Play AI: Machine Learning and Video Games

Michael Joseph McCarron

note: this work is published as another book at,

https://www.amazon.com/Play-AI-Machine-Learning-Video/dp/B0BW2X9B34

ISBN: 9798378330393

CONTENTS

1 OVERVIEW OF MACHINE LEARNING

Computer Science has been enlisted in the service of Intelligence since its inception and is a direct derivative of warfare in World War 2. Video games as a direct product of military simulations serve as a secondary market for technology developed initially for military purposes. Developing from the ideals of Alan Turing, Konrad Zuse, Heinz Billing and others during WWII, Artificial Intelligence has developed over the past several decades to do more autonomous computation not involving direct human manipulation of algorithms and informing both military simulations of combat, actual targeting in combat and in the public market video games. An algorithm is basically a computer program, a set of code or libraries of code working toward computing a problem. The basic starting block of Artificial Intelligence is based in statistics, and what is known as Statistical Learning. Also, we have seen how what is considered cutting edge commercial technical advances in AI such as Generative Adversarial Networks (GANs) have been used in military defense technology for some 20 years before being 'leaked' to the commercial world. Yet, we see in the commercial application of AI severe ethical and technological problems are constantly at the forefront of the discussion of AI, yet, there is no guarantee in the covert world with limited dialogue and availability to others research that such cutting edge developments were done with due regard to code review and testing before being put into production to meet deadlines of lucrative contracts in the often black budgeted world of Defense contractors. Given that this is a technical question it is good to have at least some technical background knowledge of some of the challenges involved in developing Automation and AI. In the following an overview of Machine Learning is presented.

First, we need to understand what we say when we are talking about Machine Learning:

> *A computer program is said to learn from experience E with respect to some class of Tasks T and performance measure P if its performance at Tasks T, as measured by P, improves with experience E. --Tom Mitchell*

Machine Learning is based in statistics (similar to statistical mechanics in physics), the early work in Artificial Intelligence was done in Statistical Learning.

> Statistical Learning - a set of approaches for estimating f, f is estimated for prediction and inference, how to estimate f: parametric (parameterization) vs. non-parametric (featureless)

Statistical learning is a field of study that deals with the development and application of statistical models to understand and make predictions or decisions from data. It encompasses a wide range of techniques, including supervised learning (such as regression and classification), unsupervised learning (such as clustering and dimensionality reduction), and reinforcement learning. These methods are used in a variety of applications, including prediction, inference, and decision making, and they are often used in combination with one another to solve complex problems. Some popular statistical learning methods include linear regression, decision trees, and neural networks. Machine learning is a field of study that uses statistical techniques to give computer systems the ability to "learn" (i.e. progressively improve performance on a specific task) with data, without being explicitly programmed. Machine learning is a type

of artificial intelligence that allows systems to automatically learn and improve from experience, rather than being explicitly programmed.

There are several programming languages commonly used in the field of machine learning, including:

Python: is the most widely used programming language in the field of machine learning. It has a large number of libraries and frameworks such as TensorFlow, PyTorch, and scikit-learn, which make it easier to build and train machine learning models.

R: is a statistical programming language that has a large number of packages for data analysis and machine learning. It's widely used for developing statistical models and is popular among data scientists.

Java: is an object-oriented programming language that is well suited for developing large-scale applications. It has libraries such as Weka and Deeplearning4j for building machine learning models.

Julia: is a relatively new programming language that's gaining popularity in the field of machine learning. It's designed to be fast and easy to use, and it has a growing number of packages for machine learning, including Flux.jl and MLJ.jl.

C++: is a low-level programming language that is often used for developing high-performance machine learning algorithms. It has libraries such as TensorFlow, Caffe, and Torch for building and training machine learning models.

In this book I focus on Python for Machine Learning examples and C# for game development, a Github repo of the code is at https://github.com/autonomous019/play_ai (download code as needed). In this section we will be focusing on Python. In Python, you can use libraries such as scikit-learn for traditional machine learning tasks, like regression, classification, and clustering. For deep learning, TensorFlow and PyTorch are popular choices. These libraries provide a high-level API for building and training neural networks, and they handle the low-level implementation details for you. An important component of Python development is the use of Pip for installing modules and various dependencies for different machine learning tasks. For instance to install matplotlib so it can be imported into your notebooks you would run in the command line or in the notebook itself:

```
pip install matplotlib #run on command line
!pip install matplotlib #if running in the notebook itself
```

For developing Machine Learning with Python notebooks are used. Recently, Google Collaboratory (https://colab.research.google.com/) has become a popular place to develop using notebooks, although you can also install Anaconda and run Jupyter Notebooks (https://jupyter.org/), another on-line resource for learning and coding Python for Machine Learning, is Kaggle.com, there you can browse open-source ML

code notebooks and implement them on your own. Jupyter notebooks are a popular tool in the Python programming community, especially in the field of data science and machine learning. A Jupyter notebook is an interactive computing environment that allows you to mix code, text, and visualizations in one document. It is an excellent tool for experimenting with code, testing ideas, and documenting your work.

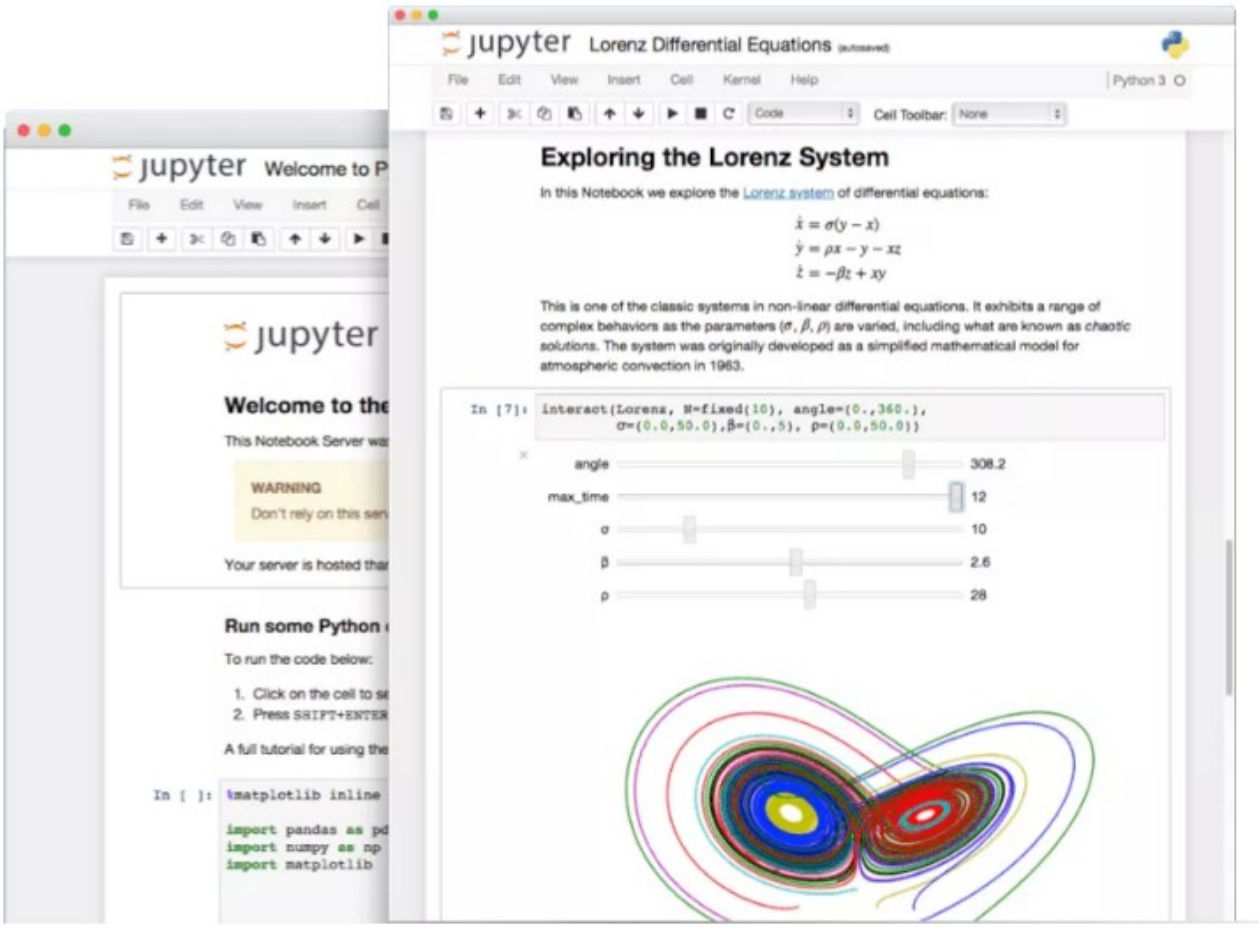
In a Jupyter notebook, you can write and run code snippets, add markdown cells for documentation, and display visualizations and outputs. The notebook allows you to see the results of your code immediately, making it easier to understand and debug your code. Jupyter notebooks also make it easy to share your work with others, as they can be easily exported to other formats such as HTML, PDF, and LaTeX.

To use a Jupyter notebook, you need to have Python installed on your computer, along with the Jupyter package. Once you have installed Jupyter, you can start a new notebook by running the command `jupyter notebook` in your terminal. This will launch a web-based interface where you can create and manage notebooks.

In a Jupyter notebook, you can write code in cells and run them by pressing Shift + Enter or by clicking the Run button in the toolbar. This allows you to see the results of your code immediately, making it an excellent tool for interactive programming and exploration.

Whether you are a beginner or an experienced Python programmer, Jupyter notebooks are a valuable tool to have in your toolkit. They make it easy to experiment with code, test ideas, and document your work, all in one place.

Jupyter notebooks are included in a programming package for Python known as Anaconda. Anaconda provides a distribution of Python and over 720 open-source packages that are pre-installed and ready to use, making it easy to get started with your project. Let’s go over the steps to install Anaconda on your computer.



Setting up a Development Environment for Machine Learning

Step 1: Download the Anaconda Installer

The first step to installing Anaconda is to download the installer. You can do this by visiting the Anaconda website (https://www.anaconda.com/products/distribution) and clicking on the "Download" button. Make sure to select the version of Anaconda that is compatible with your operating system.

Step 2: Install Anaconda

Once you have downloaded the installer, you can start the installation process. On Windows, double-click on the downloaded .exe file and follow the on-screen instructions. On Windows, click

on the downloaded executable file (.exe) and follow the installation instructions. On MacOS and Linux, open the terminal and run the following command:

```
bash Anaconda-latest-MacOSX-x86_64.sh
```

Replace Anaconda-latest-MacOSX-x86_64.sh with the name of the Anaconda installer you downloaded. Follow the on-screen instructions to complete the installation.

Step 3: Verify the Installation

To verify that Anaconda is installed, you can open a terminal or command prompt and run the following command:

```
conda –version
```

This should display the version number of Anaconda that you have installed.

Step 4: Create a Virtual Environment

Anaconda allows you to create virtual environments for your projects, which are isolated environments that have their own set of packages and dependencies. To create a virtual environment, run the following command:

```
conda create --name myenv
```

Replace myenv with the name of your virtual environment. This will create a new virtual environment with the name you specified.

Step 5: Activate the Virtual Environment

To activate the virtual environment you created, run the following command:

```
conda activate myenv
```

Replace myenv with the name of your virtual environment. This will change the active environment to the one you specified, and you will see the name of the virtual environment in the terminal prompt.

Step 6: Install Packages

Now that you have Anaconda installed and a virtual environment set up, you can start installing packages that you need for your project. To install a package, run the following command:

```
conda install package_name
```

Replace package_name with the name of the package you want to install. Anaconda comes with over 720 pre-installed packages, so you can start using them right away.

Once you have installed Anaconda and are up and running Jupyter notebooks or use Google Colab, you are ready to start importing such libraries as Scikit Learn, Pandas and Matplotlib. To work with large sets of data, programmers have come up with Dataframes to hold data and to manipulate the data, Pandas is ideal for this purpose.

Pandas is a popular Python library for data analysis and manipulation. It provides fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. One of the main data structures in Pandas is the DataFrame, which is a two-dimensional labeled data structure with columns of potentially different types.

In this article, we'll go over the basics of using Pandas, with a special focus on DataFrames. We'll start by showing you how to create a Pandas DataFrame, and then we'll go over some of the most common operations you can perform on a DataFrame.

Creating a Pandas DataFrame

You can create a Pandas DataFrame in several ways, including from a NumPy array, a list of dictionaries, or a CSV file. One of the simplest ways to create a DataFrame is from a list of dictionaries. Each dictionary represents a row in the DataFrame, and the keys in the dictionary represent the columns.

Here's an example that creates a DataFrame from a list of dictionaries:

```
import pandas as pd

data = [
    {'name': 'John', 'age': 32, 'city': 'New York'},
    {'name': 'Jane', 'age': 28, 'city': 'London'},
    {'name': 'Jim', 'age': 38, 'city': 'Paris'},
]

df = pd.DataFrame(data)
```

In this example, the data variable is a list of dictionaries, where each dictionary represents a row in the DataFrame. The pd.DataFrame function takes the data variable as input and creates a DataFrame from it.

Iterating Over a Pandas DataFrame

One of the most common operations you'll perform on a DataFrame is iteration. Pandas provides several ways to iterate over a DataFrame, including using the iterrows method.

The iterrows method returns an iterator that yields index and row data for each row. Here's an example of how to use iterrows to iterate over a DataFrame:

```
for index, row in df.iterrows():
    print(index, row['name'], row['age'], row['city'])
```

In this example, the iterrows method returns an iterator that yields the index and row data for each row in the DataFrame. The for loop iterates over the iterator and prints the index, name, age, and city for each row.

The DataFrame is one of the main data structures in Pandas and is used to store and manipulate labeled data. In this article, we've covered the basics of using Pandas, including how to create a DataFrame and how to iterate over a DataFrame using the iterrows method. Whether you're a beginner or an experienced Python developer, Pandas is an excellent library to have in your toolkit.

Using a GPU in Machine Learning: The Power of Parallel Computing

Machine learning has become a rapidly growing field in recent years, as it has become more accessible for researchers, engineers and data scientists to apply advanced techniques to a variety of problems. One of the key enabling technologies for this has been the availability of powerful Graphics Processing Units (GPUs), which have made it possible to perform computationally expensive tasks in a fraction of the time that would have been required using only a Central Processing Unit (CPU). Machine learning algorithms are often designed to operate on large amounts of data, and they can be computationally expensive to run. A typical example of this is training a neural network, which can take hours or even days on a CPU. GPUs are designed to perform many parallel computations simultaneously, and they are well suited to perform the matrix operations that are common in machine learning. This is because GPUs have thousands of cores, each of which can perform simple calculations in parallel, whereas CPUs have a few cores that are optimized for sequential computations.

Another advantage of GPUs is that they are optimized for handling large amounts of data, which makes them ideal for training large neural networks. They also have large amounts of memory, which makes it possible to store large amounts of data and intermediate results. The basic idea behind using a GPU in machine learning is to offload computationally expensive tasks from the CPU to the GPU. The CPU prepares the data and sends it to the GPU, which performs the calculations. The results are then sent back to the CPU, which processes them and uses them to update the model. The process of using a GPU in machine learning requires specialized software, such as TensorFlow, PyTorch, or CUDA. These software frameworks provide a high-level interface for programming the GPU, making it easy to implement machine learning algorithms.

