7
Learning Techniques

In this chapter, we will explore the world of machine learning through the following topics:

- Predicting actions with an N-Gram predictor
- Improving the predictor: Hierarchical N-Gram
- Learning to use a Naïve Bayes classifier
- Learning to use decision trees
- Learning to use reinforcement
- Learning to use artificial neural networks

.Introduction

In this chapter, we will explore the field of machine learning. This is a very extensive and intrinsic field in which even AAA titles have a hard time due to the amount of time that the techniques require for fine-tuning and experimentation.

However, the recipes that are contained in this chapter will give us a great head start in our endeavor to learn and apply machine-learning techniques to our games. They are used in several different ways, but the one we usually appreciate the most is difficulty adjustment.

Finally, you are advised to complement the recipes with the reading of more formal books on the subject, in order to gain theoretical insights that lie beyond the scope of this chapter.

Predicting actions with an N-Gram predictor

Predicting actions is a great way to give players a challenge by going from random selection to selection based on past actions. One way to implement learning is by using probabilities in order to predict what the player will do next, and that's what an N-Gram predictor does.

To predict the next choice, N-Gram predictors hold a record of the probabilities of making a decision (which is usually a move), given all combinations of choices for the previous *n* moves.

Getting ready...

This recipe makes use of general types. It is recommended that we have at least a basic understanding of how they work because it's critical that we use them well.

The first thing to do is implement a data type for holding the actions and their probabilities; we'll call it `KeyDataRecord`.

The `KeyDataReconrd.cs` file should look like this:

```
using System.Collections;
using System.Collections.Generic;

public class KeyDataRecord<T>
{
    public Dictionary<T, int> counts;
    public int total;

    public KeyDataRecord()
    {
        counts = new Dictionary<T, int>();
    }
}
```

How to do it...

Building N-Gram predictor is divided into five big steps. They are as follows:

1. Create the general class within a file named exactly the same:

```
using System.Collections;
using System.Collections.Generic;
using System.Text;

public class NGramPredictor<T>
{
```

```
    private int nValue;
    private Dictionary<string, KeyDataRecord<T>> data;
}
```

2. Implement the constructor for initializing the member variables:

```
public NGramPredictor(int windowSize)
{
    nValue = windowSize + 1;
    data = new Dictionary<string, KeyDataRecord<T>>();
}
```

3. Implement a static function for converting a set of actions into a string key:

```
public static string ArrToStrKey(ref T[] actions)
{
    StringBuilder builder = new StringBuilder();
    foreach (T a in actions)
    {
        builder.Append(a.ToString());
    }
    return builder.ToString();
}
```

4. Define the function for registering a set of sequences:

```
public void RegisterSequence(T[] actions)
{
    string key = ArrToStrKey(ref actions);
    T val = actions[nValue - 1];
    if (!data.ContainsKey(key))
        data[key] = new KeyDataRecord<T>();
    KeyDataRecord<T> kdr = data[key];
    if (kdr.counts.ContainsKey(val))
        kdr.counts[val] = 0;
    kdr.counts[val]++;
    kdr.total++;
}
```

5. Finally, implement the function for computing the prediction of the best action to take:

```
public T GetMostLikely(T[] actions)
{
    string key = ArrToStrKey(ref actions);
    KeyDataRecord<T> kdr = data[key];
    int highestVal = 0;
    T bestAction = default(T);
    foreach (KeyValuePair<T,int> kvp in kdr.counts)
```

```
            {
                if (kvp.Value > highestVal)
                {
                    bestAction = kvp.Key;
                    highestVal = kvp.Value;
                }
            }
            return bestAction;
        }
```

How it works...

The predictor registers a set of actions according to the size of the window (the number of actions to register in order to make predictions) and assigns them a resulting value. For example, having a window size of 3, the first three are saved as a key to predict that it's possible that the fourth one may follow.

The prediction function computes how likely it is for an action to be the one that follows, given a set of previous actions. The more registered actions, the more accurate the predictor will be (with some limitations).

There is more...

