Artificial Neural Networks, or ANN in short, is a machine learning algorithm that mimics the human brain. This makes the algorithm able to make decisions based on learned patterns.

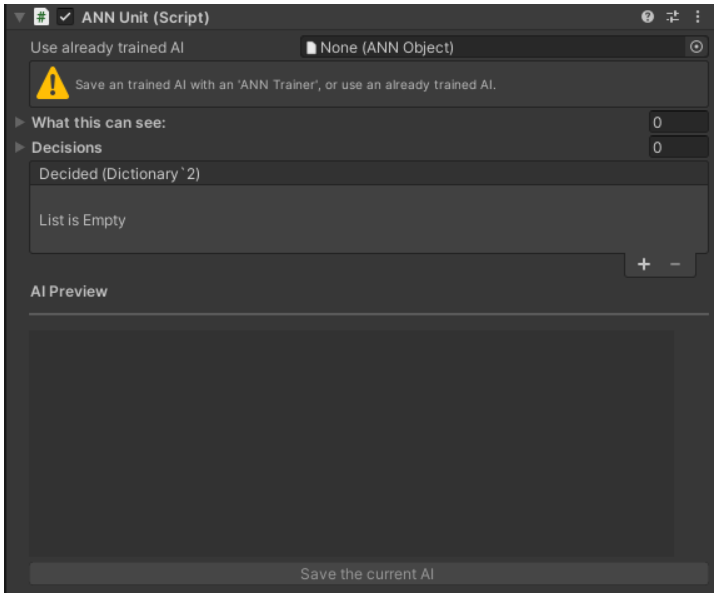
I know that ANNs can be quite complex and daunting if you don't put hours of research into it. That's what an approachable ANN is for. It simplifies the usage of the algorithm from a user perspective, allowing anyone with basic game development skills to train their perfect AI.

This documentation will cover everything you need to know and more. Here is a full table of contents:

How to use

This section covers how you can use Approachable to create the perfect AI. It will teach you how to set up the ANN unit, how to train it, and how to load a trained AI.

Prepare the ANN Unit



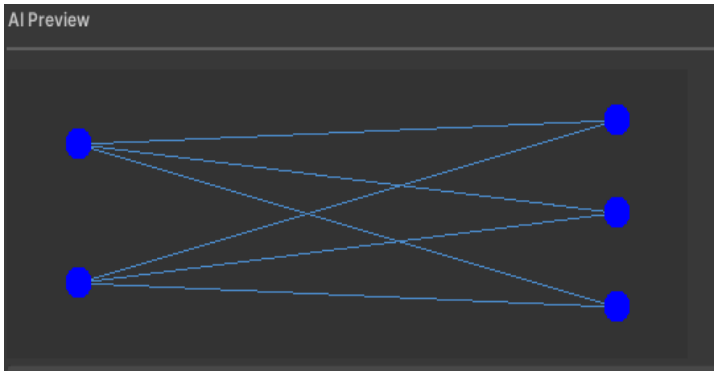
ANN units are the centerpiece of any AI. Its main job is to make decisions, so you may think of it as a brain. To enable it, you first have to attach it to the Gameobject that the AI is supposed to control.

Of course, you can't think if you don't have anything to think about. The ANN units base their decision based on what they can see. The first step is to decide that. In order to do this, you have to create senses. There are all kinds of senses provided already like the RayVision2D, that can measure the distance to another object in a straight line.

Simply attach the senses you need to any gameobject, before referencing it in the "What this can see" list.

Remember: the AI can only see what is referenced in the list. Make sure that it gets all necessary information it needs to play your game.

Now that the AI can see, it needs to know what it can possibly do. To do that, simply enter the names of the decisions as strings into the "Decisions" list. Those names will be used later when it comes to implementing the made decisions.

