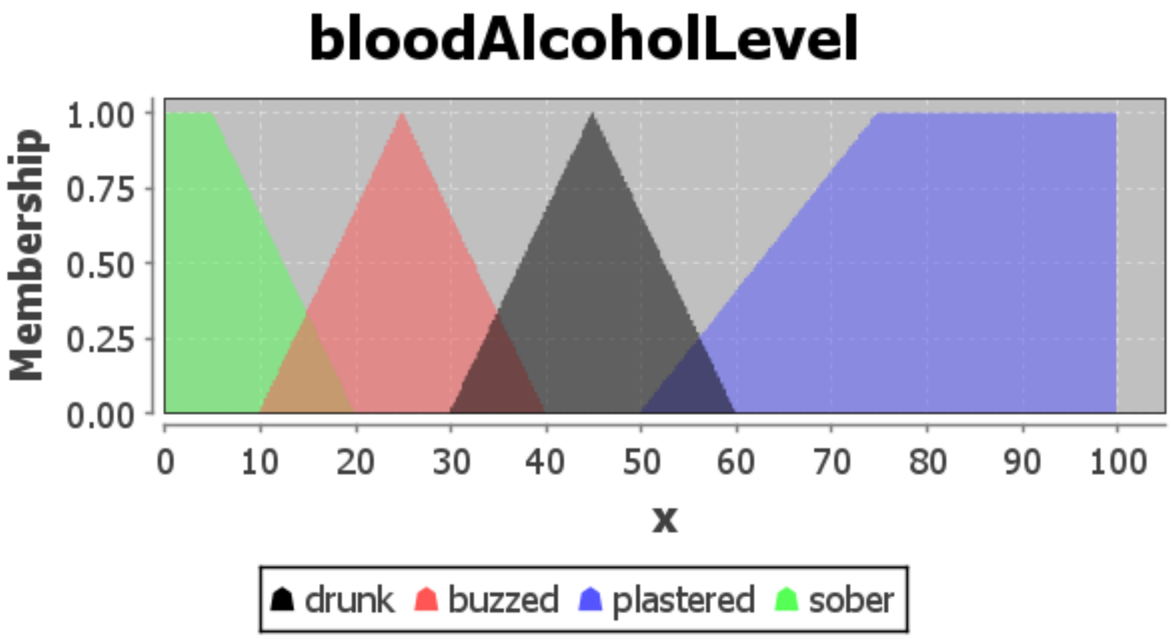
The `attack` comes in four different tiers (**weak**, **medium**, **strong**, **powerful**), each tier of size `35` with an overlap of `5` allowing a gradual fade into the next tier (this is done in all subsequent non-linear fuzzy sets). The universe of discourse is `-5` to `105`.