It is important to consider that the object of type T must override both the `ToString` function and Equals function in an admissible way for it to work correctly as an index in the internal dictionaries.

Improving the predictor: Hierarchical N-Gram

The N-Gram predictor can be improved by having a handler with several other predictors ranging from 1 to n, and obtaining the best possible action after comparing the best guess from each one of them.

Getting ready...

We need to make some adjustments prior to implementing the hierarchical N-Gram predictor.

Add the following member function to the `NGramPredictor` class:

```
public int GetActionsNum(ref T[] actions)
{
    string key = ArrToStrKey(ref actions);
```

```
        if (!data.ContainsKey(key))
            return 0;
        return data[key].total;
    }
```

How to do it...

Just like the N-Gram predictor, building the hierarchical version takes a few steps:

1. Create the new class:

```
using System;
using System.Collections;
using System.Text;

public class HierarchicalNGramP<T>
{

    public int threshold;
    public NGramPredictor<T>[] predictors;
    private int nValue;
}
```

2. Implement the constructor for initializing member values:

```
public HierarchicalNGramP(int windowSize)
{
    nValue = windowSize + 1;
    predictors = new NGramPredictor<T>[nValue];
    int i;
    for (i = 0; i < nValue; i++)
        predictors[i] = new NGramPredictor<T>(i + 1);
}
```

3. Define a function for registering a sequence, just like its predecessor:

```
public void RegisterSequence(T[] actions)
{
    int i;
    for (i = 0; i < nValue; i++)
    {
        T[] subactions = new T[i+1];
        Array.Copy(actions, nValue - i - 1, subactions, 0, i+1);
        predictors[i].RegisterSequence(subactions);
    }
}
```

4. Finally, implement the function for computing the prediction:

```
public T GetMostLikely(T[] actions)
{
    int i;
    T bestAction = default(T);
    for (i = 0; i < nValue; i++)
    {
        NGramPredictor<T> p;
        p = predictors[nValue - i - 1];
        T[] subactions = new T[i + 1];
        Array.Copy(actions, nValue - i - 1, subactions, 0, i + 1);
        int numActions = p.GetActionsNum(ref actions);
        if (numActions > threshold)
            bestAction = p.GetMostLikely(actions);
    }
    return bestAction;
}
```

How it works...

The hierarchical N-Gram predictor works almost exactly like its predecessor, with the difference being that it holds a set of predictors and computes each main function using its children. Registering sequences, or finding out the most likely future action, works by decomposing the set of actions and feeding the children with them.

Learning to use Naïve Bayes classifiers

Learning to use examples could be hard even for humans. For example, given a list of examples for two sets of values, it's not always easy to see the connection between them. One way of solving this problem would be to classify one set of values and then give it a try, and that's where classifier algorithms come in handy.

Naïve Bayes classifiers are prediction algorithms for assigning labels to problem instances; they apply probability and Bayes' theorem with a strong-independence assumption between the variables to analyze. One of the key advantages of Bayes' classifiers is scalability.

Getting ready...

Since it is hard to build a general classifier, we will build ours assuming that the inputs are positive- and negative-labeled examples. So, the first thing that we need to address is defining the labels that our classifier will handle using an `enum` data structure called `NBCLabel`:

```
public enum NBCLabel
{
    POSITIVE,
    NEGATIVE
}
```

How to do it...

The classifier we'll build only takes five great steps:

1. Create the class and its member variables:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class NaiveBayesClassifier : MonoBehaviour
{
    public int numAttributes;
    public int numExamplesPositive;
    public int numExamplesNegative;

    public List<bool> attrCountPositive;
    public List<bool> attrCountNegative;
}
```

2. Define the `Awake` method for initialization:

```
void Awake()
{
    attrCountPositive = new List<bool>();
    attrCountNegative = new List<bool>();
}
```

3. Implement the function for updating the classifier:

```
public void UpdateClassifier(bool[] attributes, NBCLabel label)
{
    if (label == NBCLabel.POSITIVE)
    {
        numExamplesPositive++;
```