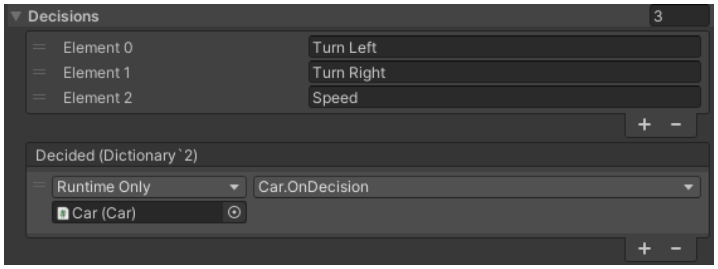


By now you have probably noticed that the AI preview has changed while editing the senses and decisions.

I'm going about the details more in depth in the "How it Works" section. For now, all you need to know is that by default, the AI associates everything it can see (dots on the left side) with any possible decision (dots on the right side).

Implement the AI decisions

As mentioned previously, the AI only makes decisions. It's up to you to actually perform them. Luckily, this process is really easy. All you have to do is to subscribe to the "Decided" event. The parameter includes a dictionary that matches every possible decision with a value from -1 to 1. It's up to you to interpret these numbers in the context of your game.



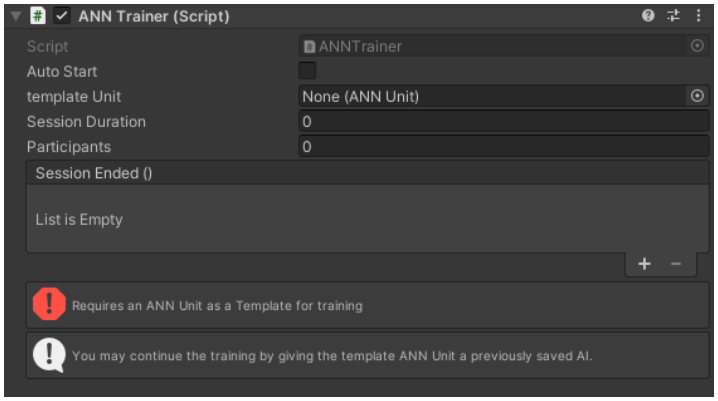
This example on the right is for a racing game. The "Speed" decision impacts how fast and in which direction the car is supposed to drive, meanwhile the "Turn Left" and "Turn Right" decisions decide if the car should turn at all, and if yes, by how much.

```
public void OnDecision(Dictionary<string, float> decisions)
{
    speed = decisions["Speed"] * maxSpeed;

    if (decisions["Turn Left"] > 0)
    {
        transform.Rotate(Vector3.forward * -150 * decisions["Turn Left"] * Time.deltaTime);
    }

    if (decisions["Turn Right"] > 0)
    {
        transform.Rotate(Vector3.forward * 150 * decisions["Turn Right"] * Time.deltaTime);
    }
}
```

Train the the AI

Now the AI can see, think, and perform the thought of actions. The problem is just that it's not really good at thinking. That's where training comes into action.
The training process is similar to biological evolution. A bunch of ANN Units, that each are little different from each other, are playing the game at the same time.
Once a certain time is over, each one gets ranked based on how good it did. The best ones are then getting "children", which replace the worst ones. This will result in the AI getting better and better at playing your game.

Luckily the ANN Trainer does a sizable chunk of the work for you.



First Attach the script to any Gameobject that is NOT an ANN Unit.

Then you can reference the previously set up ANN Unit in the "template" slot. That unit will be the starting point of the training.

Next, you have to define the duration of each training session in seconds. That's how long each generation will have until it's rated. The more time a generation has, the more the AI can learn how good it's doing. Note however that the

learning process can take quite a lot of generations, depending on how complex your game is. So you don't want to keep it too long.

Finally, you have to define the number of participants. Basically how big each generation is. The higher the number, the more ANN Units are trying out stuff at the same time, the faster the AI can learn. However, it will impact the frame rate during training, as each participant will result in a new instance of the template unit.