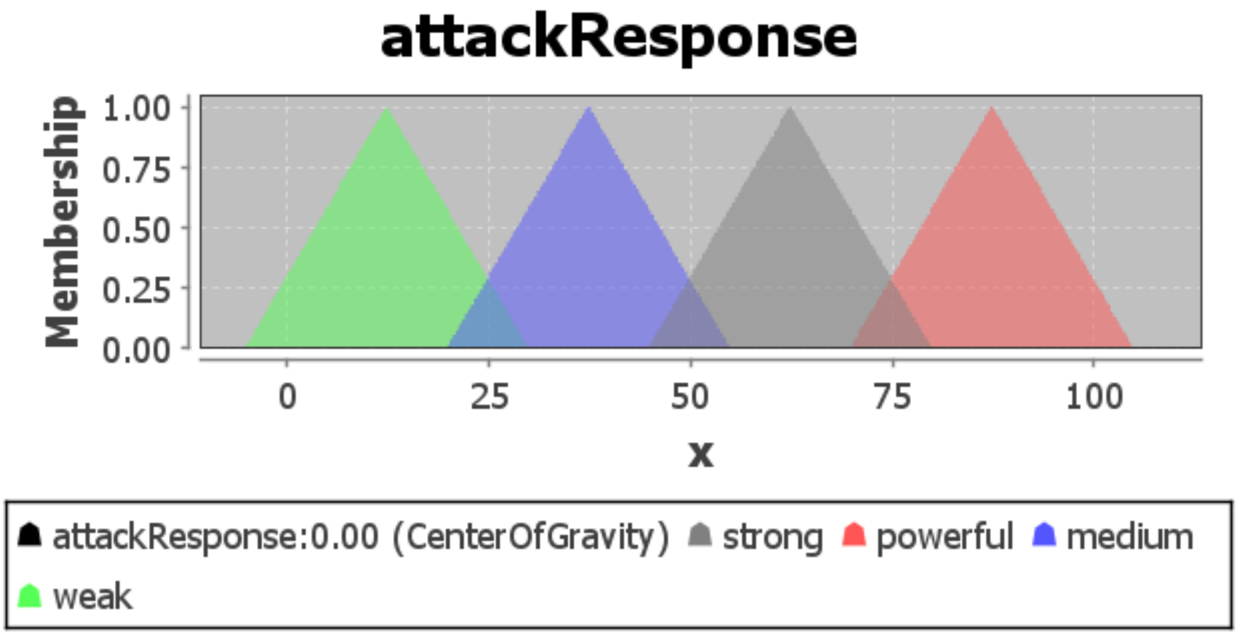
This is the current health of the player, with `100` being top health and being close to `0` is near death.

Dragon's are kind creatures, and do not want unnecessary pain or suffering.
So the players health will be taken into account whilst determining the response attack.

I made the **healthy** membership small, so the player's health will immediatly be put into a lower level, following any altercation.



The `bloodAlcoholLevel` is a measurement of drunkness of the **Dragon**, each of the membership triangles overlap to model the transition between drunken states.
Notice the shapes at the beginning and end aren't triangular, this is because its possible to be `100% sober` or `plastered`.



This fuzzy set is no different to the input `attack`, this is the tier of attack the **Dragon** responds with.

Fuzzy rules

(* If the Dragon is plastered, the attack response will be weak no matter what. *)
(* Or if the player's health is injured, use a weak attack as well. *)
```
RULE 1 : IF bloodAlcoholLevel IS plastered OR opponentHealth IS injured
THEN attackResponse IS weak;
```

(* Being buzzed will make the Dragon significantly stronger. *)
(* Make sure the player isn't injured or bruised or hurt. *)
(* For strong and powerful attacks from the player. *)
```
RULE 2 : IF bloodAlcoholLevel IS buzzed AND attack IS strong OR attack
IS powerful
AND opponentHealth IS NOT injured AND opponentHealth IS NOT bruised AND
opponentHealth IS NOT hurt
THEN attackResponse IS powerful;
```

(* Being drunk will reduce the Dragon's performance slightly, only allowing medium attacks. *)
```
RULE 3 : IF bloodAlcoholLevel IS drunk AND attack IS strong OR attack
IS powerful
AND opponentHealth IS NOT injured THEN attackResponse IS medium;
```

(* If the Dragon is sober and the attack is weak or the health is injured, use a weak attack. *)
```
RULE 4 : IF bloodAlcoholLevel IS sober AND attack IS weak OR
opponentHealth IS injured
THEN attackResponse IS weak;
```

(* If the Dragon is sober and the attack is powerful, *)
(* then the response attack will be strong. *)
(* Dragon's need to be buzzed to unleash their full potential. *)
```
RULE 5 : IF bloodAlcoholLevel IS sober AND attack IS powerful
THEN attackResponse IS strong;
```

(* If the Dragon is sober and the attack isn't powerful, then Dragon will response with medium attack *)
```
RULE 6 : IF bloodAlcoholLevel IS sober AND attack IS NOT powerful THEN
attackResponse IS medium;
```