GPUs are well suited to a wide range of machine learning tasks, including:

Training large neural networks: As mentioned earlier, training large neural networks can be computationally expensive, but GPUs can significantly reduce the amount of time required to train these networks.

Image and video processing: GPUs can be used to perform image and video processing tasks, such as object recognition and classification, in real-time.

Natural language processing: GPUs can be used to perform computationally expensive tasks, such as language translation and text generation, in real-time.

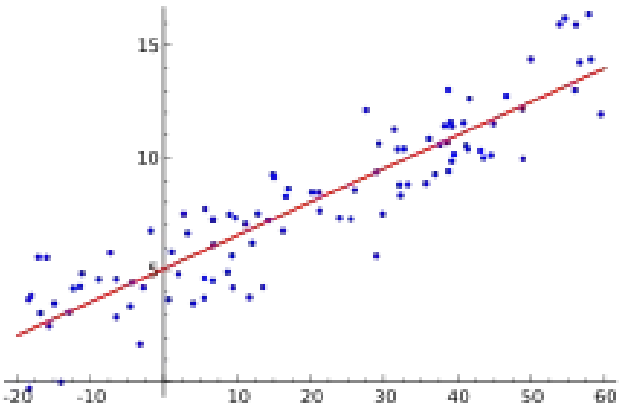
Reinforcement learning: Reinforcement learning is a type of machine learning that involves training an agent to take actions in an environment to maximize a reward signal. GPUs can be used to speed up the training process for reinforcement learning algorithms.

GPUs have become an essential tool for machine learning, providing the necessary computational power to perform complex tasks in a fraction of the time required by a CPU. With their ability to perform many parallel computations simultaneously, GPUs are well suited to perform the matrix operations that are common in machine learning. In addition, the use of GPUs has made it possible to train large neural networks and perform other complex machine learning tasks in real-time, making machine learning more accessible to researchers, engineers, and data scientists. If you're looking to get started with machine learning, or if you're already an experienced practitioner, incorporating a GPU into your workflow is an excellent way to increase your productivity and speed up your work.

The cost of a GPU is high for an entry level programmer, however, GPUs are available through Kaggle.com and Google Colab, there are other free GPUs available as well, just search for them.

Now that we have covered many of the programming needs to develop Machine Learning and AI algorithms we are ready to start diving into the theory behind Machine Learning.

Linearity



In statistics, linear regression is a linear approach to modelling the relationship between a scalar response (y) which is a single ordinary numerical value that is used to measure a quantity or variable, and one or more explanatory variables (x). The case of one explanatory variable is called simple linear regression. For more than one explanatory variable, the process is called multiple linear regression.

When you plot data points on a graph and then draw a line indicating their average on that graph you are making a linear approximation of the distribution on that graph of a data points.
The formulation for such a process is given:

$y = mx + b$

y is the temperature in Celsius—the value we're trying to predict. m is the slope of the line. x is the degrees of C—the value of our input feature. b is the y-intercept.

in machine learning, you'll write the equation for a model slightly differently:

$y' = b + w1x1$

y' is the predicted label (a desired output). b is the bias (the y-intercept), w is the weight of feature 1. Weight is the same concept as the "slope" in the traditional equation of a line. x is a feature (a known input).

A linear regression model with two predictor variables can be expressed with the following equation:

$Y = B0 + B1*X1 + B2*X2 + e.$

The variables in the model are:

Y, the response variable; X1, the first predictor variable; X2, the second predictor variable; and e, the residual error, which is an unmeasured variable. The parameters in the model are:

B0, the Y-intercept; B1, the first regression coefficient; and B2, the second regression coefficient. One example would be a model of the height of a shrub (Y) based on the amount of bacteria in the soil (X1) and whether the plant is located in partial or full sun (X2). Y is the dependent variable you are trying to predict, X1, X2 and so on are the independent variables you are using to predict it, b1, b2 and so on are the coefficients or multipliers that describe the size of the effect the independent variables are having on your dependent variable Y

Here is an example of how you can create a Pandas DataFrame from a CSV file and use scikit-learn to perform a basic linear regression on test data:

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Load the data into a Pandas DataFrame
df = pd.read_csv('data.csv')

# Split the data into features and target
features = df.drop('target_column', axis=1)
target = df['target_column']

# Split the data into training and testing sets, ‘train, test, split’
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2)

# Train the linear regression model
reg = LinearRegression().fit(X_train, y_train)

# Predict using the trained model
y_pred = reg.predict(X_test)

# Evaluate the model's performance
score = reg.score(X_test, y_test)
print('Test score: ', score)
```

In this example, we start by loading the data from a CSV file into a Pandas DataFrame using the pd.read_csv function. Then, we split the data into features and target variables. The features are the independent variables, while the target is the dependent variable that we want to predict. Next, we split the data into training and testing sets using the train_test_split function from scikit-learn's model selection module. This allows us to evaluate the performance of our model on a set of data it hasn't seen before. Once we have the training and testing sets, we train the linear regression model using the fit method from the LinearRegression class. Then, we use the trained model to make predictions on the test data using the predict method. Finally, we evaluate the model's performance using the score method, which returns the coefficient of determination (R^2). The R^2 value indicates the proportion of the variance in the target variable that is explained by the features. A value of 1.0 means that the model perfectly fits the data, while a value close to 0.0 means that the model does not fit the data well. See below for more information on measuring the model and it’s fit to make accurate predictions, also the concepts of over and under fitting are taken into consideration.

Features and Labels

In machine learning, a feature is a characteristic or attribute of an instance or observation that can be used as an input for a model. A feature can be thought of as a variable that describes an aspect of an instance or observation. For example, in a machine learning model to predict the price of a house, features might include the number of bedrooms, the square footage of the house, the neighborhood it is located in, etc. Features play a crucial role in building machine learning models as they provide the model with the information it needs to make predictions. A good set of features is often the key to building a high-performing model, while poor or irrelevant features can lead to under-performance.

The process of selecting and engineering features is often referred to as feature engineering, and it can be a time-consuming and iterative process. In many cases, the feature engineering process involves transforming or combining raw data into a more useful format for the model, as well as selecting a subset of the available features that are most relevant to the task at hand.

In machine learning, a label is a dependent variable or the target variable that you want to predict based on the input features. The label is the output that the machine learning model is trying to predict. In supervised learning, which is the most common type of machine learning, the model is trained on a labeled dataset that consists of instances (also known as samples or observations) with both features and corresponding labels. The model then learns to map the input features to the correct output label based on the relationship between the features and the labels in the training data.

The relationship between the features and the label can be thought of as the underlying function or decision boundary that the model is trying to learn. Once the model is trained, it can be applied to new, unseen instances with the same features to make predictions about the label. It is important to note that the label is not a feature, but it is related to the features in that the features are used as input to predict the label. The choice of features and the representation of the features can have a significant impact on the performance of the machine learning model.

In python code for machine learning, X is often used to represent the input features or the independent variables, while y is used to represent the target or dependent variable. For example, in a simple regression problem, X might represent the predictor variables, such as the size of a house, while y might represent the target variable, such as the price of the house. In a classification problem, X might represent the features of instances, such as the characteristics of a patient, while y might represent the class labels, such as whether a patient has a certain disease or not. In general, X is a 2D array-like object that holds the feature values for each instance in the dataset, while y is a 1D array-like object that holds the corresponding target values for each instance. When training a machine learning model, the goal is to fit a function that maps the input features X to the target values y. As seen in the ‘train, test, split’ example above where X and y are used as variable names.

Often you will see the equation f(x) which represents a mathematical function that maps input features x to the target or predicted value. The function f can be thought of as a model that is learned from the training data and used to make predictions for new, unseen instances. In the context of supervised learning, the goal is to find the best possible function f that fits the relationship between the input features x and the target values y in the training data. The quality of the function f is usually evaluated based on how well it predicts the target values for instances in the test data. For example, in a linear regression problem, the function f might be a linear combination of the input features, such as $f(x) = w_0 + w_1 * x_1 + w_2 * x_2 + ... + w_n$

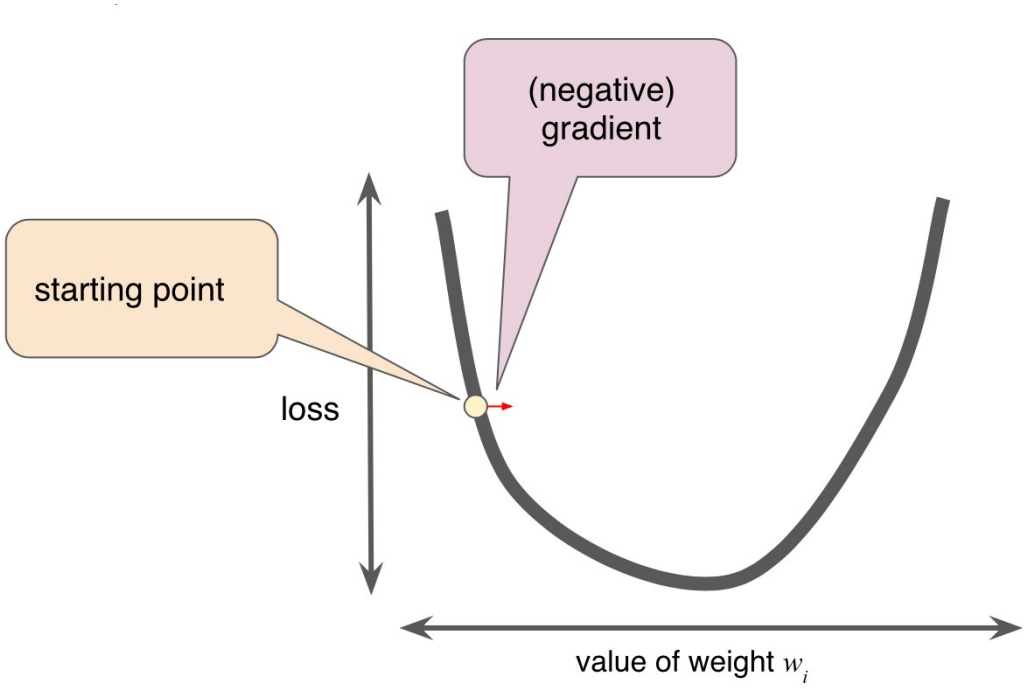
* x_n, where w are the coefficients that are learned from the training data. In a decision tree, the function f might be a tree structure that represents a series of decisions based on the input features.
In general, the function f represents the learned relationship between the input features and the target values, and it is used to make predictions for new, unseen instances.

Gradient Descent

A typical definition of gradient descent is:

> Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or approximate gradient) of the function at the current point. If, instead, one takes steps proportional to the positive of the gradient, one approaches a local maximum of that function; the procedure is then known as gradient ascent.

However, to a mechanic this is still obtuse.



The gist of gradient descent is that you want to find the lowest point in the line, the arc of descent/ascent, like the creek bed in the forest: water always finds the lowest point to run along, the local minimum. What you are seeking in dealing with gradient descent is to find a piece of gold in the creek you need to step

along the creek, if you step too far you will run right past it, if you go too slow, it will take forever. This stepping through the creek is what is known as the 'learning rate' or 'step size'. The sweet spot, or goldilocks zone, is the point where there is a low value to loss vs weights. You are trying to find that spot but not overshoot it. If the bottom of the arc is large, then your learning rate can be larger, if the gradient of the arc is steep then a small learning rate is needed. Goodfellow et al note on gradient based optimization, “Most deep learning algorithms involve optimization of some sort. Optimization refers to the task of either minimizing or maximizing some function f(x) by altering x. We usually phrase most optimization problems in terms of minimizing f(x). Maximization may be accomplished via a minimization algorithm by minimizing −f (x).” (Goodfellow et al)

The function we want to minimize or maximize is called the objective function, or criterion. When we are minimizing it, we may also call it the cost function, loss function, or error function.

Optimization algorithms that use only the gradient, such as gradient descent, are called first-order optimization algorithms. Optimization algorithms that also use the Hessian matrix, such as Newton’s method, are called second-order optimization algorithms” (Goodfellow et al, Ch 4.3).

Gradient descent is an optimization algorithm used to minimize a function, typically a cost function in machine learning. The algorithm works by iteratively updating the parameters of the model in the opposite direction of the gradient of the cost function with respect to those parameters. The mathematical equation for gradient descent is:

$$\theta = \theta - \alpha * \nabla\theta J(\theta)$$

where θ is the set of parameters being optimized, $J(\theta)$ is the cost function, and α is the learning rate, a small positive value that determines the step size at each iteration. The gradient, $\nabla\theta J(\theta)$, is a vector of partial derivatives of $J(\theta)$ with respect to each parameter in θ.

The code for $\theta = \theta - \alpha * \nabla\theta J(\theta)$ in Python would depend on the specific implementation of the variables and function. Here's an example of how this code could be written assuming that θ is a numpy array, α is a scalar value, and $\nabla\theta J(\theta)$ is a function that returns a numpy array:

```
import numpy as np

# Define the function to calculate the gradient of J(θ) with respect to θ (theta)
def grad_J(theta):
    # ... implementation of the gradient calculation ...
    return gradient

# Define the initial value of theta
theta = np.array([1.0, 2.0, 3.0])

# Define the learning rate alpha
alpha = 0.01
```

```
# Calculate the gradient of J(θ) with respect to θ
gradient = grad_J(theta)

# Update the value of theta
theta = theta - alpha * gradient
```

In this example, the grad_J function takes the current value of theta as input and returns the gradient of J(θ) with respect to theta. The theta variable is then updated using the learning rate alpha and the gradient of J(θ) with respect to theta, which is calculated using the grad_J function. The updated value of theta is stored back in the theta variable. Note that you would need to replace grad_J with the actual implementation of the gradient calculation for your specific problem.

The algorithm stops when a local or global minimum of the cost function is reached. The terms "local minimum" and "global minimum" refer to the minima of a cost function that is being optimized. The cost function is used to evaluate the performance of a machine learning model and its goal is to minimize this function. A local minimum is a minimum value of the cost function that is only the minimum within a certain region, or "neighborhood", of the parameter space. In other words, it's a minimum that is only the lowest compared to its nearby values. A global minimum is the minimum value of the cost function that is the minimum value throughout the entire parameter space. It's the absolute lowest value that the cost function can take. In optimization problems, it's often desirable to find the global minimum, as this corresponds to the best possible solution to the problem. However, finding the global minimum can be difficult, especially for complex and high-dimensional problems. In some cases, optimization algorithms may get stuck in a local minimum and fail to find the global minimum, leading to suboptimal solutions.

There are different variants of gradient descent, such as batch gradient descent, which uses the entire training set to compute the gradient at each iteration, and stochastic gradient descent, which uses a single example to compute the gradient at each iteration.

Linear gradient descent is a specific variant of the general gradient descent algorithm that is used to optimize linear models such as linear regression. The algorithm works by iteratively updating the parameters of the model, typically the coefficients of the linear equation, in the opposite direction of the gradient of the cost function with respect to those parameters. The cost function used in linear gradient descent is typically the mean squared error (MSE) between the predicted output and the true output. Linear gradient descent is a simple and efficient algorithm for linear models, but it can be sensitive to the choice of the learning rate and can get stuck in local minima. Alternative optimization techniques like batch gradient descent and stochastic gradient descent can also be applied to linear regression models.

Stochastic Gradient Descent (SGD) is a variant of the gradient descent algorithm that is used for optimization of large datasets. Unlike batch gradient descent, which computes the gradient based on the average of the gradients of all the data points, SGD computes the gradient based on a single data point at a time. This makes it computationally more efficient and can also help with escaping local minima. The mathematical equation for stochastic gradient descent is the same as that for gradient descent.

Here is an example of how the SGD algorithm can be implemented in Python for linear regression:

```
# Initialize parameters
theta = np.random.randn(2)

# Define the learning rate
alpha = 0.01

# Number of iterations
n_iter = 1000

# Loop through the number of iterations
for i in range(n_iter):
    # Select a random data point
    rand_index = np.random.randint(len(X))
    xi = X[rand_index:rand_index+1]
    yi = y[rand_index:rand_index+1]

    # Compute the gradient
    gradient = 2 * xi.T.dot(xi.dot(theta) - yi)

    # Update the parameters
    theta = theta - alpha * gradient
```

In this example, X is the feature matrix and y is the target vector, theta is the parameters vector, alpha is the learning rate and n_iter is the number of iterations.

It is worth noting that, in practice, the learning rate is often decreased over time to help the algorithm converge more efficiently, see more on the importance of learning rate below. Additionally, there are more advanced versions of the SGD algorithm like: Mini-batch Gradient Descent which uses a small batch of examples at each iteration instead of a single example. Adaptive Gradient Descent methods like Adagrad, Adadelta and Adam, which automatically adapt the learning rate on a per-parameter basis, these are known as learning rate or scheduler optimizers in PyTorch. You can use any of the above methods to improve the performance of your model.

Adagrad, Adadelta, and Adam are three popular optimization algorithms used in machine learning to adjust the learning rate and improve the efficiency of gradient descent.

Adagrad: Adagrad stands for "adaptive gradient" and is an optimization algorithm that adapts the learning rate for each parameter based on its historical gradient information. It gives more weight to parameters with sparse gradients and less weight to parameters with frequent gradients. Adagrad has proven to be effective in handling sparse data and is commonly used in natural language processing and recommendation systems.

Adadelta: Adadelta is an extension of Adagrad that seeks to address its aggressive and monotonically decreasing learning rate. Adadelta replaces the learning rate with a running average of the past gradients and past squared gradients, which allows it to adapt to changing gradients and keep the learning rate from becoming too small. Adadelta has been shown to be effective in training deep neural networks.

Adam: Adam stands for "adaptive moment estimation" and is a popular optimization algorithm that combines the benefits of both Adagrad and Adadelta. Adam uses a combination of the first and second moments of the gradients to adapt the learning rate for each parameter. It also introduces bias correction to reduce the effect of the initial gradient estimates. Adam has been shown to be effective in training deep neural networks and is commonly used in computer vision, natural language processing, and other areas of machine learning.

In general, these optimization algorithms are used to overcome the limitations of traditional gradient descent methods, which can suffer from slow convergence or get stuck in local minima. By adapting the learning rate and taking into account historical gradient information, Adagrad, Adadelta, and Adam can accelerate the learning process and improve the accuracy of the resulting models.

Convergence in gradient descent refers to the point at which the optimization algorithm has found an optimal set of parameter values such that the loss function is minimized or optimized to a satisfactory degree. In other words, it is the point where further iterations of the algorithm no longer lead to significant improvement in the loss function. Gradient descent is an iterative optimization algorithm that seeks to find the optimal set of parameter values by adjusting the values in the direction of the steepest descent of the loss function. At each iteration, the algorithm calculates the gradient of the loss function with respect to the parameters and updates the parameter values by taking a step in the opposite direction of the gradient. This process is repeated until convergence is reached.

There are several ways to determine convergence in gradient descent. One common approach is to monitor the change in the loss function or the parameter values between iterations. If the change falls below a certain threshold, the algorithm is considered to have converged. Another approach is to set a maximum number of iterations and terminate the algorithm when that limit is reached. It is important to note that reaching convergence does not necessarily mean that the solution found is the global minimum of the loss function. In fact, in many cases, gradient descent may only converge to a local minimum or a saddle point. To address this issue, various modifications to the gradient descent algorithm have been proposed, such as using different initialization values, optimizing with respect to a subset of the parameters, or using more sophisticated optimization algorithms like Adagrad, Adadelta, and Adam.

Loss Functions

Loss function (also known as a cost function or objective function) is a function that measures the difference between the predicted values of a model and the actual values of the training data. The goal of a machine learning model is to minimize the loss function in order to improve its predictive accuracy. There are many different types of loss functions, and the choice of which one to use depends on the specific problem being solved. Some of the most common loss functions include:

Mean Squared Error (MSE): This is the most commonly used loss function for regression problems. It measures the average squared difference between the predicted and actual values.

Binary Cross-Entropy: This loss function is used for binary classification problems, where the output is either 0 or 1. It measures the difference between the predicted and actual probabilities.

Categorical Cross-Entropy: This loss function is used for multi-class classification problems, where the output can take on more than two values. It measures the difference between the predicted and actual probability distributions.

Hinge Loss: This loss function is commonly used for support vector machines (SVMs) in binary classification problems. It penalizes misclassifications and encourages the model to separate the data points with a large margin.

KL Divergence: This loss function is used in generative models such as variational autoencoders (VAEs) to measure the difference between the predicted distribution and the actual distribution of the data.

Huber Loss: This loss function is a combination of mean squared error and mean absolute error. It is more robust to outliers than MSE and is often used in regression problems.

The choice of which loss function to use is an important consideration when designing a machine learning model. Different loss functions may lead to different optimal parameter values and affect the performance of the model. It is important to select a loss function that is appropriate for the problem being solved and to monitor its behavior during training in order to adjust the model accordingly.

Models

A model defines the relationship between features and label. Two phases of a model's life:

Training means creating or learning the model. That is, you show the model labeled examples and enable the model to gradually learn the relationships between features and label.

Inference means applying the trained model to unlabeled examples. That is, you use the trained model to make useful predictions (y'). For example, during inference, you can predict median HouseValue for new unlabeled examples.

A model in machine learning is a mathematical representation of a system or process that is used to make predictions or decisions. A model is trained using a set of input data and corresponding output data, and the goal of training is to find the set of parameters that best fit the data.

Hyperparameters are parameters that are set before the training of a model, unlike the parameters that are learned during the training process. They are used to control the behavior of the model and the learning algorithm. Examples of hyperparameters include the learning rate, the number of hidden layers in a neural network, and the number of trees in a random forest. Hyperparameters can have a significant impact on the performance of a model. Choosing appropriate hyperparameters can lead to a better-performing model, while choosing inappropriate ones can result in a model that underfits or overfits the data. Hyperparameter tuning is the process of finding the best set of hyperparameters for a given task and data. Hyperparameter

tuning is an important aspect of effective ML. You can use GridSearchCV to optimize boosted decision trees, and in PyTorch you can use a library like Optuna to optimize PyTorch training hyperparameters.

Here is an example of training a model with hyperparameters and optimizing the hyperparameters with a grid search algorithm in Python using the scikit-learn library:

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

# Define the hyperparameter grid
param_grid = {'n_estimators': [10, 50, 100], 'max_depth': [None, 5, 10]}

# Create the model
rf = RandomForestClassifier()

# Create the grid search object
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=5)

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Print the best hyperparameters
print("Best Hyperparameters: ",grid_search.best_params_)
```

In this example, we are using a random forest classifier and tuning the n_estimators and max_depth hyperparameters. The GridSearchCV object is used to perform a grid search over the defined hyperparameter grid and find the best set of hyperparameters. The fit method is used to fit the grid search to the training data, and the best_params_ attribute is used to print the best hyperparameters found.

Supervised vs. Unsupervised Learning

In supervised learning a label is the thing we're predicting—the y variable in simple linear regression. The label could be the future price of wheat, the kind of animal shown in a picture, the meaning of an audio clip, or just about anything.
One example of Unsupervised learning is clustering (K-Means). Cluster analysis- ascertain on the basis of X1...Xn, whether the observations fall into relatively distinct groups. In semi-supervised- some instances have labels others do not.

Variables (Features, Parameters, Coefficients)

A feature is an input variable—the x variable in simple linear regression. A simple machine learning project might use a single feature, while a more sophisticated machine learning project could use millions of features
either quantitative/regression (num) or qualitative/classification (text) Exceptions: least square linear regression is used with quantitative responses, logistic regression is used with qualitative (two-class or

binary response: male, female; it estimates class probabilities, it can be thought of as a regression method as well. Whether the predictors are qualitative or quantitative is considered less important

Hyperparameters

Hyperparameters are the knobs that programmers tweak in machine learning algorithms. Most machine learning programmers spend a fair amount of time tuning the learning rate. Iterations is another important hyperparameter in linear regression. It is important not to confuse hyperparameters with model features (also known as parameters). A very good explanation of hyperparameters is given by Xavier Amatriain :

In machine learning, we use the term hyperparameter to distinguish from standard model parameters. So, it is worth to first understand what those are. Wheras, variables are used in the logic of an algorithm, the hyperparameters are variables that make the algorithm itself run.

A machine learning model is the definition of a mathematical formula with a number of parameters that need to be learned from the data. That is the crux of machine learning: fitting a model to the data. This is done through a process known as model training. In other words, by training a model with existing data, we are able to fit the model parameters.

However, there is another kind of parameters that cannot be directly learned from the regular training process. These parameters express “higher-level” properties of the model such as its complexity or how fast it should learn. They are called hyperparameters. Hyperparameters are usually fixed before the actual training process begins.

So, how are hyperparameters decided? That is probably beyond the scope of this question, but suffice to say that, broadly speaking, this is done by setting different values for those hyperparameters, training different models, and deciding which ones work best by testing them.

So, to summarize. Hyperparameters: Define higher level concepts about the model such as complexity, or capacity to learn. Cannot be learned directly from the data in the standard model training process and need to be predefined. Can be decided by setting different values, training different models, and choosing the values that test better Some examples of hyperparameters:

- Number of leaves or depth of a tree
- Number of latent factors in a matrix factorization
- Learning rate (in many models), a very important hyperparameter that can make or break a model.
- Number of hidden layers in a deep neural network
- Number of clusters in a k-means clustering

Loss and Measuring Quality of Fit

Training a model simply means learning (determining) good values for all the weights and the bias from labeled examples. In supervised learning, a machine learning algorithm builds a model by examining many examples and attempting to find a model that minimizes loss; this process is called empirical risk minimization.

Loss is the penalty for a bad prediction. That is, loss is a number indicating how bad the model's prediction was on a single example. If the model's prediction is perfect, the loss is zero; otherwise, the loss is greater. The goal of training a model is to find a set of weights and biases that have low loss, on average, across all examples

In order to eval the performance of a statistical learning method on a given data set, we need a measure, we need to quantify the extent to which the predicted response value for a given observation is close to the true response value for that observation. There are different kinds of eval metrics: MEA- Mean Absolute Error, is the mean of the abs err. MSE- Mean Squared Error, mean of the squared errors RMSE- sq.rt. of the MSE. Also see regularization below.

The Bias-Variance Trade off

In order to min the expected error, we need to select a statistical learning method that simultaneously achieves low variance and low bias.

> variance- refers to the error that is introduced by approximating a real-life problem, by a much simpler model. For example, linear regression assumed that there is a linear relationship between Y and X1, X2,...Xn. It is unlikely any real-life problem truly has such a simple linear relationship.

As a general rule, as we use more flexible methods, the variance will increase and the bias decrease. The relative rate of change of these two quantities determines whether the test MSE increases or decreases. As we increase the flexibility of a class of methods, the bias tends to mutually decrease faster than the variance increases. Consequently, the expected test MSE declines. However at some point increasing flexibility has little impact on the bias but starts to significantly increase the variance. When this happens the MSE increases.

The relationship b/w bias, variance + MSE is known as bias-variance trade off.

Bayes' Theorem: describes the probability of an event, based on prior knowledge of conditions that might be related to the event. One of the many applications of Bayes' Theorem is Bayesian inference, a particular approach to statistical inference. When applied, the probs involved in Bayes' theorem may have different probabilistic interpretations. With the Bayesian probability interpretation, the theorem expresses how a subjective degree of belief should rationally change to account for availability of related evidence.

Bias-Unbiased

The term "bias" refers to the error introduced by approximating a real-world problem with a simplified

model. The bias of a model is the difference between the expected value of the predictions and the true values of the data. A model with high bias tends to be too simplistic and may underfit the data, while a model with low bias may overfit the data. On the other hand, the term "unbiased" in machine learning typically refers to an estimator that has an expected value that is equal to the true value of the parameter being estimated. In other words, an unbiased estimator does not systematically overestimate or underestimate the parameter.

The relationship between these two concepts and the parameter μ depends on the specific problem being addressed. In some cases, the parameter μ may represent the true value of a population parameter, such as the mean or variance of a distribution. In other cases, μ may represent the optimal parameter values for a machine learning model. If the goal is to estimate the parameter μ using a machine learning model, the bias of the model will depend on how well the model captures the underlying structure of the data. A model with high bias will tend to underestimate or overestimate μ, leading to a biased estimator. An unbiased estimator, on the other hand, will have an expected value that is equal to μ, regardless of the complexity of the model. If we use the mean (avg) of the sample $\hat{\mu}$ to estimate μ this estimate is unbiased, in the sense that on average, we expect $\hat{\mu} = \mu$, meaning on one set of observations $y1,...y_n$ $\hat{\mu}$ might overestimate, with another set of observations it might underestimate, but with an average of a large volume of observations it is more accurate.

It is important to realize that one can get stuck in optimization problems in what are known as local and global maxima and minima. Machine learning algorithms such as gradient descent algorithms may get stuck in local minima during the training of the models. Gradient descent is able to find local minima most of the time and not global minima because the gradient does not point in the direction of the steepest descent. Current techniques to find global minima either require extremely high iteration counts or a large number of random restarts for good performance. Global optimization problems can also be quite difficult when high loss barriers exist between local minima.

Any greedy algorithm -- which try to maximize your objective by choosing whichever is locally best -- may get stuck in local minima.

These include, but are not limited to:

- gradient descent optimization like neural networks, which try to gradually minimize the loss function by making small changes to the weights in the direction that minimize the loss. This also includes algorithms like gradient boosting. You can mitigate such problems by using second-order derivatives (Hessian) or smart heuristics.

- one look-ahead algorithms like decision trees, which locally choose the feature x and threshold t whose split x<t maximizes/minimizes a given metric like gini or entropy (and, as a subsequent, random forest) -- if a decision tree was able to see the final impact of all possible combinations, then it might make different choices, but that would be infeasible.

- expectation-maximization (as used by e.g. k-means) is also heavily influenced by the initialization.

Predictions

3 Kinds of uncertainty associated with prediction:

a. coefficient estimates are the least square plane which is only an estimate for the true population regression plane. The inaccuracy in the coefficient estimates is related to the reducible error. We can compute a confidence interval in order to determine how close Y will be to f(x)

b. in practice assuming a linear model for f(x) is almost always an approximation of reality, model bias- when we use a linear model, we are in fact estimating the best linear approximation to the true surface, ignoring the real topology for linear approximation.

c. even if we knew f(x) the response value cannot be predicted perfectly because of the random error (ε), the irreducible error.