```
            attrCountPositive.AddRange(attributes);
        }
        else
        {
            numExamplesNegative++;
            attrCountNegative.AddRange(attributes);
        }
    }
```

4. Define the function for computing the Naïve probability:

```
public float NaiveProbabilities(
        ref bool[] attributes,
        bool[] counts,
        float m,
        float n)
{
    float prior = m / (m + n);
    float p = 1f;
    int i = 0;
    for (i = 0; i < numAttributes; i++)
    {
        p /= m;
        if (attributes[i] == true)
            p *= counts[i].GetHashCode();
        else
            p *= m - counts[i].GetHashCode();
    }
    return prior * p;
}
```

5. Finally, implement the function for prediction:

```
public bool Predict(bool[] attributes)
{
    float nep = numExamplesPositive;
    float nen = numExamplesNegative;
    float x = NaiveProbabilities(ref attributes,
attrCountPositive.ToArray(), nep, nen);
    float y = NaiveProbabilities(ref attributes,
attrCountNegative.ToArray(), nen, nep);
    if (x >= y)
        return true;
    return false;
}
```

How it works...

The `UpdateClassifier` function takes the example input values and stores them. This is the first function to be called. The `NaiveProbabilities` function is the one responsible for computing the probabilities for the prediction function to work. Finally, the `Predict` function is the second one to be called by us in order to get the results of classification.

Learning to use decision trees

We already learned the power and flexibility of decision trees for adding a decision-making component to our game. Furthermore, we can also build them dynamically through supervised learning. That's why we're revisiting them in this chapter.

There are several algorithms for building decision trees that are suited for different uses such as prediction and classification. In our case, we'll explore decision-tree learning by implementing the ID3 algorithm.

Getting ready...

Despite having built decision trees in a previous chapter, and the fact that they're based on the same principles as the ones that we will implement now, we will use different data types for our implementation needs in spite of the learning algorithm.

We will need two data types: one for the decision nodes and one for storing the examples to be learned.

The code for the `DecisionNode` data type is as follows:

```
using System.Collections.Generic;

public class DecisionNode
{
    public string testValue;
    public Dictionary<float, DecisionNode> children;

    public DecisionNode(string testValue = "")
    {
        this.testValue = testValue;
        children = new Dictionary<float, DecisionNode>();
    }
}
```

The code for the `Example` data type is as follows:

```
using UnityEngine;
using System.Collections.Generic;

public enum ID3Action
{
    STOP, WALK, RUN
}

public class ID3Example : MonoBehaviour
{
    public ID3Action action;
    public Dictionary<string, float> values;

    public float GetValue(string attribute)
    {
        return values[attribute];
    }
}
```

How to do it...

We will create the `ID3` class with several functions for computing the resulting decision tree.

1. Create the `ID3` class:

```
using UnityEngine;
using System.Collections.Generic;
public class ID3 : MonoBehaviour
{
    // next steps
}
```

2. Start the implementation of the function responsible for splitting the attributes into sets:

```
public Dictionary<float, List<ID3Example>> SplitByAttribute(
        ID3Example[] examples,
        string attribute)
{
    Dictionary<float, List<ID3Example>> sets;
    sets = new Dictionary<float, List<ID3Example>>();
    // next step
}
```

3. Iterate though all the examples received, and extract their value in order to assign them to a set:

```
foreach (ID3Example e in examples)
{
    float key = e.GetValue(attribute);
    if (!sets.ContainsKey(key))
        sets.Add(key, new List<ID3Example>());
    sets[key].Add(e);
}
return sets;
```

4. Create the function for computing the entropy for a set of examples:

```
public float GetEntropy(ID3Example[] examples)
{
    if (examples.Length == 0) return 0f;
    int numExamples = examples.Length;
    Dictionary<ID3Action, int> actionTallies;
    actionTallies = new Dictionary<ID3Action, int>();
    // next steps
}
```

5. Iterate through all of the examples to compute their action quota:

```
foreach (ID3Example e in examples)
{
    if (!actionTallies.ContainsKey(e.action))
        actionTallies.Add(e.action, 0);
    actionTallies[e.action]++;
}
```

6. Compute the entropy :

```
int actionCount = actionTallies.Keys.Count;
if (actionCount == 0) return 0f;
float entropy = 0f;
float proportion = 0f;
foreach (int tally in actionTallies.Values)
{
    proportion = tally / (float)numExamples;
    entropy -= proportion * Mathf.Log(proportion, 2);
}
return entropy;
```

7. Implement the function for computing the entropy for all the sets of examples. This is very similar to the preceding one; in fact, it uses it:

```
public float GetEntropy(
        Dictionary<float, List<ID3Example>> sets,
        int numExamples)
{
    float entropy = 0f;
    foreach (List<ID3Example> s in sets.Values)
    {
        float proportion;
        proportion = s.Count / (float)numExamples;
        entropy -= proportion * GetEntropy(s.ToArray());
    }
    return entropy;
}
```

8. Define the function for building a decision tree:

```
public void MakeTree(
        ID3Example[] examples,
        List<string> attributes,
        DecisionNode node)
{
    float initEntropy = GetEntropy(examples);
    if (initEntropy <= 0) return;
    // next steps
}
```

9. Declare and initialize all the required members for the task:

```
int numExamples = examples.Length;
float bestInfoGain = 0f;
string bestSplitAttribute = "";
float infoGain = 0f;
float overallEntropy = 0f;
Dictionary<float, List<ID3Example>> bestSets;
bestSets = new Dictionary<float, List<ID3Example>>();
Dictionary<float, List<ID3Example>> sets;
```

10. Iterate through all the attributes in order to get the best set based on the information gain:

```
foreach (string a in attributes)
{
    sets = SplitByAttribute(examples, a);
    overallEntropy = GetEntropy(sets, numExamples);
    infoGain = initEntropy - overallEntropy;
```

```
        if (infoGain > bestInfoGain)
        {
            bestInfoGain = infoGain;
            bestSplitAttribute = a;
            bestSets = sets;
        }
    }
```

11. Select the root node based on the best split attribute, and rearrange the remaining attributes for building the rest of the tree:

```
node.testValue = bestSplitAttribute;
List<string> newAttributes = new List<string>(attributes);
newAttributes.Remove(bestSplitAttribute);
```

12. Iterate through all the remaining attributes. calling the function recursively:

```
foreach (List<ID3Example> set in bestSets.Values)
{
    float val = set[0].GetValue(bestSplitAttribute);
    DecisionNode child = new DecisionNode();
    node.children.Add(val, child);
    MakeTree(set.ToArray(), newAttributes, child);
}
```

How it works...

The class is modular in terms of functionality. It doesn't store any information but is able to compute and retrieve everything needed for the function that builds the decision tree. `SplitByAttribute` takes the examples and divides them into sets that are needed for computing their entropy. `ComputeEntropy` is an overloaded function that computes a list of examples and all the sets of examples using the formulae defined in the ID3 algorithm. Finally, `MakeTree` works recursively in order to build the decision tree, getting hold of the most significant attribute.

See also

- *Chapter 3*, *Decision Making*, the *Choosing through a decision tree* recipe

Learning to use reinforcement

Imagine that we need to come up with an enemy that needs to select different actions over time as the player progresses through the game and his or her patterns change, or a game for training different types of pets that have free will to some extent.

For these types of tasks, we can use a series of techniques aimed at modeling learning based on experience. One of these algorithms is Q-learning, which will be implemented in this recipe.

Getting ready...

Before delving into the main algorithm, it is necessary to have certain data structures implemented. We need to define a structure for game state, another for game actions, and a class for defining an instance of the problem. They can coexist in the same file.

The following is an example of the data structure for defining a game state:

```
public struct GameState
{
    // TODO
    // your state definition here
}
```

Next is an example of the data structure for defining a game action:

```
public struct GameAction
{
    // TODO
    // your action definition here
}
```

Finally, we will build the data type for defining a problem instance:

1. Create the file and class:

```
public class ReinforcementProblem
{
}
```

2. Define a virtual function for retrieving a random state. Depending on the type of game we're developing, we are interested in random states considering the current state of the game:

```
public virtual GameState GetRandomState()
{
    // TODO
    // Define your own behaviour
    return new GameState();
}
```

3. Define a virtual function for retrieving all the available actions from a given game state:

```
public virtual GameAction[] GetAvailableActions(GameState s)
{
    // TODO
    // Define your own behaviour
    return new GameAction[0];
}
```

4. Define a virtual function for carrying out an action, and then retrieving the resulting state and reward:

```
public virtual GameState TakeAction(
        GameState s,
        GameAction a,
        ref float reward)
{
    // TODO
    // Define your own behaviour
    reward = 0f;
    return new GameState();
}
```

How to do it...

We will implement two classes. The first one stores values in a dictionary for learning purposes, and the second one is the class that actually holds the Q-learning algorithm:

1. Create the `QValueStore` class:

```
using UnityEngine;
using System.Collections.Generic;

public class QValueStore : MonoBehaviour
{
    private Dictionary<GameState, Dictionary<GameAction, float>>
store;
}
```

2. Implement the constructor:

```
public QValueStore()
{
    store = new Dictionary<GameState, Dictionary<GameAction,
float>>();
}
```

3. Define the function for getting the resulting value of taking an action in a game state. Carefully craft this, considering an action cannot be taken in that particular state:

```
public virtual float GetQValue(GameState s, GameAction a)
{
    // TODO: your behaviour here
    return 0f;
}
```

4. Implement the function for retrieving the best action to take in a certain state:

```
public virtual GameAction GetBestAction(GameState s)
{
    // TODO: your behaviour here
    return new GameAction();
}
```

5. Implement the function for :

```
public void StoreQValue(
        GameState s,
        GameAction a,
        float val)
{
    if (!store.ContainsKey(s))
    {
        Dictionary<GameAction, float> d;
        d = new Dictionary<GameAction, float>();
        store.Add(s, d);
    }
    if (!store[s].ContainsKey(a))
    {
        store[s].Add(a, 0f);
    }
    store[s][a] = val;
}
```

6. Let's move on to the `QLearning` class, which will run the algorithm:

```
using UnityEngine;
using System.Collections;

public class QLearning : MonoBehaviour
{
    public QValueStore store;
}
```

7. Define the function for retrieving random actions from a given set:

```
private GameAction GetRandomAction(GameAction[] actions)
{
    int n = actions.Length;
    return actions[Random.Range(0, n)];
}
```

8. Implement the learning function. Be advised that this is split into several steps. Start by defining it. Take into consideration that this is a coroutine:

```
public IEnumerator Learn(
        ReinforcementProblem problem,
        int numIterations,
        float alpha,
        float gamma,
        float rho,
        float nu)
{
    // next steps
}
```

9. Validate that the store list is initialized:

```
if (store == null)
    yield break;
```

10. Get a random state:

```
GameState state = problem.GetRandomState();
for (int i = 0; i < numIterations; i++)
{
    // next steps
}
```

11. Return null for the current frame to keep running:

```
yield return null;
```

12. Validate against the length of the walk :

```
if (Random.value < nu)
    state = problem.GetRandomState();
```

13. Get the available actions from the current game state:

```
GameAction[] actions;
actions = problem.GetAvailableActions(state);
GameAction action;
```

14. Get an action depending on the value of the randomness of exploration:

```
if (Random.value < rho)
    action = GetRandomAction(actions);
else
    action = store.GetBestAction(state);
```

15. Calculate the new state for taking the selected action on the current state and the resulting reward value:

```
float reward = 0f;
GameState newState;
newState = problem.TakeAction(state, action, ref reward);
```

16. Get the `q` value, given the current game, and take action and the best action for the new state computed before:

```
float q = store.GetQValue(state, action);
GameAction bestAction = store.GetBestAction(newState);
float maxQ = store.GetQValue(newState, bestAction);
```

17. Apply the Q-learning formula:

```
q = (1f - alpha) * q + alpha * (reward + gamma * maxQ);
```

18. Store the computed `q` value, giving its parents as indices:

```
store.StoreQValue(state, action, q);
state = newState;
```

How it works...

In the Q-learning algorithm, the game world is treated as a state machine. It is important to take note of the meaning of the parameters:

- `alpha`: This is the learning rate
- `gamma`: This is the discount rate
- `rho`: This is the randomness of exploration
- `nu`: This is the length of the walk

Learning to use artificial neural networks

Imagine a way to make an enemy or game system emulate the way the brain works. That's how neural networks operate. They are based on a neuron, we call it `Perceptron`, and the sum of several neurons; its inputs and outputs are what makes a neural network.

In this recipe, we will learn how to build a neural system, starting from `Perceptron`, all the way to joining them in order to create a network.

Getting ready...

We will need a data type for handling raw input; this is called `InputPerceptron`:

```
public class InputPerceptron
{
    public float input;
    public float weight;
}
```

How to do it...

We will implement two big classes. The first one is the implementation for the `Perceptron` data type, and the second one is the data type handling the neural network:

1. Implement a `Perceptron` class derived from the `InputPerceptron` class that was previously defined:

```
public class Perceptron : InputPerceptron
{
    public InputPerceptron[] inputList;
    public delegate float Threshold(float x);
    public Threshold threshold;
    public float state;
    public float error;
}
```

2. Implement the constructor for setting the number of inputs:

```
public Perceptron(int inputSize)
{
    inputList = new InputPerceptron[inputSize];
}
```

3. Define the function for processing the inputs:

```
public void FeedForward()
{
    float sum = 0f;
    foreach (InputPerceptron i in inputList)
    {
        sum += i.input * i.weight;
    }
    state = threshold(sum);
}
```

4. Implement the functions for adjusting weights:

```
public void AdjustWeights(float currentError)
{
    int i;
    for (i = 0; i < inputList.Length; i++)
    {
        float deltaWeight;
        deltaWeight = currentError * inputList[i].weight * state;
        inputList[i].weight = deltaWeight;
        error = currentError;
    }
}
```

5. Define a function for funneling the weights with regard to the type of input:

```
public float GetIncomingWeight()
{
    foreach (InputPerceptron i in inputList)
    {
        if (i.GetType() == typeof(Perceptron))
            return i.weight;
    }
    return 0f;
}
```

6. Create the class for handling the set of `Perceptron` as a network:

```
using UnityEngine;
using System.Collections;

public class MLPNetwork : MonoBehaviour
{
    public Perceptron[] inputPer;
    public Perceptron[] hiddenPer;
    public Perceptron[] outputPer;
}
```

7. Implement the function for transmitting inputs from one end to the other of the neural network:

```
public void GenerateOutput(Perceptron[] inputs)
{
    int i;
    for (i = 0; i < inputs.Length; i++)
        inputPer[i].state = inputs[i].input;

    for (i = 0; i < hiddenPer.Length; i++)
```

```
            hiddenPer[i].FeedForward();

        for (i = 0; i < outputPer.Length; i++)
            outputPer[i].FeedForward();
    }
```

8. Define the function for propelling the computation that actually emulates learning:

```
public void BackProp(Perceptron[] outputs)
{
    // next steps
}
```

9. Traverse the output layer for computing values:

```
int i;
for (i = 0; i < outputPer.Length; i++)
{
    Perceptron p = outputPer[i];
    float state = p.state;
    float error = state * (1f - state);
    error *= outputs[i].state - state;
    p.AdjustWeights(error);
}
```

10. Traverse the internal `Perceptron` layers, but the input layer:

```
for (i = 0; i < hiddenPer.Length; i++)
{
    Perceptron p = outputPer[i];
    float state = p.state;
    float sum = 0f;
    for (i = 0; i < outputs.Length; i++)
    {
        float incomingW = outputs[i].GetIncomingWeight();
        sum += incomingW * outputs[i].error;
        float error = state * (1f - state) * sum;
        p.AdjustWeights(error);
    }
}
```

11. Implement a high-level function for ease of use:

```
public void Learn(
        Perceptron[] inputs,
        Perceptron[] outputs)
{
    GenerateOutput(inputs);
    BackProp(outputs);
}
```

How it works...

We implemented two types of Perceptrons in order to define the ones that handle external input and the ones internally connected to each other. That's why the *basic* Perceptron class derives from the latter category. The `FeedForward` function handles the inputs and irrigates them along the network. Finally, the function for back propagation is the one responsible for adjusting the weights. This *weight adjustment* is the emulation of learning.

Creating emergent particles using a harmony search

Being a musician myself, this recipe is one that is close to my heart. Imagine a group of musicians with a base theme in mind. But, they've never played with each other, and as the song changes, they must adapt to the core tones with their instruments and their styles. The emulation of this adaptation is implemented using an algorithm called harmony search.

Getting ready...

We will need to define an objective function as a delegate in order to set it up before calling the function.

How to do it...

We will now implement the algorithm in a class:

1. Create the `HarmonySearch` class:

```
using UnityEngine;
using System.Collections;

public class HarmonySearch : MonoBehaviour
{

}
```

2. Define the public inputs that are to be tuned:

```
[Range(1, 100)]
public int memorySize = 30;
public int pitchNum;
// consolidation rate
[Range(0.1f, 0.99f)]
public float consRate = 0.9f;
```

```
// adjustment rate
[Range(0.1f, 0.99f)]
public float adjsRate = 0.7f;
public float range = 0.05f;
public int numIterations;
[Range(0.1f, 1.0f)]
public float par = 0.3f;
```

3. Define a list of bounds. This is a `Vector2`, so `x` will represent the lowest bound and `y` the highest bound. The number of bounds must be equal to the number of pitches:

```
public Vector2[] bounds;
```

4. Define the private members for the algorithm:

```
private float[,] memory;
private float[] solution;
private float fitness;
private float best;
```

5. Implement the initialization function:

```
private void Init()
{
    memory = new float[memorySize, pitchNum];
    solution = new float[memorySize];
    fitness = ObjectiveFunction(memory);
}
```

6. Start defining the function for creating harmony:

```
private float[] CreateHarmony()
{
    float[] vector = new float[pitchNum];
    int i;
    // next steps
}
```

7. Iterate through the number of pitches (instruments):

```
for (i = 0; i < pitchNum; i++)
{
    // next steps
}
```

8. Compute the new number of the possible new harmonies, given a random value:

```
if (Random.value < consRate)
{
    int r = Random.Range(0, memory.Length);
    float val = memory[r, i];
```

```
        if (Random.value < adjsRate)
            val = val + range * Random.Range(-1f, 1f);
        if (val < bounds[i].x)
            val = bounds[i].x;
        if (val > bounds[i].y)
            val = bounds[i].y;
        vector[i] = val;
    }
```

9. Define the value in case it needs to be randomized:

```
    else
    {
        vector[i] = Random.Range(bounds[i].x, bounds[i].y);
    }
```

10. Retrieve the new vector:

```
    return vector;
```

11. Define the function that will make everything work:

```
    public float[] Run()
    {
        // next steps
    }
```

12. Initialize the values:

```
    Init();
    int iterations = numIterations;
    float best = Mathf.Infinity;
```

13. Call the previous functions and computations to find the best list of pitches:

```
    while (iterations != 0)
    {
        iterations--;
        float[] harm = CreateHarmony();
        fitness = ObjectiveFunction(harm);
        best = Mathf.Min(fitness, best);
        memory = harm;
    }
```

14. Return the best list of pitches:

```
    return
```

How it works...

The algorithm initializes all the values, given the public inputs and its inner members. It iterates several times in order to find the best list of pitches among the set of bounds and the different tones created using the previously defined objective function.