There's only more things to do before you can begin the training. You have to make sure to tell the AI when it does something good and when it does something bad. To do that, simply call the "Reward(float)" or "Punish(float)" method of the ANN Unit to punish or reward it by a certain amount.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    var unit = collision.gameObject.GetComponent<ANNUnit>();

    if (unit == null) return;

    unit.Punish(1);
}
```

If you notice throughout the training that the AI does not get any better over a longer period of time and is not playing the game the way you wanted it to play, you might want to revisit how you reward and punish the ANN Units. Chances are that the AI found a way to maximize their rewards in a way you didn't expect.

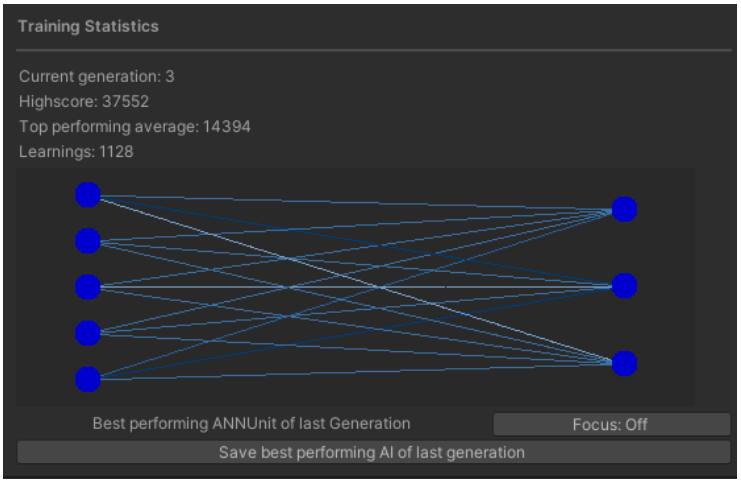
Now you can begin the training by entering Unity's play mode, which lets a bunch of ANN Units play your game at once. Make sure to put them into as many different situations as possible during the training session. This way, it can learn the ins and outs of your game.

Throughout the training, you can track the progress in the statistics section of the ANN Trainer.

The generation shows you how many training sessions were already completed.

The high score is self explanatory, but the top performing average is probably the most important one. It tells you how good the top 50% of each generation are on average.

The other 50% are trying out new tactics, which makes them usually lose the game. I say usually, because some of them will probably be better than the ones from the previous generation.



That's what the learning score is for. It tells you how much better the top 50% got after each generation.

Lastly, you can also see a visualization of how complex the best performing AI of the previous generation was. While this information does not help you directly in the training process, it gives you a nice idea of how the AI is evolving throughout the training.

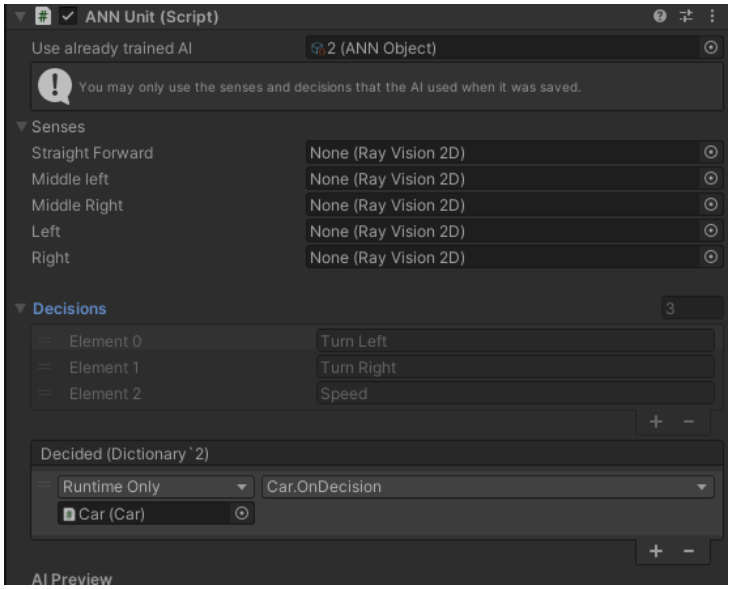
Other than the statistics, you also have a "Focus" button if the scene becomes too messy for you. Pressing the button hides all current training units, except the best one from the previous generation.

Once you are happy with how the AI plays your game, you are able to save it as a scriptable object with the save button.

Load the AI

Saved AIs appear as scriptable objects in your project structure.
Selecting it gets a brief summary about how complex it is, when it was created and how successful it was, as well as which senses decisions it used during training.

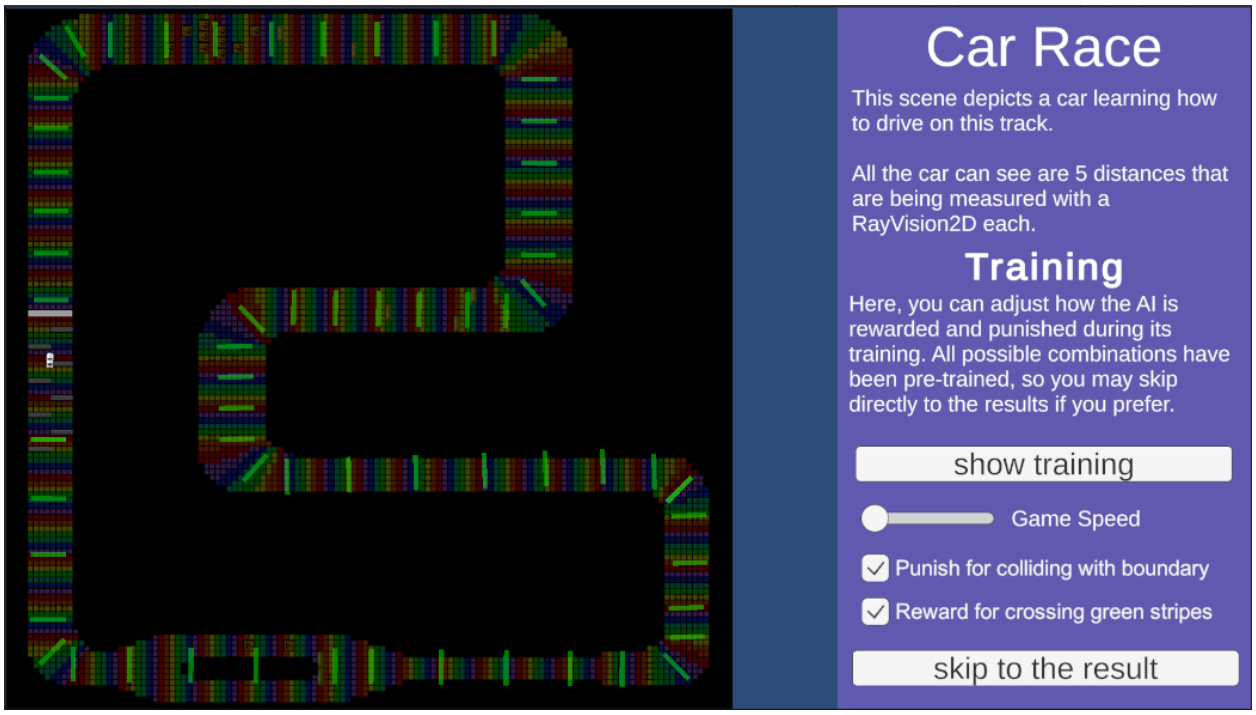
Now that you have saved your AI no longer need the ANN Trainer to play with it. All you have to do to load the saved AI is to reference it in the "use already trained AI" slot of the ANN Unit.