2. Orc

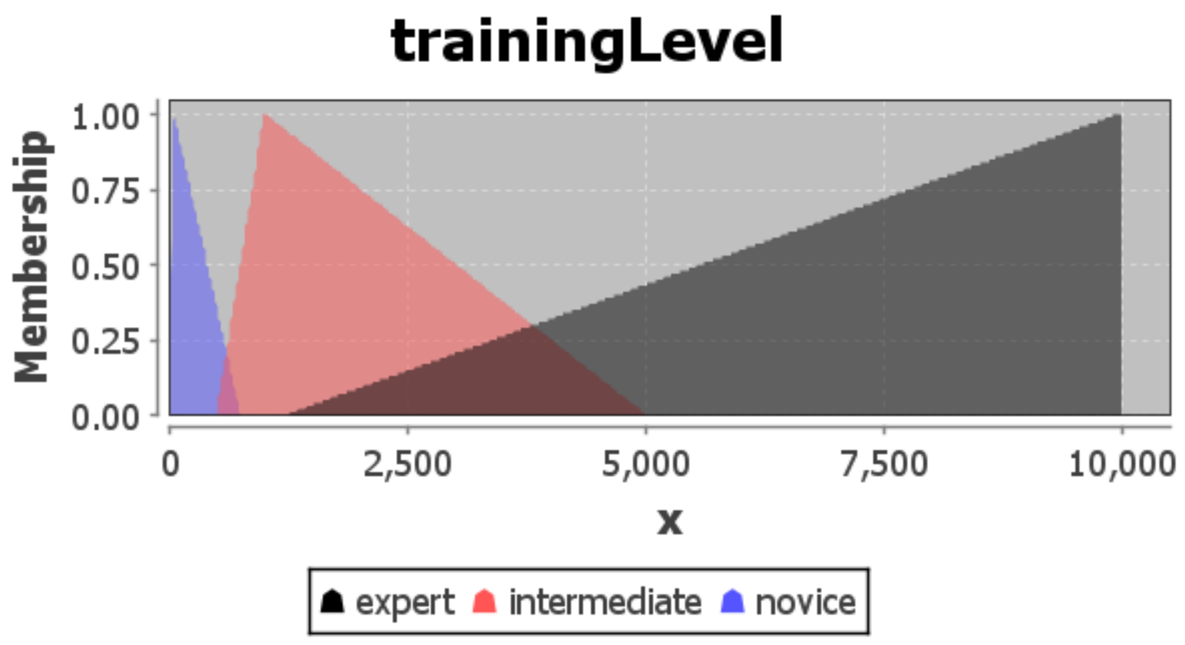
Type: **Sugeno**
FCL: `./resources/fuzzy/orc.fcl`

For the **Orc**, I'm using **Sugeno** interference, as seen in the membership functions and the diagrams below, they are all linear.

The inputs are the following: `trainingLevel` and `attack`.
And the output being the attack response of the Orc (`attackResponse`).

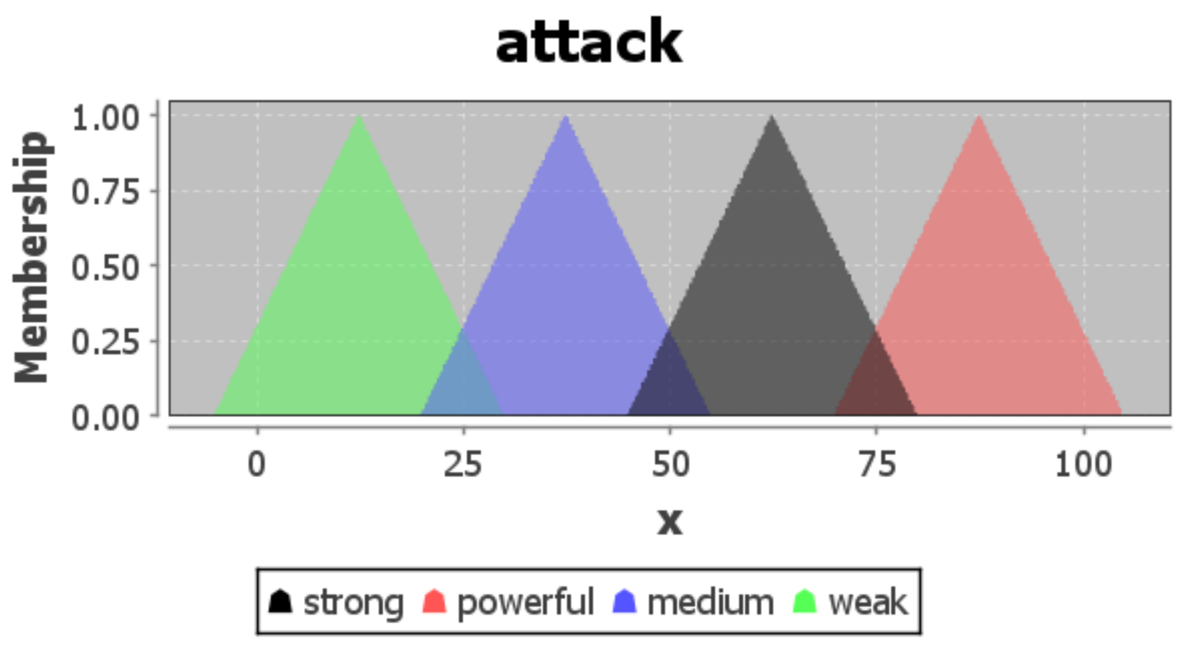
I am using the **Center of Gravity Singleton** (COGS) defuzzifier, because it works best with linear membership functions, defined by a single point (**Sugeno**).

Fuzzy sets and membership functions

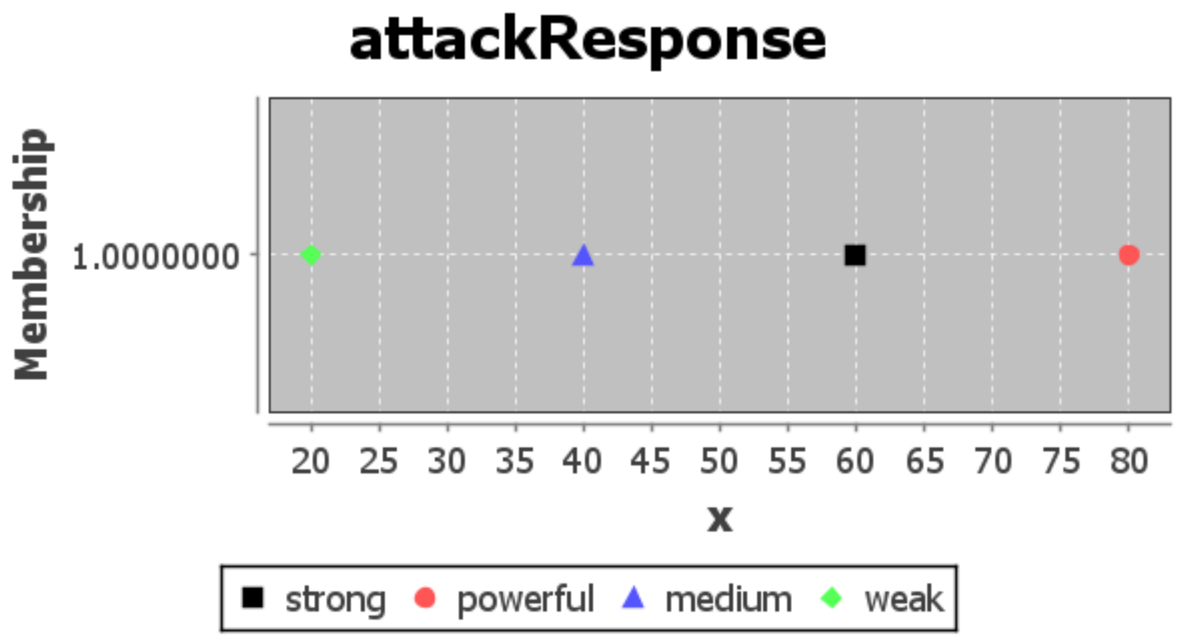


This fuzzy set is used to determine if the **Orc** is a novice (`50` hours), intermediate or an expert at attacking. With some practice, as in most crafts, an Orc can learn enough to be dangerous with some practice (`1,000` hours) (**intermediate**), but to be a true **expert**, you need `10,000` hours.