Note that you cannot change the sense types or decisions of an already saved AI. You need to train a new AI if you want to make any changes to these parameters.

And that's it. From now on, the ANN Unit will behave exactly as trained. You can even use it at different levels and it will try to use the solutions to similar problems it had to overcome during its training.

Sample Scene



If you need a more practical example of how to use the AI, you may want to check out the provided example scene for a simple racing game. Here can see the training process and are able to experiment a bit with how certain rewards and punishments influence the AI behavior.

Starting the scene automatically lets you see how the trained car drives along the track.

If you want to see the training in action, you can press the “show training” button.
Try for yourself what happens if you adjust the training settings. You may also directly skip to the pre-trained results if you want to.

Here is a short overview of why the AI behaves like it does under specific training circumstances:

No rewards or punishments:
Obviously, the AI can’t learn if you don’t tell it how it's doing. The results are completely random and only highlight how important the correct rewarding/punishing is throughout the training process.

Only punishing it for hitting boundaries:
What’s the best way to never hit a wall? Sure, you could just drive in circles, just drive very slowly, or even more save: not move at all. This is a great example to be careful what you wish for. The car may not do exactly what you wanted it to do, but it most certainly found a way to get

the least amount of punishment. Be mindful of potential exploits when it comes to AI rewards and punishments.

Only rewarding it for crossing green stripes for the first time:
Now we are talking. The AI finally starts to race along the track. The constant stripes motivate the AI to constantly move forwards and the faster the AI gets, the more time it has to cross even more stripes before the training session is over. However, you should also notice that the AI drives quite recklessly and constantly strifes the track borders. It also did waste quite a lot of time during its training moving forward by repeatedly driving into a wall…

Rewarding the AI for crossing green stripes and punishing it for colliding with the boundary:
The best way to train the car in this example. It not only motivates the AI to drive forward via the stripes, it also actively discourages it from getting forward by strifing the wall. The AI would have figured out that driving too close to a wall is not the optimal way to move forward, but this punishment speeds up the learning process.

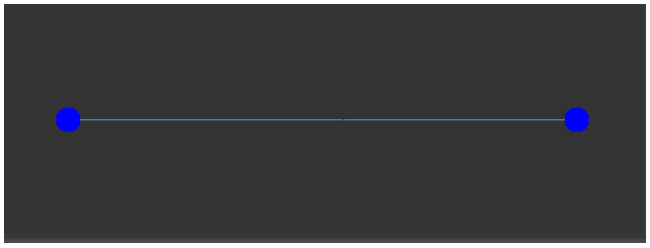