The triangle are overlapping to show the transition between intermediate and expert. I've stretched the expert triangle out to show the long and drawn-out process of becoming an expert. When **Orc**'s spawn in they will pick a value between `0` and `10,000`, this is also the Universe of Discourse.



This fuzzy set is implemented exactly the same way as the **Dragon**'s `attack` fuzzy set.



This fuzzy set is similar to the `attack` fuzzy set above; the difference is that this is using singleton functions, and the location of the singletons is where the tips of the triangles were in the `attack` set. Each of the membership functions is spaced out by 20.

Fuzzy rules

(* If the Orc is a novice, then all their attacks will be weak, *)
(* no matter the attack of the player. *)
```
RULE 1 : IF trainingLevel IS novice THEN attackResponse IS weak;
```

(* If the player's attack is weak or medium and the Orc is an expert, *)
(* Use a medium response attack. *)
```
RULE 2 : IF trainingLevel IS expert AND attack IS weak
THEN attackResponse IS medium;
```

(* If the Orc is of intermediate level, send back a medium level of attack. *)
```
RULE 3 : IF trainingLevel IS intermediate THEN attackResponse IS
medium;
```

(* If the Orc is an expert and the players attack powerful, respond with powerful. *)
```
RULE 4: IF trainingLevel IS expert AND attack IS powerful
THEN attackResponse IS powerful;
```

(* If the Orc is an expert and the attack isn't weak or powerful, then return medium attack *)
```
RULE 5 : IF trainingLevel IS expert AND attack IS NOT weak AND attack
IS NOT powerful THEN
attackResponse IS medium;
```

D. Neural Network

I have implemented 3 characters: `Goblin`, `Imp` and the `Troll`. Each one is implemented in its own class. The neural network code is implemented statically, allowing multiple instances of each character without having to retrain each instance.

The purpose of the neural networks is to determine the attack values of the characters, using the weapon attributes has input. Using the command: `FIGHT GOBLIN WITH WEAPON_NAME`, you can fight a character. You must make sure you have a weapon first, check the current location weapons using `LOOK`. Find a weapon that is listed as present, use the `GET WEAPON_NAME` to retrieve it. If an enemy character isn't at your current location, move to other locations using `WEST`, `NORTH` . . . (the current enemies at that location, are shown before the prompt symbol >.

When running the game, you can use the command `TRAIN` to train all the neural networks or `TRAIN ENEMY_NAME` to train a specific character. Trained neural networks are automatically saved to the `./resources/neural` directory. Adjusting the variable `FORCE_RETRAIN_ON_START` in the Runner, will force the NNs to be trained automatically on the start of the game, or disable it, allowing you to load pretrained NNs.