In the end, there is no perfect way of training any AI. It does take a while of experimentation to figure out the best way to reward and punish them.
What I advise you to do when creating your own reward structure is to keep looking at the training process every now and then. Nothing feels worse than letting the AI do its thing for a couple of hours, only to come back and realize that they found an exploit in the way you reward and punish them.

How it Works

Now that you know how to use the AI, you might have become curious about how it works exactly. Feel free to look through the fully commented source code or keep reading this section if you want a more guided learning experience. I won't go over everything in great detail, but it should give you a rudimentary overview of the algorithm. I will also provide a couple of sources if you want to go even deeper into the topic.

Structure of an ANN

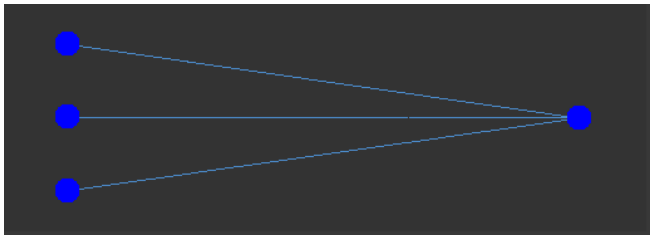
As mentioned in the introduction, the ANN algorithm is made to mimic our human brains on a basic level. This means that it too consists of neurons that are connected with each other. Take a look at a visualization of the most basic ANN possible:



most basic ANN

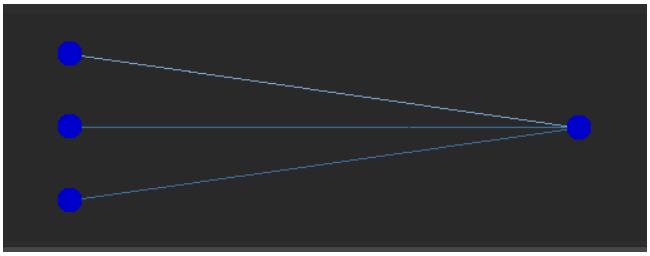
What you can see here is that two neurons are connected to each other. One neuron on the left where the information enters the algorithm (Input Neuron), and another on the right where the information leaves the algorithm (Output Neuron). Between them is a line that connects these. This means that the AI bases the value of the output neuron purely on a single input.

With that in mind take a look at this ANN:



slightly more complex ANN

This ANN still has only one output neuron, but three input neurons. It now considers three different things when it comes to making a decision. Seems easy so far, doesn't it?
But what if we want the first input to matter more for the decision outcome than the other two? That's where weights come into play. Each of these connections is associated with one that gets multiplied with the neuron value prior to it.



ANN with different weights

Here, the connection from the first input neuron to the only output neuron is visualized as brighter than the others to indicate that it has a higher weight, meaning a higher importance when it comes to the value of the output neuron.

This is how ANNs operate at a very basic level. Most of them will have more neurons, but the general idea stays the same. Values enter the ANN through the neurons on the left and leave the neurons on the right. The value of a neuron is based on the neurons that connect to it, with each connection having a weight that decides its importance.

Learning algorithms

Now that we know how the ANN operates, we can take a look at the arguably more complex topic: the training process.