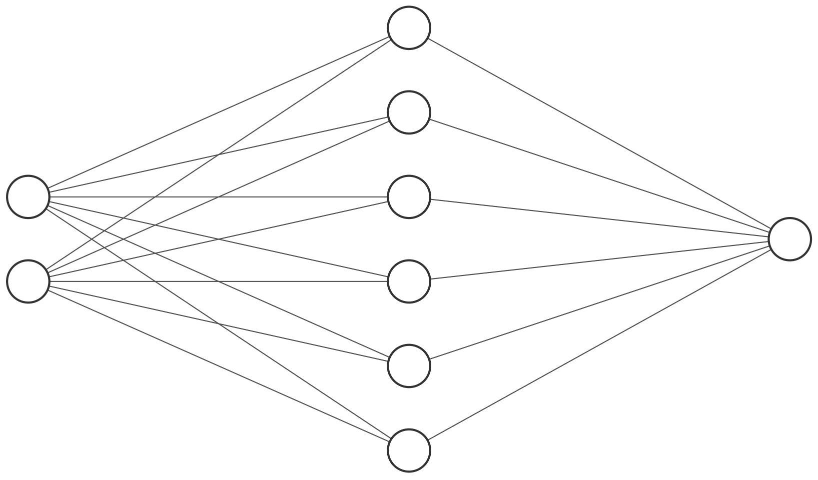
The `GenerateTrainingData` class generates training data for each character and stores it in `./resources/neural` as a `*.csv`. (This automatically happens upon starting the game). Neural networks can become too familiar with the training data, so I'm also generating some validation data that the neural networks haven't seen. This is put in the same directory as the training data. You can test the validation data on all characters using the `VALIDATE` command, or on a specific character using `VALIDATE ENEMY_NAME`.

Each character has a tolerance range of error when checking the validation data, `Goblin: 2`, `Troll: 0.08` and the `Imp` is absolute integer checking. I'm using this as a metric to determine the best loss function to use for the regression characters (`Goblin` and `Troll`). I've had success with the MAE for the Goblin and the MSE for the Troll.

For each of the NNs, I've used the smallest hidden layer sizes possible that perform well with the validation data. This will help reduce training time. The ranges of the validation data is the same as the training data, but the only difference is that input values are randomly generated.

1. Goblin

Type: **Regression**



Inputs: 0. Attack, 1. Defense **Outputs:** 0. Damage **Loss:** MAE

Input Layer: *Size:* 2
Hidden Layer: *Activation:* ReLU *Size:* 6
Output Layer: *Activation:* Linear *Size:* 1

The purpose of this NN is to learn the function below (The `Imp` and the `Troll` are also learning functions). The goblin needs to calculate its response attack (outputting a value), it will need to factor in the defence of the weapon, and goblins are 25% weaker than the player.

```
private static final BiFunction<Double, Double, Double> goblinAttack =
    (attack, defence) -> ((attack + defence) * 0.75);
```

The NN struggled with the validation data, until I normalised the input values, I normalised them between `0` and `1`. I used ReLU because its quick, and it fit the problem, being regression.

Deciding on a loss function, based on the amount of epochs it took to train, was a bad metric; as I've said above, generating correct values for unseen data is more favorable, plus you only have to train a network once, so epoch count shouldn't be taken into account. The epoch counts in the table below are very volatile because the length of time it takes to train an NN is determined by the initial random weights. I settled on MAE because it had the lowest error count for the validation data.

Loss	Attempt 1 - Epochs	Attempt 2 - Epochs	Attempt 3 - Epochs	Mean Epochs
MAE	31351	37087	47648	38695
MSE	85	10197	95	3459
SSE	94	16394	89	5525

Training Data:
0. Input Attack: `n >= 0` and `n <= 100`
1. Input Defence: `n >= 0` and `n <= 100`

Output Damage: `n >= 0` and `n <= 150`

The code below is used to produce the training data. It iterates through the possible input values of attack and defence in steps of `5`.

```
for (double attack = 0; attack <= 100; attack += 5) {
    for (double defence = 0; defence <= 100; defence += 5) {
        double output = goblinAttack.apply(attack, defence);

        pw.printf("%.0f, %.0f, %.2f\n",
            attack, defence, output);
    }
}
```

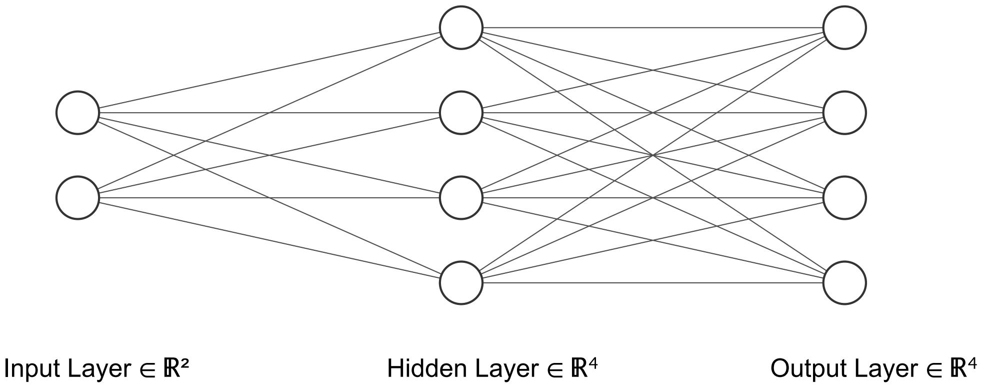
Here is a sample of the generated training data (only showing first 10 records) (`goblin_training.csv`):

```
# 1. Attack, 2. Defence, 3. Output
0, 0, 0.00
0, 10, 7.50
0, 20, 15.00
0, 30, 22.50
0, 40, 30.00
0, 50, 37.50
0, 60, 45.00
0, 70, 52.50
0, 80, 60.00
0, 90, 67.50
0, 100, 75.00
...
```

Validation Data:
To produce the validation data I randomly generated values in the range of the attack and defence, and put them into the `goblinAttack` function and noted the output, this is done `100` times. Validation data for the `Imp` and `Troll` is generated using the same technique.

2. Imp

Type: **Classification**



Inputs: 0. Attack, 1. Defense **Output:** 0. One Hot Vector **Loss:** CEE

Input Layer: *Size:* 2
Hidden Layer: *Activation:* Leaky ReLU *Size:* 4

Output Layer: *Activation:* Linear   *Size:* 4

The purpose of this NN is to learn the function below, where the attack and defence of the weapon are added together, minimum possible value is `0` and the maximum is `200`. This value is divided by `50`, and the floor division can have `4` possible values (`0`, `1`, `2` and `3`). Each of these values equates to a tier of attack from the Imp. These values being `{12.5, 25, 37.5, 50}`, the output of the NN will be the index.

```
private static final BiFunction<Double, Double, Double> impAttack =
    (attack, defence) -> Math.floor((attack + defence) / 50);
```

I found success normalising the input between `-1` and `0`, leading to better matches with the validation data. I then had to use Leaky ReLU to avoid the dead ReLU problem, due to the input having negative values. Aicme4J automatically applies softmax on the output layer, returning the index of highest value, the Imp will now use that attack tier on the player.

The outputs of the function `impAttack` above, need to be converted to one hot vectors, I made a method in the `Imp` called `genOneHotVector` that can do this.

```
0 => {1, 0, 0, 0}
1 => {0, 1, 0, 0}
2 => {0, 0, 1, 0}
3 => {0, 0, 0, 1}
```

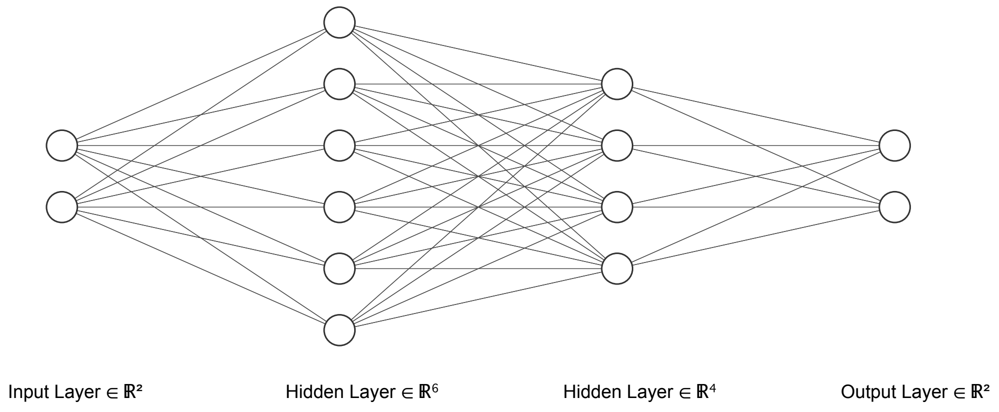
I used the CEE loss function as it was the only loss function in Aicme4J that was most appropriate for classification.