The goal of every training process is to adjust the neurons and weights of each ANN in such a way that the ANN is capable of completing the given task. In theory, you may be able to do this manually, after all it's just a bunch of float values, however ANNs usually end up having hundreds, if not thousands, of different neurons to solve a task. That's why learning algorithms were invented to do this task for us.

Backpropagation

The most common algorithm for this is a mathematical approach called backpropagation. This way of learning requires you to already have a big chunk of training data. The idea is that the network makes a decision, you then tell it how far it was off from the ideal solution, before calculating how to adjust these neurons and weights to get closer to the ideal solution. However the need for a high quantity of training data makes it rather impractical for game development.

Instead, it is most commonly used for image recognition tasks. If you want to learn more about backpropagation, I wholeheartedly recommend this series of articles that goes over the classic example of recognizing handwritten digits.
https://www.3blue1brown.com/lessons/neural-networks#title

NEAT

If you took close attention to the “How to use” section, you probably already noticed that this AI learns in an evolutionary way. The algorithm it makes use of is called NEAT, which is short for neuroevolution of augmenting topologies.

The algorithm essentially boils down to to the following:

1. Create a population of ANN Units

Then the following steps happen over and over again:

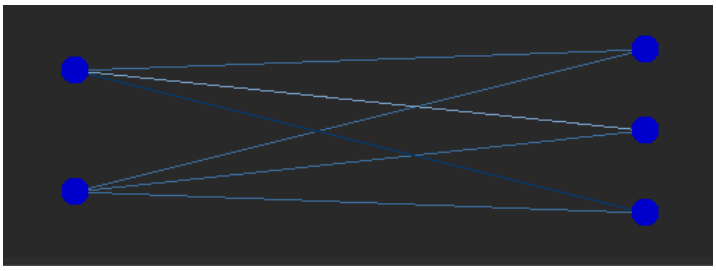
2. Let the population play the game for a certain amount of time
3. Evaluate the fitness of the generation and discard the worst 50% of it.
4. The remaining 50% are creating offsprings to replace the lost 50%. Essentially combining the structure of two different ANNs

Let’s take a closer look at these steps:

Create a population of ANN Units

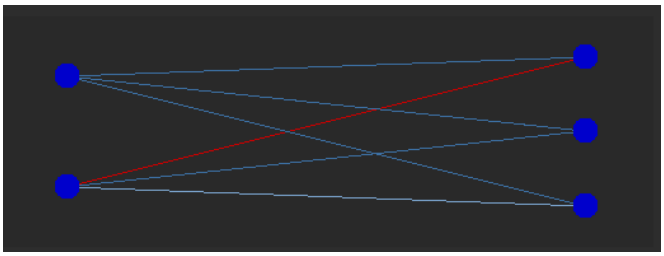
This process seems as simple as instantiating a bunch of instances of the template ANN Unit that you reference in the ANN Trainer. However, there is not much of a point in having a bunch of ANNs if each one is the exact same. That’s where mutations come into play to slightly alter each ANN more or less. They are a bunch of modifiers that each have a small chance of getting applied.

This asset makes use of the following ones:



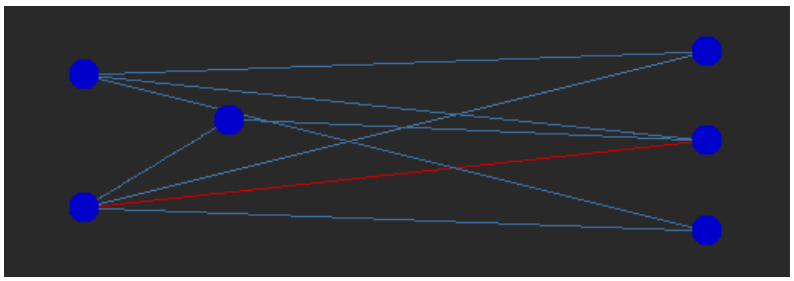
Increasing/Decreasing the weights

Here, each connection has a small chance to either increase (getting brighter) or decrease (getting dimmer) their respective weight value. The higher the weight, the more important the neuron the connection is coming from is for the decision of the neuron that the connection goes to.



Disable/Enable a random link

Here, each enabled connection has a small chance to get disabled (turning red), and each disabled connection has a small chance to get enabled. Disabled connections are not taken into consideration when making a decision. In this case, the information that the ANN gets from the second input neuron has no impact on the decision of the first output neuron.



Adding a new Neuron between two existing ones

Finally, the most important modifier. Here, a new neuron is created between the input and output neurons. Those are called hidden neurons, because the user doesn't interact with them directly. This means that the network bases one decision on a previous decision, allowing it to solve more complex problems.

During this process, a random connection gets disabled while creating one of those hidden neurons.
This new neuron then comes with two new connections that lead from the disabled connection, to the new neuron, to the target of the disabled connection. Essentially putting one neuron in between.

Evaluating the fitness

This step is pretty straight-forward. Fitness is just a float value that gets used to evaluate whether or not an AI is worth keeping. In other words, how well did it do its task? That's basically all you are doing when you reward or punish an ANN unit during its training. You increase or decrease its fitness value, making the better-performing units more likely to survive until the next generation.

Creating offsprings

The 50% with the highest fitness are then taken over by the new generation. In addition, they are creating offspring to replace the lost 50%. The goal of this process is to combine two different ANNs into a new one. This increases the likelihood of creating a new ANN that is at least as competitive as the parents.

The short version of how this works is that two parents are selected. The parent with the higher fitness is the dominant partner, while the other one is recessive. The offspring keep the general structure of the dominant one while taking over elements of the recessive one.

The actual code implementation of the whole NEAT algorithm is a bit more complicated. So feel free to look through the source code or read through the following article that summarizes the original paper if you want to learn even more about it:

https://macwha.medium.com/evolving-ais-using-a-neat-algorithm-2d154c623828

In short

The algorithm is made of many interconnected neurons. Information enters the ANN through the input neurons, where it gets calculated through different connections with the output neurons.

This ANN learns through the NEAT algorithm, which constantly creates a population, evaluates their fitness after a period of time, and then replaces the weaker ones with the children of the remaining.

Wiki

You can find a full overview of the available classes and methods in the wiki:
https://azure-gwenore-45.tiiny.site