Training/Validation Data:
The training/validation data generated the same way as the Goblin, only difference is the function it calls being `impAttack`, this returns an index, that will be turned into one hot vectors during training.

First 10 records of the generated training data (`imp_training.csv`):
```
# 1. Attack, 2. Defence, 3. Output Index
0, 0, 0
0, 10, 0
0, 20, 0
0, 30, 0
0, 40, 0
0, 50, 1
0, 60, 1
0, 70, 1
0, 80, 1
0, 90, 1
...
```

3. Troll

Type: **Regression**



Inputs: 0. Attack, 1. Sharp **Outputs:** 0. Punch Damage, 1. Kick Damage **Loss:** MSE

Input Layer: *Size:* 2
Hidden Layer 0: *Activation:* ReLU *Size:* 6
Hidden Layer 1: *Activation:* Hyperbolic Tangent *Size:* 4
Output Layer: *Activation:* Linear *Size:* 2

This NN will be learning two functions unlike the two above, with the output of the first going into index `0` of the output layer (Troll punch), and the second function's output going into index two (Troll kick). These are two features, and this is a more complex problem requiring an additional hidden layer.

```
BiFunction<Double, Boolean, Double> trollAttackPunch =
    (attack, sharp) -> (attack * (sharp ? 0.2 : 0.3));
BiFunction<Double, Boolean, Double> trollAttackKick =
    (attack, sharp) -> (attack * 0.3) * (sharp ? 1.25 : 1)
```

The input data and the output data are both normalised between `-1` and `1`, this means the two outputs of the NN, will need to be denormalised, I am doing this with the `denormalise` function in the `Troll` class. I found scaling between `-1` and `1` led to less errors when checking validation data. Unlike the Goblin this is two value output regression. `NeuralNetwork.process` only returns one value, so I am using the `NeuralNetwork.getOutputLayer()` to extract the two values.

I've tried swapping the positions of each of the activator functions, and found this configuration works best. The tanh is a non-linear function, this is useful in learning complex problems like this one.

Training Data:
0. Input Attack: `n >= 0` and `n <= 100`
1. Input Sharp: `{0, 1}`

0. Output Punch Damage: `n >= 0` and `n <= 30`
1. Output Kick Damage: `n >= 0` and `n <= 37.5`

This is the code used to produce the training data. Similar to the previous enemies above, this is generating the possible inputs/outputs by iterating through the range of the inputs. Notice steps of `2` this time for attack, this is due to the complexity of the problem, which requires more training data. Sharp is a boolean value, being true or false.

```
for (double attack = 0; attack <= 100; attack += 2) {
    for (int sharp = 0; sharp <= 1; sharp++) {
        double resultPunch =
            trollAttackPunch.apply(attack, sharp == 1);
        double resultKick =
            trollAttackKick.apply(attack, sharp == 1);

        pw.printf("%.0f, %d, %.2f, %.2f\n",
            attack, sharp, resultPunch, resultKick);
    }
}
```

First 10 records of the generated training data (`troll_training.csv`):

```
# 1. Attack, 2. Sharp, 3. Output Punch, 4. Output Kick
0, 0, 0.00, 0.00
0, 1, 0.00, 0.00
2, 0, 0.60, 0.60
2, 1, 0.40, 0.75
4, 0, 1.20, 1.20
4, 1, 0.80, 1.50
6, 0, 1.80, 1.80
6, 1, 1.20, 2.25
8, 0, 2.40, 2.40
8, 1, 1.60, 3.00
...
```