Prep and Clean the Data

A key step before running a linear regression on data is known as EDA. Exploratory Data Analysis (EDA) is an essential step in the machine learning process that provides valuable insights into the underlying patterns and relationships within a given dataset. It involves a thorough examination of the data to understand its characteristics and identify any potential issues or biases that may impact the modeling process. The goal of EDA is to gain a deeper understanding of the data and to inform the selection of appropriate models and algorithms for a given problem. One of the key advantages of EDA is that it helps to identify any missing or incorrect data, which can be crucial in avoiding incorrect or suboptimal results. This is particularly important in the case of large and complex datasets, where manual checks may not be feasible. By identifying and correcting any issues in the data, EDA ensures that the modeling process is based on accurate and reliable data, leading to more accurate predictions and improved model performance.

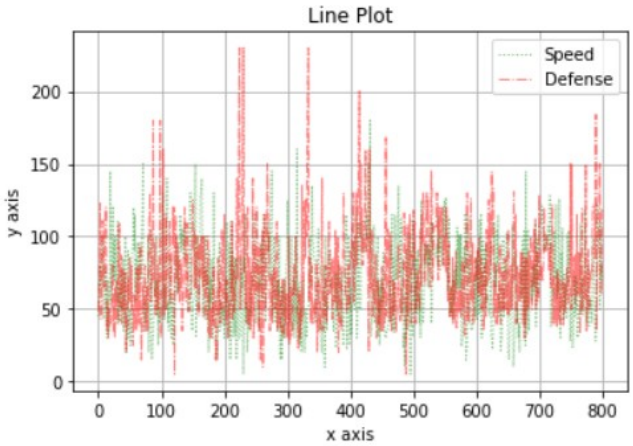
EDA also helps to identify patterns and relationships within the data, which can inform the selection of appropriate algorithms and models. For example, it may reveal correlations between variables that can be leveraged to improve the accuracy of predictive models. In addition, EDA can also provide insights into the distribution of the data, allowing for appropriate pre-processing and normalization steps to be taken. Another important aspect of EDA is that it helps to gain a deeper understanding of the problem being solved. This can be particularly useful in real-world applications where domain knowledge and expert insights are essential for effective problem-solving. By exploring the data and identifying patterns and relationships, EDA can provide valuable insights into the problem and help to guide the modeling process.

Visualizations are a good way to see the big picture of your data and understand the relationships in the data and identify any patterns or correlations. This information can be used to inform and guide the modeling process, leading to more accurate and effective results.

Matplotlib is a widely-used data visualization library in the Python programming language that provides a wide range of visualization capabilities. With its rich set of plotting functions and customization options, Matplotlib enables the creation of a wide range of visual representations, including line plots, scatter plots, bar charts, histograms, and more.

One of the key benefits of using Matplotlib for data visualization is its versatility. Matplotlib can be used to create simple plots and charts, as well as more complex visualizations, making it an ideal tool for exploring and understanding data in a wide range of applications. For example, scatter plots can be used to identify correlations between variables, while histograms can provide insights into the distribution of the data.

In addition to its versatility, Matplotlib also provides a high level of customization, enabling users to create visualizations that are tailored to their specific needs. This can include customizing the appearance of the plot, such as the axis labels, title, and color palette, as well as adding annotations and highlighting specific data points. Another important aspect of using Matplotlib for data visualization is its integration with other libraries and tools. For example, Matplotlib can be used in conjunction with other libraries such as NumPy and Pandas to create complex visualizations that leverage the full capabilities of these libraries. Additionally, Matplotlib provides the ability to export visualizations in a wide range of formats, including PNG, PDF, and SVG, making it easy to share visualizations and communicate insights.



Plot using matplotlib from https://www.kaggle.com/code/kanncaa1/data-sciencetutorial-for-beginners

The above plot can be coded as such:

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt

data.Speed.plot(kind = 'line', color = 'g',label = 'Speed',linewidth=1,alpha = 0.5,grid =
True,linestyle = ':')
data.Defense.plot(color = 'r',label = 'Defense',linewidth=1, alpha = 0.5,grid =
True,linestyle = '-.')
plt.legend(loc='upper right')     # legend = puts label into plot
plt.xlabel('x axis')              # label = name of label
plt.ylabel('y axis')
plt.title('Line Plot')            # title = title of plot
plt.show()
```

Here are some common steps in exploratory data analysis (EDA) for machine learning:

1. Understand the problem and the data: Before beginning the EDA process, it is important to have a clear understanding of the problem you are trying to solve and the data you are working with. This includes understanding the features, the target variable, and any domain-specific considerations.

2. Import the data: The first step in EDA is to import the data into your environment. This can be done using Python libraries like Pandas, NumPy, or Scikit-learn.

3. Clean the data: Once the data is imported, it is important to clean it to remove any missing or irrelevant data, and correct any errors or inconsistencies.

4. Visualize the data: Data visualization is a powerful tool for understanding the distribution of the data, identifying patterns, and detecting outliers or anomalies. Common visualization techniques include histograms, scatterplots, boxplots, and heatmaps.

5. Check for correlations: Correlations between features can help to identify relationships between the variables and guide feature selection. Pearson correlation coefficient, Spearman rank correlation, and Kendall rank correlation are some of the commonly used correlation measures.

6. Feature engineering: Feature engineering is the process of creating new features from existing ones or selecting a subset of features for the model. It is important to select relevant features and remove redundant or noisy ones.

7. Check for imbalanced data: In some cases, the target variable may be imbalanced, with one class having many more instances than the other. This can lead to biased models and reduced predictive power. It is important to identify and address any imbalances in the data.

8. Check for outliers: Outliers are extreme values that can have a disproportionate impact on the model. It is important to identify and handle outliers appropriately to ensure that they do not adversely affect the model's performance.

9. Summarize the findings: Finally, it is important to summarize the findings of the EDA process in a clear and concise manner, including any insights gained, any issues identified, and any recommendations for future work.

After you have performed this initial step you are then ready to train a linear regression model using a regression algorithm.

A Recipe for a Linear Regression Algorithm

1. Train test split: The first step is to split the data into a training set and a test set. The training set is used to train the model, while the test set is used to evaluate the model's performance on unseen

data. The scikit-learn library in Python provides a convenient way to split the data into training and test sets.

2. Create and train the model: The next step is to create a linear regression model and train it on the training data. In Python, this can be done using the scikit-learn library.

3. Fit the data: Once the model is created, it needs to be fit to the training data. This involves finding the optimal values of the model parameters that minimize the difference between the predicted and actual values.

4. Evaluate the model: After fitting the model to the training data, the next step is to evaluate its performance on the test data. This can be done by checking the coefficients of the model and interpreting them in the context of the problem being solved.

5. Check coefficients: The coefficients of the model represent the strength and direction of the relationship between each feature in the dataset and the target variable. By examining these coefficients, we can gain insights into the factors that are driving the model's predictions.

6. Make predictions: Once the model is trained and evaluated, we can use it to make predictions on new data. This involves applying the model to the feature values of the new data and predicting the target variable.

7. Check metrics: Finally, we need to check the metrics of the model, such as accuracy, confusion matrixes, precision, etc., to evaluate its performance on the test data. This will give us a sense of how well the model is performing and whether it needs to be improved or refined.

An example of a simple linear regression algorithm in Python using the scikit-learn library:

```
# Import libraries
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Load the data
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([2, 4, 6, 8, 10])

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Create a linear regression model
model = LinearRegression()

# Fit the model to the training data
model.fit(X_train, y_train)
```

```
# Evaluate the model on the test data
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)

# Print the coefficients and metrics
print("Coefficients:", model.coef_)
print("Intercept:", model.intercept_)
print("Mean squared error:", mse)
```

In this example, we first load a small dataset with five points, where the input feature X is the integers 1 through 5 and the target variable y is the corresponding even integers. We then split the data into training and test sets using the train_test_split function from scikit-learn. Next, we create a LinearRegression model and fit it to the training data using the fit method. We then use the model to make predictions on the test data using the predict method, and evaluate the performance of the model using the mean squared error metric. Finally, we print the coefficients of the model and the mean squared error.

Potential Problems of Linear Regression

There are several tricky areas related to linear regression, some of these areas are covered below. It is important to understand that linear regression is a mathematical methodology of dealing with quantifiable objects, a numerical relationship to other numerical relationships.

1. Non-linearity of the r-p relationships

The term "non-linearity of the r-p relationships" usually refers to the fact that the relationship between input features (also called predictors or independent variables) and the target variable (also called the response or dependent variable) may not be linear, but rather nonlinear. The "r-p" refers to the correlation (r) and regression (p) relationships between the input features and the target variable.

Linear models assume that the relationship between the input features and the target variable is linear, which means that the target variable can be modeled as a linear combination of the input features, with a fixed slope and intercept. However, in many real-world scenarios, the relationship between the input features and the target variable is more complex and nonlinear. For example, in image recognition, the relationship between the pixel values of an image and the object in the image is highly nonlinear.

Nonlinear models can capture these complex relationships by using nonlinear functions of the input features, such as polynomial or exponential functions, or by using more complex models like decision trees or neural networks. These models can capture the nonlinear relationships between the input features and the target variable, and may provide better predictive power than linear models.

If the true relationship is far from linear, then virtually all of the conclusions that we draw from the fit are questionable, thus the prediction accuracy can be reduced.

Residual plots are a tool to detect non-linearity if the residual plots indicate there are non-linear associations in the data, then a simple approach is to use a non-linear transformation of the predictors; logX, sq. rt of X and X^2 in the regression model.

2. Correlation of Error Terms

important assumption in linear regression is that error terms ε1,ε2...,εn are uncorrelated. If there is correlation then the estimated standard error with underestimate the true standard error. Often occurs in time series data which is detectable through tracking (sim values for i) in adjacent residuals.

3. Non-Constant Variance of Error Terms

Another important assumption of the linear regression model is that the error terms have a constant variance, The standard errors, confidence intervals, and hypothesis tests associated with the linear model rely upon this assumption. But often they are non-constant. the assumption of constant variance of error terms is an important assumption in many models, especially linear regression models. The error term is the difference between the predicted values of the target variable and the actual values. If the error term has a constant variance, it means that the spread of the errors is roughly the same across all values of the predictor variables.

The constant variance assumption is also known as homoscedasticity. Homoscedasticity is important because it ensures that the model is consistent across the entire range of predictor variables, and that the model is not overly sensitive to outliers or extreme values.

In contrast, when the variance of the error term changes across different levels of predictor variables, it is known as heteroscedasticity. Heteroscedasticity can lead to biased and inconsistent estimates of the model parameters, and can also affect the accuracy and reliability of the predictions.

To check for homoscedasticity, analysts can plot the residuals (the difference between the actual and predicted values) against the predicted values. A scatter plot of the residuals should show no obvious pattern, and the spread of the residuals should be roughly constant across all levels of the predictor variables.

If heteroscedasticity is detected, analysts may need to use techniques such as weighted least squares or generalized least squares to correct for it. Alternatively, they may need to transform the data to achieve a more constant variance, or use different models that are more robust to heteroscedasticity.

4.Outliers

An outlier is a point for which y, is far from the val predicted by the model residual plots identify outliers we can plot the studentized residuals- computed by dividing each residual e, by its estimated standard error. Possible outliers have a value of >3, if outliers are data recording error, remove it. an outlier is an observation that differs significantly from other observations in the dataset. Outliers are data points that are unusually far from the majority of the data points, either in terms of their values or their behavior.

Outliers can occur for many reasons, including measurement errors, data entry errors, or legitimate but extreme values. Outliers can have a significant impact on the results of machine learning models, particularly on models that are sensitive to the presence of extreme values. Outliers can be detected using various statistical methods, such as the z-score or the interquartile range (IQR). The z-score is a measure of

how many standard deviations an observation is away from the mean of the data, while the IQR is a measure of the spread of the data that is less sensitive to extreme values than the standard deviation.

Once outliers are identified, there are several options for how to handle them in machine learning. One approach is to remove the outliers from the dataset entirely, which can improve the performance of some models. Another approach is to treat the outliers as a separate category, which can be useful in some cases where the outliers represent a distinct group of observations. Alternatively, some models can be designed to be more robust to the presence of outliers, for example by using techniques like robust regression or decision trees.

5. High Leverage Points

observations with high leverage have an unusual value for predictor x. They have high impact on the estimated regression line. Leverage statistics is used to compute an observations leverage, "leverage" refers to the extent to which an individual observation affects the model's estimates of the parameters. An observation with high leverage has a value for a predictor variable that is unusual compared to the other observations in the dataset. High leverage observations can have a significant impact on the results of a model, particularly if they are also outliers.

The presence of high leverage points can affect the accuracy and reliability of a model, particularly linear regression models. In linear regression, the influence of an observation on the model's estimates of the parameters is proportional to the distance of the observation's predictor value from the mean predictor value. Therefore, high leverage points can have a large influence on the model's estimates, particularly if they are also outliers.

To identify high leverage points, analysts can examine the studentized residuals, which are residuals that have been adjusted for their leverage. A plot of studentized residuals against the leverages can help identify observations with high leverage. If high leverage points are identified, there are several options for how to handle them in machine learning. One approach is to remove the high leverage points from the dataset entirely, which can improve the performance of some models. Another approach is to transform the predictor variables to reduce the influence of high leverage points, or to use robust regression techniques that are less sensitive to the presence of high leverage points.

6. Collinearity

Collinearity (also known as multicollinearity) refers to the situation where two or more predictor variables in a model are highly correlated with each other. When two or more predictors are highly correlated, it becomes difficult for the model to determine the individual effects of each predictor variable on the outcome variable, because the effects of each predictor are confounded with the effects of the other predictors. Collinearity can lead to unreliable and unstable estimates of the regression coefficients in a linear regression model, because small changes in the data can lead to large changes in the estimates. In some cases, the collinearity may be so severe that the model cannot be estimated at all.

Collinearity can be detected using several techniques, such as correlation matrices, variance inflation factors (VIFs), and eigenvalues. If collinearity is detected, one option is to remove one or more of the

highly correlated predictors from the model. Alternatively, techniques such as ridge regression or principal component regression can be used to reduce the impact of collinearity on the model's estimates. It is important to note, however, that removing variables or using complex techniques to account for collinearity should be done with care, as they can sometimes result in loss of important information or overfitting the model.

2 or more predictor variables are closely related to each other. The presence of collinearity can pose problems in the regression context, since it can be difficult to separate out the individual effects of collinear variables on the response. Collinearity reduces the accuracy of the estimates of the regression coefficients it causes the standard error for β to increase. The hypothesis test- the probability of correctly detecting a non-zero coefficient- is reduced. Detection: look at correlation matrix of the predictors- a large value means collinearity for multicollinearity assess the variance inflation factor (VIF). As a rule a VIF val > 5 = bad.

Overfitting and Underfitting

Overfitting and underfitting are two common problems that can occur in machine learning. They occur when a model is trained on a limited dataset and then tested on new data, resulting in poor performance. Overfitting occurs when a model is trained too well on the training data, and as a result, it performs poorly on new data. This happens when the model is too complex and is able to fit the noise in the training data. The model ends up memorizing the training data and is not able to generalize to new examples. This can be identified by a high training accuracy and a low test accuracy.

Underfitting occurs when a model is not trained well enough on the training data and as a result, it performs poorly on new data. This happens when the model is too simple and is not able to capture the underlying pattern in the data. This can be identified by a low training accuracy and a low test accuracy.

To avoid overfitting and underfitting, several techniques can be used such as cross-validation, regularization, and early stopping.

Cross-validation is a technique used to evaluate a model's performance by dividing the data into training and test sets. The model is trained on the training set and its performance is evaluated on the test set. This helps to identify if the model is overfitting or underfitting.

Regularization is a technique used to prevent overfitting by adding a penalty term to the model's loss function. This term discourages the model from assigning too much importance to any one feature.

Early stopping is a technique used to prevent overfitting by stopping the training process when the model's performance on the validation set stops improving.

L2 regularization, also known as weight decay, is a technique used to prevent overfitting in machine learning models. It works by adding a penalty term to the model's loss function, which discourages the model from assigning too much importance to any one feature. The regularization term is added to the loss function in the form of the sum of the squares of the model's weights (also known as the L2 norm of the weights). The regularization term is multiplied by a scalar value called the regularization strength or lambda, which controls the amount of regularization applied.

The L2 regularization term has the effect of shrinking the model's weights towards zero, which can help to reduce the variance in the model's predictions. This is because small weights are less likely to fit the noise in the training data, resulting in a more generalizable model.

In practice, L2 regularization is often used in conjunction with other techniques such as cross-validation and early stopping to prevent overfitting. It is also commonly used in neural network models, such as in

weight decay for training of deep learning models. One important thing to note is that L2 regularization can also have a computational cost, because the added term in the loss function increases the size of the computation graph and the number of parameters to be updated during the optimization process.

A confusion matrix is a table that is often used to evaluate the performance of a machine learning model, particularly in the context of classification problems. It is used to compare the predicted class labels of a model with the true class labels of the data. The confusion matrix shows the number of correct and incorrect predictions for each class, allowing for a more detailed analysis of the model's performance. A confusion matrix is typically represented as a table with rows representing the true class labels and columns representing the predicted class labels. The cells in the table contain the count of observations for each combination of true and predicted class labels.

For example, consider a binary classification problem where the true class labels are "positive" and "negative" and the predicted class labels are also "positive" and "negative". A confusion matrix for this problem would have four cells, with the top left cell representing the number of true positive predictions, the top right cell representing the number of false positive predictions, the bottom left cell representing the number of false negative predictions, and the bottom right cell representing the number of true negative predictions.

The information contained in the confusion matrix can be used to calculate various performance metrics, such as accuracy, precision, recall, and F1-score.

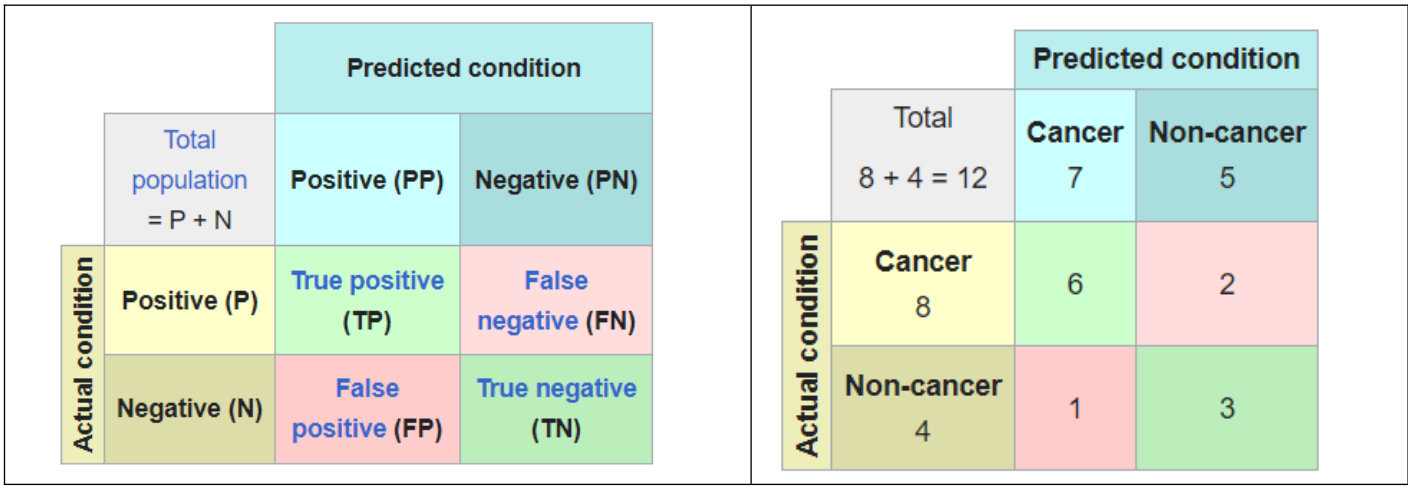
Accuracy is the proportion of correct predictions made by the model. It is calculated as the number of correct predictions (true positives and true negatives) divided by the total number of predictions.

Precision is the proportion of true positive predictions among all positive predictions made by the model. It is calculated as the number of true positives divided by the sum of true positives and false positives.

Recall, also known as sensitivity or true positive rate, is the proportion of true positive predictions among all actual positive observations. It is calculated as the number of true positives divided by the sum of true positives and false negatives.

F1-score is the harmonic mean of precision and recall. It is a measure of the balance between precision and recall, with a high F1-score indicating a good balance.

In addition to these performance metrics, the confusion matrix also allows for an analysis of the model's behavior across different classes. For example, if a model is frequently misclassifying a certain class, this information can be used to improve the model's performance on that class.



A Confusion Matrix

A confusion matrix is a useful tool for evaluating the performance of a machine learning model, particularly in the context of classification problems. It shows the number of correct and incorrect predictions for each class, allowing for a more detailed analysis of the model's performance. The information contained in the confusion matrix can be used to calculate various performance metrics, such as accuracy, precision, recall, and F1-score. It also allows for an analysis of the model's behavior across different classes and can help identify areas for improvement.

A confusion matrix can be generated using the confusion_matrix function from the sklearn.metrics library. Here is an example of how to use this function to generate a confusion matrix for a binary classification problem:

```
import numpy as np
from sklearn.metrics import confusion_matrix

# True class labels
y_true = np.array([0, 0, 1, 1, 1, 1])

# Predicted class labels (generated by a model)
y_pred = np.array([0, 1, 0, 1, 1, 1])

# Generate confusion matrix
cm = confusion_matrix(y_true, y_pred)

print(cm)
This will output the confusion matrix in the form of a 2x2 numpy array:

Copy code
[[1 1]
 [1 4]]
'''
The above matrix can be read as:

1 true negatives
1 false positives
1 false negatives
```

```
4 true positives
It's also possible to use PyTorch's torch.tensor instead of numpy array
'''
```

Once you have the confusion matrix, you can use it to calculate the performance metrics such as accuracy, precision, recall and F1-score.

In PyTorch, the accuracy of a model can be calculated using the torch.sum() and torch.mean() functions. Here is an example of how to calculate the accuracy of a binary classification model:

```
import torch

# True class labels
y_true = torch.tensor([0, 0, 1, 1, 1, 1], dtype=torch.long)

# Predicted class labels (generated by a model)
y_pred = torch.tensor([0, 1, 0, 1, 1, 1], dtype=torch.long)

# Calculate accuracy
accuracy = torch.mean((y_true == y_pred).float())

print(accuracy)
```

In the above code, the torch.tensor() function is used to create the true and predicted class labels. The == operator is used to compare the true labels with the predicted labels and create a tensor of the same shape with True for correct predictions and False for incorrect predictions. The .float() method is used to convert the tensor to float and the torch.mean() function is used to calculate the mean of the tensor which is equivalent to the accuracy.

It's also possible to use torch.sum() function to calculate the number of correct predictions and divide it by the total number of predictions to get the accuracy:

```
# Calculate accuracy
accuracy = torch.sum(y_true == y_pred).float() / y_true.size(0)

print(accuracy)
```

It's important to note that accuracy is not always the best metric to evaluate a model's performance, particularly in cases where the classes are imbalanced or when the model's performance is poor. Other metrics such as precision, recall, and F1-score should also be considered to get a more complete picture of the model's performance.

Bias in machine learning refers to the difference between the expected predictions of a model and the true values in the data. A model with high bias is one that makes strong assumptions about the data and as a result, is not able to capture the complexity of the underlying patterns in the data. A model with high bias will have a high training error and will also perform poorly on unseen data. This is because the model is not able to generalize well from the training data to new examples. High bias models are also known as underfitting models.

In a supervised learning problem, bias can be thought of as the difference between the average prediction of our model and the true values of the output we are trying to predict. A high bias model will have a large

difference between the average prediction and the true values, while a low bias model will have a smaller difference.

Bias is often caused by using a model that is too simple for the data or by not having enough data to properly train the model. To reduce bias, one can use more complex models or increase the amount of training data available. For example, using a deep learning model such as a neural network that can learn a more complex representation of the data can help to reduce bias.

3 Types of Machine Learning

There are three main types of machine learning: supervised learning, unsupervised learning, and reinforcement learning. Supervised learning is the most common type of machine learning, where the system is provided with labeled data (i.e. input-output pairs) and the goal is to learn a mapping from inputs to outputs. Examples of supervised learning tasks include image classification and linear regression. Unsupervised learning, on the other hand, deals with unlabeled data and the goal is to uncover hidden patterns or structures in the data. Clustering and dimensionality reduction are examples of unsupervised learning, see KNN below. Reinforcement learning is a type of machine learning where an agent learns to make decisions by interacting with its environment and receiving feedback in the form of rewards or penalties, as such A* search is an early example of using costs and rewards to give better ‘learned’ results from processing data inputs. A* was originally developed at Stanford Research Institute for the purpose of creating paths for robots to follow toward goals.

Machine learning algorithms are used in a wide range of applications, including natural language processing, computer vision, speech recognition, and robotics. Some popular machine learning algorithms include k-nearest neighbors, decision trees (boosted trees), and neural networks.

The k-nearest neighbor (KNN) algorithm is a simple and commonly used machine learning algorithm for both classification and regression tasks. It is a type of instance-based learning or non-parametric method, meaning that the algorithm doesn't make any assumptions about the underlying data distribution.

The basic idea behind the KNN algorithm is to find the k-number of training examples that are closest to the new data point, and then use these "neighbors" to make a prediction or a classification. The k number is a user-defined parameter that represents the number of nearest neighbors to take into account. For classification tasks, KNN makes a prediction for a new data point by majority voting among the k nearest neighbors. It assigns the class label that is most common among the k nearest training examples. For regression tasks, the KNN algorithm makes a prediction for a new data point by averaging the values of the k nearest neighbors.

A decision tree is a type of algorithm used in machine learning for both classification and regression tasks. The tree is constructed by recursively splitting the data into subsets based on the values of the input features. Each internal node of the tree represents a feature or attribute, and each leaf node represents a class label or a predicted value. The goal of the decision tree is to create a model that accurately predicts the class label or value of a new data point by traversing the tree from the root to a leaf node. Decision trees are simple to understand and interpret and can handle both categorical and numerical data. However, they can be prone to overfitting, which can be addressed through techniques such as pruning or ensembling. A popular decision tree used in ML is that of boosted trees, which you can code using XGBoost for example, more on this below.

Neural Network Overview

A neural network is a type of machine learning model that is inspired by the structure and function of the human brain. It consists of layers of interconnected "neurons," which process and transmit information. The inputs to a neural network are passed through these layers and transformed by the neurons into outputs. Neural networks can be used for a wide range of tasks, such as image recognition, natural language processing, and decision making. They can be trained using large amounts of data, and they are able to learn and improve over time. They have been widely used in industry, finance and other areas. PyTorch is a popular open-source machine learning library that provides a convenient interface for building and training neural networks. It allows for easy creation of complex network architectures and provides a variety of pre-built modules and functions for building neural networks. PyTorch also includes support for automatic differentiation, which allows for easy computation of gradients during training. One of the key features of PyTorch is its dynamic computation graph, which allows for greater flexibility in building and modifying neural network models. This is in contrast to other libraries such as TensorFlow, which use a static computation graph. PyTorch also provides support for a wide range of neural network layers, including fully connected layers, convolutional layers, and recurrent layers. These can be easily combined to build complex network architectures for tasks such as image classification, natural language processing, and time-series prediction. Additionally, PyTorch has a large community of developers who have created a wide range of pre-trained models and libraries that can be easily used for a variety of tasks, making it easier for developers to get started with neural network development.

Here is a pseudocode example of a basic neural network implemented in Python using the PyTorch library:

```
import torch
import torch.nn as nn

class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.input_layer = nn.Linear(in_features=28*28, out_features=256)
        self.hidden_layer = nn.Linear(in_features=256, out_features=128)
        self.output_layer = nn.Linear(in_features=128, out_features=10)

    def forward(self, x):
        x = x.view(-1, 28*28) # reshape input
        x = torch.relu(self.input_layer(x)) # apply activation function
        x = torch.relu(self.hidden_layer(x))
        x = self.output_layer(x)
        return x

model = NeuralNetwork()
```

This code creates a neural network class NeuralNetwork that inherits from nn.Module. The class has three layers, an input layer, a hidden layer, and an output layer. Each layer is defined as an instance of the nn.Linear class, which creates a fully connected layer. The input layer has 784 neurons (28*28 pixels) and 256 output neurons, the hidden layer has 128 neurons and the output layer has 10 neurons (10 different classes). The forward method takes the input x and applies the linear layers with relu activation function.

This method is called when the input is passed through the model to get the output. Finally, an instance of the NeuralNetwork class is created and assigned to the variable model. This instance can then be used for training and making predictions.

Neural Nets in NLP

Natural Language Processing (NLP) is a field of Artificial Intelligence (AI) that focuses on the interaction between computers and human languages. NLP enables computers to understand, interpret, and generate human language. It is a subfield of AI and draws on multiple disciplines including computer science, linguistics, and cognitive science. NLP has a wide range of applications, such as language translation, speech recognition, sentiment analysis, chatbots, and text summarization. One of the most popular NLP tasks is language translation, which involves converting text from one natural language to another. Another popular task is speech recognition, which involves converting speech to text. This technology is used in virtual assistants like Siri and Alexa, and also in voice-controlled devices like smart home assistants.

Sentiment analysis is another popular NLP task, which involves determining the sentiment or emotion in a given piece of text. This is useful in various fields such as marketing and customer service, where it is important to understand how people feel about a product or service.

Chatbots are computer programs that can conduct a conversation with humans using natural language. They are widely used in customer service, e-commerce, and other industries. Chatbots are used in video games by non-player characters (NPCs) which populate a game world to drive the game forward, provide narratives and fashion game play in a certain direction, steering (cybernetics) the game play of the human player. Text summarization is the task of automatically creating a shorter version of a piece of text, while still retaining the most important information. This can be used for a variety of applications such as summarizing news articles, scientific papers, and even long emails.

NLP models, such as DeBerta can be refined and customized to be applied to specific games, such as the fine-tuning of a pre-existing model to a specific game genre, such as Dungeons and Dragons, etc. DeBERTa (Decoding-Enhanced BERT) is a pre-trained natural language processing model that is based on the BERT architecture. It is trained on a large corpus of text data and fine-tuned on specific tasks such as question answering, named entity recognition, and sentiment analysis—which is used in player and NPC interactions, say something aggressive in a game and you will see the NPC mirror back the aggressive sentiment. DeBERTa has several improvements over the original BERT architecture, such as an additional layer of self-attention and a new training objective that focuses on token-level predictions. These changes lead to an improvement in performance on various NLP tasks. DeBERTa-v3 is the latest version of the DeBERTa model and has been trained on a much larger corpus of text data and has been fine-tuned on more tasks, leading to even better performance compared to previous versions.

Here is an example pseudocode of training a model using DeBERTa-v3 in Python using PyTorch:

```
import torch
from transformers import DeBERTaModel, DeBERTaTokenizer
```

```
# Load the DeBERTa-v3 model and tokenizer
model = DeBERTaModel.from_pretrained("deberta-base-v3")
tokenizer = DeBERTaTokenizer.from_pretrained("deberta-base-v3")

# Prepare input data
text = "The cat sat on the mat."
input_ids = tokenizer.encode(text, return_tensors='pt')

# Forward pass
outputs = model(input_ids)

# Fine-tune the model on a specific task
# Let's say we are fine-tuning the model on a named entity recognition task
from transformers import Trainer, TrainingArguments

# Define the training arguments
training_args = TrainingArguments(
    output_dir='./results',
    overwrite_output_dir=True,
    num_train_epochs=3,
    per_device_train_batch_size=16,
    save_steps=10_000,
    save_total_limit=2
)

# Create the trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset
)

# Start the fine-tuning
trainer.train()
```

In this example, we first load the DeBERTa-v3 model and tokenizer using the from_pretrained method. Then, we prepare the input data by encoding the text using the tokenizer. Next, we perform a forward pass through the model and get the outputs. Finally, we fine-tune the model on a specific task, in this case named entity recognition, using the Trainer class from the transformers library. The TrainingArguments class is used to define the training arguments such as the number of training epochs and batch size, and the Trainer class is used to fine-tune the model on the task. Tokenization is the process of turning alphabetic data into numerical representations of that data which are then inserted into a Tensor for data processing and learning on that representation.

Here's an example of tokenizing the text input "move forward and shoot the enemy" in Python using the Natural Language Toolkit (NLTK):

```
import nltk
nltk.download('punkt')

text = "move forward and shoot the enemy"
tokens = nltk.word_tokenize(text)
print(tokens)
```

```
# Output: ['move', 'forward', 'and', 'shoot', 'the', 'enemy']
```

In this example, the nltk.word_tokenize() function is used to tokenize the text input into individual words (tokens). The nltk.download() function is used to download the Punkt tokenizer, which is used by nltk.word_tokenize() to tokenize text into words.

In order to represent the tokens as numerical tensors, we need to convert them into numerical representations, such as word embeddings or one-hot encodings. Here's an example of converting the tokens into one-hot encodings using the keras.preprocessing.text.Tokenizer class from the Keras library:

```
import numpy as np
from keras.preprocessing.text import Tokenizer

text = "move forward and shoot the enemy"
tokens = nltk.word_tokenize(text)

# Initialize the Tokenizer
tokenizer = Tokenizer()

# Fit the Tokenizer on the tokens
tokenizer.fit_on_texts([tokens])

# Convert the tokens into one-hot encodings
one_hot_encodings = tokenizer.texts_to_matrix([tokens], mode='binary')
print(one_hot_encodings)

# Output: [[0. 1. 0. 0. 0. 0.]
#          [0. 0. 1. 0. 0. 0.]
#          [0. 0. 0. 1. 0. 0.]
#          [0. 0. 0. 0. 1. 0.]
#          [0. 0. 0. 0. 0. 1.]
#          [1. 0. 0. 0. 0. 0.]]
```

The results is a tensor. A tensor is a mathematical object that represents multi-dimensional arrays of data. You can think of a tensor as a generalization of matrices to higher dimensions. Just like a matrix is a two-dimensional array of numbers, a tensor can be thought of as a multi-dimensional array of numbers. The dimensions of a tensor can represent different things depending on the context, such as time, space, or any other physical or abstract quantities. For example, a scalar (a single number) is a tensor with zero dimensions, a vector is a tensor with one dimension, and a matrix is a tensor with two dimensions. Tensors are used in many areas of mathematics and science, including physics, computer graphics, and machine learning, to describe and manipulate complex data structures and relationships. The Tokenizer class is used to fit the Tokenizer on the list of tokens, and then to convert the tokens into one-hot encodings. The mode argument of the texts_to_matrix() function is set to 'binary' to represent the tokens as binary one-hot encodings, which are binary arrays with ones at the positions of the unique tokens and zeros elsewhere. The resulting one_hot_encodings tensor has shape (6, 6), where 6 is the number of tokens and 6 is the number of unique tokens in the input text.

Transformers are a type of neural network architecture that have been widely used in natural language processing (NLP) tasks. The transformer architecture was introduced in the 2017 paper "Attention Is All

You Need" by Google researchers. The transformer architecture uses self-attention mechanisms to process input sequences, which allows it to effectively handle long-term dependencies in the input data. The key component of the transformer architecture is the attention mechanism, which allows the model to weigh different parts of the input sequence when making a prediction. This allows the model to focus on the most relevant parts of the input when making a prediction, rather than using a fixed-length context window as in previous architectures like RNNs and LSTMs. The transformer architecture has been applied to a wide range of NLP tasks, such as language translation, text generation, question answering, and sentiment analysis. One of the most popular transformer models is BERT (Bidirectional Encoder Representations from Transformers), such as DeBerta model, which has been pre-trained on a large corpus of text data and fine-tuned on various NLP tasks, achieving state-of-the-art results on a wide range of benchmarks. Another popular transformer model is GPT-2 (Generative Pre-training Transformer 2) which is trained to generate human-like text. It is trained on a massive amount of data and is able to generate text that is often difficult to distinguish from text written by humans.
Other transformer-based models like XLNet, RoBERTa, ALBERT, T5, and DeBERTa have also been proposed and trained on large corpus of data, they have been fine-tuned on a variety of NLP tasks achieving state-of-the-art results.

Self-attention is a mechanism that allows a neural network to weigh different parts of the input sequence when making a prediction. It is a key component of the transformer architecture, which has been widely used in natural language processing (NLP) tasks. Self-attention works by computing a set of weights, called attention weights, for each element in the input sequence. These attention weights indicate the importance of each element in the input sequence when making a prediction. The attention mechanism then uses these weights to weigh the different elements of the input sequence and create a weighted sum of the elements, which is used as input to the next layer of the network.
Self-attention has several advantages over traditional neural network architectures like RNNs and LSTMs. One of the main advantages is its ability to handle long-term dependencies in the input data. Traditional architectures like RNNs and LSTMs use a fixed-length context window, which can make it difficult to model long-term dependencies. Self-attention, on the other hand, allows the model to focus on the most relevant parts of the input when making a prediction, regardless of their position in the input sequence. Self-attention has been used in a wide range of NLP tasks, such as language translation, text generation, question answering, and sentiment analysis. It has been particularly useful in transformer-based models like BERT, GPT-2, XLNet, RoBERTa, ALBERT, T5, and DeBERTa, which have been pre-trained on large corpus of text data and fine-tuned on various NLP tasks achieving state-of-the-art results.

Backpropagation is a supervised learning algorithm used to train artificial neural networks. It is used to update the weights of the network in order to reduce the error between the predicted output and the actual output. The goal of backpropagation is to find the gradient of the loss function with respect to the weights of the network, so that the weights can be updated in the direction that minimizes the loss.

Here's how the backpropagation algorithm works in steps:

Feedforward: The input is passed through the network, and the predicted output is computed.

Loss calculation: The error between the predicted output and the actual output is calculated using a loss function, such as mean squared error.

Propagation of the error: The error is then propagated backwards through the network, starting from the output layer and moving towards the input layer. This process involves computing the gradient of the loss with respect to the activations of each layer.

Weight update: The gradients are then used to update the weights of the network. This is typically done using an optimization algorithm, such as gradient descent, which adjusts the weights in the direction of the negative gradient.

Repeat: The process is repeated multiple times, updating the weights at each iteration until the error reaches an acceptable level or a pre-determined number of iterations have been performed.

Backpropagation is an efficient and effective algorithm for training neural networks and is widely used in deep learning and other artificial intelligence applications.

Neural Nets in Visual Recognition

Another area of AI and ML that uses Neural Nets is Visual recognition, the ability of a machine to understand and interpret visual information from images or videos. PyTorch is a popular library for building and training neural networks, and it can be used for a wide range of visual recognition tasks, such as image classification, object detection, and semantic segmentation. cv2 is a computer vision library for Python that provides a wide range of image processing and computer vision functions. It can be used in conjunction with PyTorch for image pre-processing and data augmentation, as well as for post-processing of the output of a PyTorch model. YOLO (You Only Look Once) is a popular object detection algorithm that is implemented in PyTorch. YOLO is known for its fast detection speed and its ability to detect objects in real-time. It uses a single neural network to simultaneously predict multiple bounding boxes and class probabilities for objects in an image. YOLO can be used with PyTorch to build object detection models for tasks such as self-driving cars, surveillance, and robotics. Typically you use cv2 and yolo together to create a image collection from videos and then use YOLO to classify the objects in the images, for example as a ‘person’, ‘car’, ‘bicycle’.

Convolutional Neural Networks (CNNs) are a specific type of neural network that are widely used for image recognition tasks. They are designed to automatically and adaptively learn spatial hierarchies of features from input images. They consist of multiple layers, including convolutional layers, pooling layers, and fully connected layers.

The convolutional layers are responsible for detecting local patterns or features in the input images. These patterns are learned through the use of filters, which are convolved with the input image to produce a feature map. The pooling layers are used to reduce the spatial dimensions of the feature maps, while maintaining the important information. This helps to reduce the computational cost and to make the model more robust to small changes in the position of the objects in the image. The fully connected layers are used to classify the objects in the images based on the features extracted by the convolutional and pooling layers. The MNIST dataset is a widely used dataset for image recognition tasks, it contains 70,000 images of

handwritten digits, each labeled with the corresponding digit. It is a simple dataset but it is often used as a benchmark for testing the performance of image recognition models, including CNNs. A good way to learn about visual recognition is to use the MNIST dataset as a first run at these methods.

Generative Adversarial Networks (GANs)

A Generative Adversarial Network (GAN) is a type of deep learning model that is used for generative tasks, such as image synthesis and video generation. It consists of two main components: a generator and a discriminator. The generator is responsible for generating new, synthetic data samples, while the discriminator is responsible for distinguishing between real and synthetic samples.

The generator and discriminator are trained simultaneously, with the generator trying to produce samples that are indistinguishable from real data, and the discriminator trying to correctly identify which samples are real and which are synthetic. This results in a competition or "adversarial" relationship between the generator and discriminator, with the generator trying to "fool" the discriminator and the discriminator trying to correctly identify the synthetic samples.

One of the main applications of GANs is in video games, where they can be used to generate new levels, characters, and other game assets. For example, GANs can be trained on a dataset of existing levels to generate new, unique levels. They can also be used to generate new characters or game items that are consistent with the art style and design of the game. GANs can also be used to generate new animations, cutscenes and even entire game scenarios. In this way, GANs can be used to help game developers create new content more quickly and efficiently, without having to manually design and create each individual asset.

Here is an example of a basic Generative Adversarial Network (GAN) implemented in Python using the PyTorch library:

```
import torch
import torch.nn as nn

class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.input_layer = nn.Linear(in_features=100, out_features=256)
        self.hidden_layer = nn.Linear(in_features=256, out_features=512)
        self.output_layer = nn.Linear(in_features=512, out_features=784)

    def forward(self, x):
        x = torch.relu(self.input_layer(x)) # apply activation function
        x = torch.relu(self.hidden_layer(x))
        x = torch.tanh(self.output_layer(x))
        return x

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.input_layer = nn.Linear(in_features=784, out_features=512)
        self.hidden_layer = nn.Linear(in_features=512, out_features=256)
        self.output_layer = nn.Linear(in_features=256, out_features=1)

    def forward(self, x):
```

```
            x = torch.relu(self.input_layer(x))
            x = torch.relu(self.hidden_layer(x))
            x = torch.sigmoid(self.output_layer(x))
            return x

generator = Generator()
discriminator = Discriminator()
```

This code creates two classes, one for the generator, and one for the discriminator. The generator class has three layers, an input layer, a hidden layer, and an output layer, each defined as an instance of the nn.Linear class. The input layer has 100 neurons and 256 output neurons, the hidden layer has 512 neurons and the output layer has 784 neurons (28x28 pixels). The forward method takes the input x and applies the linear layers with relu activation function on the input and hidden layers and tanh activation function on the output layer. The discriminator class also has three layers, an input layer, a hidden layer, and an output layer. The input layer has 784 neurons, the hidden layer has 256 neurons and the output layer has 1 neuron that outputs the probability of the input being real. The forward method takes the input x and applies the linear layers with relu activation function on the input and hidden layers and sigmoid activation function on the output layer. Finally, an instance of the generator and discriminator classes are created and assigned to the variable generator and discriminator respectively. These instances can then be used for training and making predictions. Please note that this is a very basic example and it doesn't provide the full picture of how to train GANs, as it doesn't include loss functions and optimizers for the generator and discriminator. GANs are known to be hard to train and there are many techniques that can help to stabilize the training process.

Boosted Decision Trees

We previously touched the topic of decision trees, a successful type of decision tree used in ML is that of boosted trees. Boosted decision trees are a type of ensemble learning method that are used to improve the performance of a single decision tree by combining the predictions of multiple weak decision trees. The basic idea behind boosting is to train a sequence of decision trees, where each tree is trained to correct the errors made by the previous trees in the sequence. One common algorithm for training boosted decision trees is called gradient boosting. The basic idea behind gradient boosting is to iteratively train decision trees to correct the residual errors made by the previous trees in the sequence. At each iteration, a new decision tree is trained to minimize the residual errors using a mathematical equation to split the tree.

The mathematical equation used to split a tree in gradient boosting is called a cost function, which is used to measure the quality of a split. The cost function is typically a measure of the impurity of the split, such as Gini impurity or information gain. The goal is to find the split that results in the lowest impurity. Gini impurity is a measure of how likely a randomly chosen element from a set would be classified incorrectly if it were randomly labeled according to the class distribution in the set. In the context of boosted decision trees, Gini impurity is used as a criterion for splitting nodes in the tree. In a decision tree, each internal node represents a test on a feature, and each branch represents the outcome of that test. When building a decision tree, the goal is to find splits that minimize the Gini impurity of the resulting subsets, so that the samples in each subset are as pure as possible with respect to their target class. The Gini impurity is calculated as the sum of the squared probabilities of each class in the set. The lower the Gini impurity, the more "pure" the set is, meaning the samples are more similar to each other with respect to the target class. In the context of gradient boosting, the Gini impurity can be used as a loss function to update the weights of the decision

trees, so that the subsequent trees attempt to correct the mistakes made by previous trees. The final prediction is made by combining the predictions from all the trees through a weighted sum.

The process of splitting a tree in gradient boosting can be mathematically represented as follows:

> Let f_m(x) be the prediction of the mth decision tree in the sequence, where x is an input sample. The goal is to find the best split point (x,y) that minimize the cost function J(x,y)
> $J(x,y) = \Sigma(y\_i - f\_m(x\_i))^2$
>
> where x_i and y_i are the input and target values of the i-th sample in the dataset.

The cost function is minimized over all possible splits (x,y) to find the best split point that minimize the residual error. This process is repeated for each decision tree in the sequence until a stopping criterion is met.

A popular and effective algorithm for boosted trees is XGBoost (eXtreme Gradient Boosting) an open-source implementation of the gradient boosting algorithm that is specifically designed for large-scale and high-performance machine learning tasks. It is widely used in various applications such as computer vision, natural language processing, and recommendation systems. An added benefit is that for many applications it does not require an expensive graphical processing unit (GPU) to be used against a data set, unlike neural networks, which are more appropriate for very large amounts of data, with longer training times. XGBoost is built on top of the gradient boosting algorithm and extends it in several ways to make it more efficient and scalable. Some of the key features of XGBoost include:

- Tree Pruning: XGBoost uses a technique called tree pruning to remove unnecessary splits and reduce the size of the decision trees. This helps to prevent overfitting and improve the generalization performance of the model.

- Regularization: XGBoost includes several regularization techniques such as L1 and L2 regularization to prevent overfitting. L1 and L2 regularization are techniques used in machine learning to prevent overfitting and improve the generalization performance of models.

  > L1 regularization, also known as Lasso regularization, adds a penalty term to the loss function of a model that is proportional to the absolute value of the model's coefficients. This penalty term encourages the model to have as many coefficients as possible equal to zero, effectively performing feature selection by removing unimportant features from the model. L1 regularization can result in sparse models with few non-zero coefficients, which can be useful in situations where there are many features and only a small number of them are expected to be important.
  >
  > L2 regularization, also known as Ridge regularization, adds a penalty term to the loss function of a model that is proportional to the square of the model's

coefficients. This penalty term encourages the model to have smaller coefficients, effectively shrinking the coefficients towards zero. L2 regularization can help to reduce the impact of collinearity and improve the stability of the model's estimates.

In both cases, the strength of the regularization penalty is controlled by a hyperparameter that needs to be tuned using a validation set or cross-validation. By adding the regularization penalty to the loss function, the model's capacity to fit the training data is reduced, which can help prevent overfitting and improve the model's ability to generalize to new data.

L1 and L2 regularization are commonly used in linear regression, logistic regression, and neural networks, but they can be applied to a wide range of other models as well.

- Parallel Processing: XGBoost supports parallel processing by distributing the computation of the decision trees across multiple machines or cores. This makes it possible to train large models on large datasets in a relatively short amount of time.

- Handling missing values: XGBoost uses a technique called column sampling with column-wise split to handle missing values in the dataset.

- Handling sparse data: XGBoost implements a technique called block structure to handle sparse data. It stores and processes the data in a block format, which is more memory-efficient.

- Out-of-Core Learning: XGBoost can handle large datasets that don't fit in memory by using a technique called out-of-core learning. It loads a small subset of the data into memory, trains a model on that subset, and then loads the next subset of the data and repeats the process.

Here is an example of how to train an XGBoost classifier in Python using the scikit-learn API, with a learning rate hyperparameter:

```
import xgboost as xgb
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Generate some sample data
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2)

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Create an XGBoost classifier with a learning rate of 0.1
clf = xgb.XGBClassifier(learning_rate=0.1)
```

```
# Fit the classifier to the training data
clf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = clf.predict(X_test)

# Print the accuracy of the classifier
print("Accuracy:", accuracy_score(y_test, y_pred))
```

In this example, we first import the necessary libraries (xgboost, sklearn), generate sample data using the make_classification function from scikit-learn, split the data into training and test sets using the train_test_split function, create an XGBoost classifier with a learning rate of 0.1, fit the classifier to the training data, make predictions on the test set, and print the accuracy of the classifier. You can adjust the learning rate value to see how it affects the classifier's performance. Also, you can try other hyperparameters such as max_depth, subsample, colsample_bytree, n_estimators to see how they affect the performance.

Learning rate is one of the most important hyperparameters in machine learning, especially in deep learning. It controls the step size at which the algorithm updates the parameters of the model during training. A small learning rate results in slow convergence, while a large learning rate can cause the model to overshoot the optimal solution and diverge. In this article, we will discuss the importance of the learning rate, how it affects the training process, and different strategies for setting the learning rate. The learning rate controls the step size at which the algorithm updates the parameters of the model during training. A small learning rate results in slow convergence, while a large learning rate can cause the model to overshoot the optimal solution and diverge. The learning rate is often represented by the Greek letter eta (η) and is typically denoted as α in the literature.

The learning rate is an important hyperparameter because it controls the trade-off between the speed of convergence and the accuracy of the model. A small learning rate will converge slowly but will result in a more accurate model, while a large learning rate will converge quickly but will result in a less accurate model. Finding the optimal learning rate is a trade-off between these two factors and is crucial for the performance of the model. When training a model, the learning rate is typically set using one of two methods: a fixed learning rate or an adaptive learning rate. A fixed learning rate is set to a constant value throughout the training process, while an adaptive learning rate changes the learning rate during training based on the performance of the model.

One popular method for setting a fixed learning rate is to use a small value for the initial stages of training and gradually increase it as the model converges. This is known as the learning rate schedule or learning rate decay. This approach helps the model converge quickly in the initial stages and then fine-tune the parameters as it gets closer to the optimal solution.

Another popular method for setting the learning rate is to use an adaptive learning rate algorithm, such as Adam or Adagrad, the learning rate scheduler in PyTorch. These algorithms adjust the learning rate during training based on the performance of the model. For example, Adam uses a combination of gradient descent and moving averages to adjust the learning rate, while Adagrad adapts the learning rate to the parameters, giving more weight to the parameters that are updated more frequently. The learning rate is a crucial hyperparameter in machine learning, especially in deep learning. It controls the step size at which the algorithm updates the parameters of the model during training. Finding the optimal learning rate is a trade-

off between the speed of convergence and the accuracy of the model. There are several strategies for setting the learning rate, including fixed learning rate, learning rate schedule, and adaptive learning rate algorithms. Experimenting with different learning rate values and strategies is an important part of the model training process.

Cautionary Tales of Statistical Methods

There are some very important things to understand with Statistical Methods in Artificial Intelligence. The very basis of using statistics is a mathematical approximation of real things in the real world. Therefore, this approximation will always be an oversimplification of real things. However, for a mathematical based machine, such as the computer, it must use mathematical methods and not necessarily related to natural mechanics, to formulate a response, that which is returned from an Algorithm. The next section will go over Machine Bias that develops from these mathematical statistical methods. We should not treat returns from computation to be some sort of objective truth, rather it is affected by mathematical equations and the limitations of approximation.

Another complicating factor, and source for noise and results filled with errors if algorithms that are not properly set up, is the use of Probability in AI. Ian Goodfellow notes the use of probability in AI:

> Probability theory is a mathematical framework for representing uncertain statements. It provides a means of quantifying uncertainty as well as axioms for deriving new uncertain statements. In artificial intelligence applications, we use probability theory in two major ways. First, the laws of probability tell us how AI systems should reason, so we design our algorithms to compute or approximate various expressions derived using probability theory. Second, we can use probability and statistics to theoretically analyze the behavior of proposed AI systems.
>
> Machine learning must always deal with uncertain quantities and sometimes stochastic (nondeterministic) quantities. Uncertainty and stochasticity can arise from many sources. Researchers have made compelling arguments for quantifying uncertainty using probability since at least the 1980s. (Goodfellow, et al, 2016, Ch. 3)

Goodfellow et al go on to talk about 3 possible sources of uncertainty: Inherent stochasticity (non-determinism) in the system being modeled; Incomplete observability and Incomplete modelling. In the first case randomness has an effect, in the second, which happens in an open world, if we do not have complete data or know all the pieces of a chess game and a limited board, then we have Incomplete Observability. In the third case, the problem of outliers is a good example, AI algorithms ignore data that lies at the periphery in relation to the other data points, and on the other hand, if there is too much specificity in rules, it can lead to overfitting and break the model.

Probability extends logic to deal with uncertainty. Logic provides a set of formal rules, such as Piercean Logic, for determining what propositions are implied to be true or false given the assumption that some other set of propositions is true or false or in the Piercean Logic the third option: true & false, which raises certain questions about non-Boolean methods for validation checking, etc. Probability theory provides a set of formal rules for determining the likelihood of a proposition being true given the likelihood of other

propositions. As one will not find any examples from Russell & Norvig, researchers and authors of what is considered the textbook on AI of Piercean Logic other than mention of C.S. Pierce in historical summaries, if developers use Piercean Logic then what form of validation do they use to check it is an accurate result? Since, Piercean Logic is used in defense sector computer engineering for automation and control this is an important question.

Another source of noise or error in computational algorithms that rely on Artificial Intelligence is the problem of numerical computation. Goodfellow et al explain:

> The fundamental difficulty in performing continuous math on a digital computer is that we need to represent infinitely many real numbers with a finite number of bit patterns. This means that for almost all real numbers, we incur some approximation error when we represent the number in the computer. In many cases, this is just rounding error. Rounding error is problematic, especially when it compounds across many operations, and can cause algorithms that work in Numerical Computation theory to fail in practice if they are not designed to minimize the accumulation of rounding error. One form of rounding error that is particularly devastating is underflow. Underflow occurs when numbers near zero are rounded to zero. Many functions behave qualitatively differently when their argument is zero rather than a small positive number. Another highly damaging form of numerical error is overflow. Overflow occurs when numbers with large magnitude are approximated as ∞ or $-\infty$. Further arithmetic will usually change these infinite values into not-a-number values. One example of a function that must be stabilized against underflow and overflow is the softmax function.
> (Goodfellow et al, 2016, Ch. 4)

Yet, as many programmers of algorithms can tell you there is often the binary choice between 0,1. So, if we always assume a division between 0,1 then the rounding error may come into play. As one can see just because we have a machine that performs at unbelievably fast rates, and usually produces correct results, with these and other complicating factors such a system that is not accurate and precise can generate large errors, that can also grow exponentially over time. One journalist writing about the use of AI in major corporations makes a great point:

> Mathematicians say that it's impossible to make a "perfect decision" because of systems of complexity and because the future is always in flux, right down to a molecular level. It would be impossible to predict every single possible outcome, and with an unknowable number of variables, there is no way to build a model that could weigh all possible answers. Decades ago, when the frontiers of AI involved beating a human player at checkers, the decision variables were straightforward. Today, asking an AI to weigh in on a medical diagnosis or to predict the next financial market crash involves data and decisions that are orders of magnitude more complex. So instead, our systems are built for optimization. Implicit in optimizing is unpredictability—to make choices that deviate from our own human thinking. (Webb, 2019)

These are just some of the factors that can also lead to Machine Bias in AI Models, since results from an AI algorithm depend on the correctness of the Model a bad model can lead to a completely destructive algorithm. There are also other sources of Machine Bias which we shall now cover.

Machine Bias

In a much reported incident it was discovered that Africans in Image Recognition software were being identified as Gorillas rather then as humans[1]. This raised great concerns about the bias of Machine Intelligence algorithms employed in Image Recognition, although it is true that a hard-right political activist, Robert Mercer, was involved in the development of Image Recognition software while at IBM, it was discovered that the problem was in the AIs inability to properly deal with dark colors. This is just one example of Machine Bias in an AI system, once again, if we do consider a machine to be neutral and objective, this is not always true. For instance, problems are also reported in automated policing systems that rely on AI, where it seems to target in the US, African-Americans. These results were also demonstrated in automated judicial processes where AI is employed to decide court cases. As well there is the episode where Microsoft released a chat bot AI that quickly was skewed to voice far-right neo-Nazi rhetoric based on 'data poisoning' attacks (see below), and Google Search algorithms that also skewed toward the far right [2]. In dealing with automated Intelligence systems, never mind that generally the culture in Intelligence is biased toward the Right, there would be necessarily already existing biases which could slant the AI system into an even more biased depiction of actual events in the world. Just ask yourself, how would a system that is investigating ISIS, or other Islamist Jihadist groups, view Muslims in general, would it be biased? A paper on the dangers of Lethal Autonomous Weapons Systems (LAWS) points out some of the dangers of machine bias:

> However, as 'intelligent' software and machines need to be 'fed' by a huge amount of data in order to 'learn' (a trait that we deem 'intelligent'), there exists the risk that they learn human prejudices from biased data. And so-called machine biases constitute a danger for AI-controlled or autonomous systems that some experts regard as far more acute than LAWS. Based on the data a bot is fed in order to learn, it could learn, e.g., to discriminate against people of color or minorities, or gain a strict political attitude. (Shurber, 2018, pg. 17-8)

Bias is not just limited to the problem of Intelligence or of Social Sciences. It can even be encountered in autonomous systems calculating each other in Financial Markets, when autonomous agents (or semiotic agents) are left to their own devices they capitalize, leading to economic bias:

> Researchers at the University of Bologna in Italy created two simple reinforcement-learning-based pricing algorithms and set them loose in a controlled environment. They discovered that the two completely autonomous algorithms learned to respond to one another's behavior and quickly pulled the price of goods above where it would have been had either operated alone.
> (Calvano et al, 2018)

Although, we may assume that since, in many cases, we can look at the code of an AI algorithm that we

would be able to understand what is happening from the results of an AI system. However, this is not always true, here we encounter what is known as the 'blackbox' problem. In Software Engineering we have such terms as white box and black box testing, in these cases a black box tester does not have access to the code or inner workings of the software undergoing testing, but has to figure out bugs based on this blindedness, which can actually be of more value, since the tester is not biased by the code and has knowledge of what it's intended purpose is but rather can only go on the effects. In AI the blackbox comes into affect in a more complex way:

> That inability to observe how AI is optimizing and making its decisions is what's known as the "black box problem." Right now, AI systems built by the Big Nine might offer open-source code, but they all function like proprietary black boxes. While they can describe the process, allowing others to observe it in real time is opaque. With all those simulated neurons and layers, exactly what happened and in which order can't be easily reverse-engineered. (Webb, 2019)

As is pointed out even the developers that code the algorithm cannot fully give an account or even properly anticipate the results of an AI algorithm due to the blackbox problem. Now imagine, developing an Automated Surveillance system with this problem in mind and also bias, the dangers of such a system are not hard to realize.

Security of AI

Data poisoning in machine learning refers to the intentional or unintentional injection of malicious or incorrect data into the training dataset of a machine learning model with the goal of manipulating its behavior. This can happen in various ways, for example:

> An attacker might add instances to the training data that are specifically designed to cause the model to make incorrect predictions. An attacker might also manipulate existing instances in the training data to cause the model to make incorrect predictions.

A researcher might unknowingly use biased or contaminated data in their experiments, resulting in a poisoned model. Data poisoning can have severe consequences, as it can cause a machine learning model to make incorrect or biased predictions, undermining its usefulness and potentially causing harm. The effects of data poisoning can be especially severe in sensitive applications such as medical diagnosis, self-driving cars, and financial fraud detection, where incorrect predictions can have serious consequences.

To prevent data poisoning, it is important to ensure that the training data is cleaned, preprocessed and validated. One way to do this is by using techniques such as data sanitization and data validation. Additionally, it's important to monitor the performance of the model during training and testing to detect any signs of data poisoning.

Machine learning also poses certain cyber security concerns that need to be addressed. Some of the main cyber security concerns with machine learning include:

> Data privacy and security: Machine learning models are trained on large amounts of data, which can include sensitive information such as personal information, financial data, and medical records.

This data needs to be protected from unauthorized access and misuse to prevent data breaches and protect individuals' privacy.

Adversarial attacks: Machine learning models are vulnerable to adversarial attacks, in which an attacker attempts to manipulate the input data to cause the model to make incorrect predictions. This can be done by introducing small, carefully crafted changes to the input data (such as adding noise to an image), which can cause the model to make incorrect predictions without being detected.

Model inversion attacks: Machine learning models can be used to reverse-engineer sensitive information from the model's internal representations. This can be done by feeding the model input data and observing the internal representations, which can reveal sensitive information such as personal information, medical records, and financial data.

Poisoning attacks: Machine learning models are also vulnerable to poisoning attacks, in which an attacker injects malicious data into the training data in an attempt to manipulate the model's behavior. This can cause the model to make incorrect predictions and can be difficult to detect.

Privacy issues with federated learning: Federated learning is a machine learning technique in which data is distributed among multiple devices, rather than being centrally stored. This can improve data privacy, but it also raises concerns about the security of the data on individual devices and the potential for data breaches.

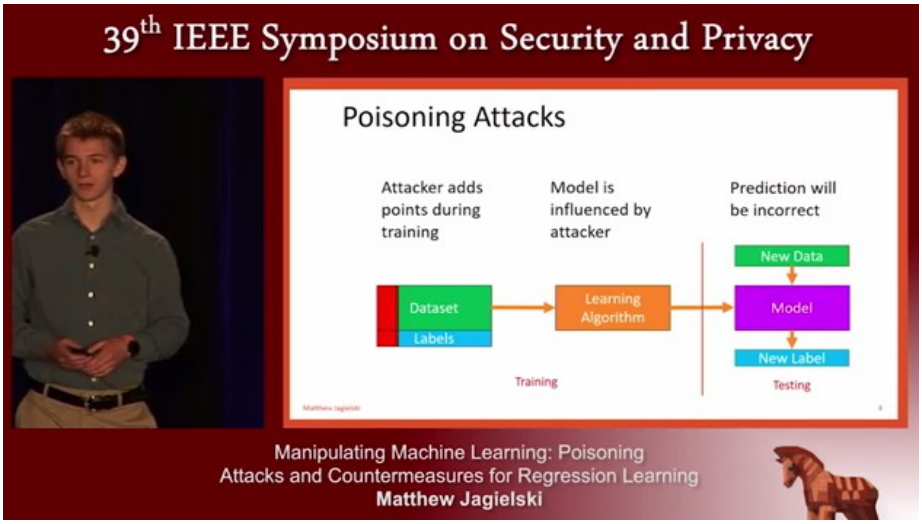
To mitigate these cyber security concerns, it's important to implement robust security measures such as data encryption, secure communication protocols, and access controls. Additionally, it's important to monitor the performance of the model during training and testing to detect any signs of attacks, and to use techniques such as adversarial training to make the model more robust to attacks. It should also be pointed out that ML is increasingly being used by crackers or ‘hackers’ to infiltrate systems in an automated way.

As mentioned above attacks on Machine Learning algorithms can be accomplished through data poisoning. Researchers define data poisoning:

> Machine learning systems trained on user-provided data are susceptible to data poisoning attacks, whereby malicious users inject false training data with the aim of corrupting the learned model. While recent work has proposed a number of attacks and defenses, little is understood about the worst-case loss of a defense in the face of a determined attacker. (Steinhardt et al, 2018)

In a certain sense one can see a parallel between data poisoning and Information Operations, or Thought Injection attacks, which resembles the hacking technique of SQL Injection attacks were a hacker injects SQL code via http requests to corrupt or poison a database or dataset or collection. The study of data poisoning began around 2012, some time after ML was used in Defense contracting, whether there are mechanisms to protect against data poisoning in their systems is unknown due to the classified nature of such work. There are countermeasures being developed against such attacks. At the 2018 IEEE

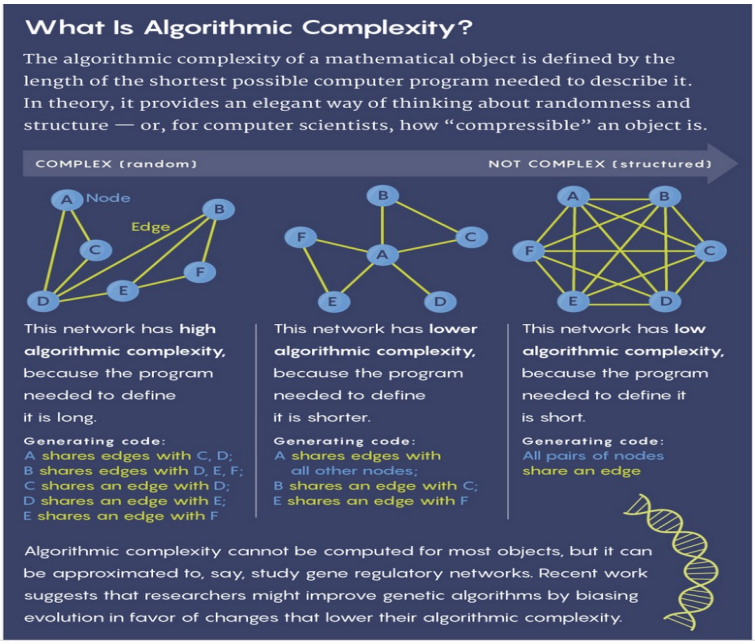
Symposium on Security and Privacy Matthew Jagielski provided one such methodology. In the below diagram we can see the flow of the attack vector and the results. Again, validating results becomes a key criteria in developing resilient and accurate algorithms, so that developers will have to be vigilant with their Test and Validate cycles in AI as it becomes larger and larger in our society.



(Jagielski, 2018)

Algorithmic Complexity

Aside from Machine Bias another area of concern in algorithm engineering is that of Algorithmic Complexity (See Graph below). One element that is a complicating factor in algorithmic complexity is the degree to which other components of a system are visible and connected to other elements within the program. As such, a compartmentalized non-visible architecture as used in covert operations and computation will by necessity of it's invisibility to each component create an complex algorithmic environment. Thus leading to unforeseen programming outputs and possibly circular and contradictory logic. This is the opposite of that foreseen in the Cybernetics of Stafford Beer. Where each node in the algorithmic management system is visible. In this sense Beer's algorithms, which are being developed by sustainability advocates for such things as democratic finance, is the opposite of a covert system. Transparency thus giving it a technical advantage. Beer's system would resemble the nodes on the right in the chart below, whereas a covert system would resemble the graph on the left below.



A complete example of a XGBoost Algorithm that puts the preceding concepts into action:

```
#Imported python modules
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from xgboost import XGBClassifier
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score, KFold
from sklearn.metrics import accuracy_score, classification_report

#Reading Dataset
df = pd.read_csv('data/stars csv.csv') #reads data into a pandas.DataFrame
print(df.head())
print(df.shape)

#Checking for null values
print(df.isnull().sum())
print(df.describe())
print(df.corr())
sns.pairplot(df)
plt.show()

#EDA Section
#Correlations of each feature in dataset
corrmat = df.corr()
top_features = corrmat.index
plt.figure(figsize = (20,20))

g = sns.heatmap(df[top_features].corr(), annot = True, cmap = "Blues")
```

```
plt.show()

plt.figure()
df.hist(figsize=(20,20))
plt.show()

#Setting independant and target variables
X = df.drop(['Star color', 'Spectral Class'],axis=1) #labels, features
y = df['Star type'] #target variable
#Splitting Data into training and testing data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15,
random_state=42,shuffle=True)

print(X.head())
print(y.head())

xgbc = XGBClassifier(max_depth=3,
                     subsample = 0.8,
                     n_estimators=200,
                     learning_rate=0.8,
                     min_child_weight=1,
                     reg_alpha=0,
                     reg_lambda=1
                     )
print(xgbc)

xgbc.fit(X_train, y_train)
y_predict = xgbc.predict(X_test)
y_train_predict = xgbc.predict(X_train)

# - cross validataion
scores = cross_val_score(xgbc, X_train, y_train, cv=5)
print("Mean cross-validation score: %.2f" % scores.mean())

kfold = KFold(n_splits=10, shuffle=True)
kf_cv_scores = cross_val_score(xgbc, X_train, y_train, cv=kfold )
print("K-fold CV average score: %.2f" % kf_cv_scores.mean())

ypred = xgbc.predict(X_test)
#Confusion Matrix
cm = confusion_matrix(y_test,ypred)
print(cm)
#Accuracy
print('train accuracy', accuracy_score(y_train, y_train_predict))
print('test accuracy', accuracy_score(y_test, ypred))
```

code file available at: https://github.com/autonomous019/stars_classifier_xgboost/blob/main/stars.py

2 AI IN VIDEO GAME DESIGN

Video games or simulation software are of several genres. An overview is presented below before we dive into the programming involved in creating military and intelligence simulations or real games.

Game and/or Simulation Genres:

Action - fast-paced requiring quick judgement and snap decisions. Often they can be single player, with AI Agents as team members, fighting adversarial AI agents, or Non-Player Characters (NPCs).

Adventure - a sequence basis to the game, with a rigid structure. With no wide game maps, usually the gamer is presented only with their immediate surroundings.

Role-Playing (RPGs) - gamer takes on the role of a character.

Vehicle Simulation - where the gamer controls a vehicle, such as 'Pole Position'

Strategy - of course the big dog in this category is 'Real-Time Strategy Games' (RTS), which are covered extensively below. In this genre the gamer has a high level view of the game world.

Management - sometimes referred to as ‘god’ games, lhas a high-level games, like creating cities or forts, resources can be manipulated in the game world, change it in different ways for the NPCs in the game world.

Puzzle - small games in single player mode, gamer uses logic and deduction to complete goals.

One concept that is important to understand in Simulation Software is that of Game World- the environment the game is played in. In video game design, a game world is the virtual space or environment where the game takes place. It is the setting in which the player interacts with the game's characters, objects, and systems. The game world is often designed to be immersive, allowing the player to feel as if they are a part of the game's fictional universe. A well-designed game world can contribute greatly to the overall gaming experience by providing a sense of place and atmosphere. It can also impact gameplay by providing different environments that require different strategies and tactics to navigate successfully.

A game world can take on many different forms, depending on the genre and design of the game. It can range from a realistic representation of a historical city, to a fantastical world of magic and mythical creatures. The game world can also change over time, such as in games with dynamic weather or day/night cycles. Creating a game world is a complex process that involves many different elements, including level design, art design, narrative, and gameplay mechanics. The world must be designed to support the game's objectives and the player's experience. This requires careful consideration of the game's story, characters, and gameplay mechanics, as well as the technical limitations and capabilities of the game engine.

Game World types:

Accessible vs. Inaccessible

Accessible- an actor has knowledge of every aspect of the game world and knows of everything that is going on within that game world

Inaccessible- there are limits to what an actor may know about the game world (for example using the concept of fog-of-war, stochastic information or incomplete information about game world)

Environmentally Discrete vs. Environmentally Continuous

Actors within the game world may make a number of possible actions at any point, determined by the range of potential actions within a game world

Discrete-a finite set of actions that an actor can take (for example only being able to move one square in any one of the cardinal directions on a grid)

Continuous- there is a continuum of possible actions, such as allowing an actor to turn to any direction

Static vs. Dynamic

Static- the game world remains the same until an participant has made a move
Dynamic- the game world alters whilst an participant is “thinking”

Deterministic vs. Non-deterministic

Deterministic-the next state can be explicitly concluded from the present state of the game world and the actions carried out by the actors.

Non-deterministic-there is an element of uncertainty, or if the game world changes despite actions by the actors

Episodic vs. Non-Episodic

Episodic-If an actor can take an action, the results of which have no relation on future actions, the environment is episodic.

Non-episodic-the consequence of one action relates directly or indirectly to the available information or set of actions at a future point

Turn-Based vs. Real-time

Turn-based-place the players in a game-playing sequence. Whilst this type of game could be, theoretically, applied to any game, there are only a small number of genres where this mechanic is used, primarily 4X, strategy, some role-playing, and some life-simulation games. These games can require a great deal of strategic thinking, and as such having the time to analyse a situation and make decisions based on that is almost necessary.

Semi-Turn-based- the player has the opportunity to pause the game to make decisions or queue up actions, and then return to normal real-time playing afterwards; or where certain sections of the game are turn-based, and
the rest is real-time.

Real-Time- Non-turn-based games

Artificial Intelligence is used in almost all simulations or games currently. There are different AI’s that meet different programming needs in different game genres. The following are the different components of AI used in video game design:

Pathfinding and Navigation: This is the ability of game characters to navigate the game world intelligently. It allows them to move around obstacles, find the best routes, and avoid dangerous areas. This is important in games where movement and positioning are key components of gameplay, such as strategy games or first-person shooters.

Behavior Trees: This is a way of creating intelligent and realistic game characters. A behavior tree is a decision-making system that allows game developers to create complex behaviors and actions for their characters. This can include things like attacking, defending, or running away.

Procedural Generation: This is a technique used in game design to create game content automatically. It allows game developers to create large, complex game worlds with less manual effort. This can include things like generating terrain, landscapes, and even enemy AI.

Machine Learning: This is the ability of AI systems to learn and adapt over time. Machine learning can be used in games to create more intelligent and challenging opponents for players. This can include things like adaptive difficulty levels, where the game AI adjusts to the player's skill level.

Physics Simulations: This is the ability of games to simulate the laws of physics in the game world. It allows game developers to create realistic environments, where objects can be manipulated and interacted with in a realistic way. This is important in games that involve physics-based puzzles or real-time physics-based combat.

Natural Language Processing: This is the ability of AI systems to understand and interpret human language. In games, this can be used to create more immersive and interactive dialogue systems, where players can engage with game characters in a more natural and intuitive way.

World Interfacing

...building robust and reusable world interfaces using two different techniques: event passing and polling. The event passing system will be extended to include simulation of sensory perception, a hot topic in current game AI. Polling [when the polling objects are also game characters] Looking for interesting information is called polling. The AI code polls various elements of the game state to determine if there is anything interesting that it needs to act on. (Millington & Funge 2016)

Discrete action games within game theory consist of a finite number of participants, turns, or outcomes, resulting in a finite set of strategies which can be plotted in a matrix format for evaluation. Continuous action games, however, can have participants joining and leaving the game, or the stakes changing between actions, resulting in a continuous set of strategies. This represents a subset of the potential actions that the game world allows. Within our taxonomy, this relates directly to environmentally discrete and environmentally continuous game worlds.

Simultaneous vs. Sequential

In direct relation to turn-based versus real-time games, sequential games have all the players within a game make their moves in sequence, and one at a time. Simultaneous games are those where any or all players may make their moves at the same time. Classically, sequential games are also called dynamic games: however this would cause confusion in our taxonomy. Sequential games allow the

construction of the extensive form of the game - essentially a hybrid decision tree of all players and all possible moves with their rewards.

Information Visibility

It is not necessary for all participants within a game to have access to all information about the state of the game at any point. The available information can be perfect, where all participants have access to
the current state of the game, all possible strategies from the current state, and all past moves made by all other participants. The latter implies that all games that impart perfect information to the participants are by their nature also sequential games. Imperfect games impart partial information about the game to at least one participant. A special case of imperfect information visibility is complete information where all participants are aware of all possible strategies and the current state of the game, however the previous moves by other participants are hidden. In this taxonomy, this relates to accessible and inaccessible game worlds. However, a common observation in games is that artificial
participants have access to perfect information of the game world, whereas the human player only has imperfect information. This breaks the immersion of the game.

Noisy vs. Clear

Noisy game worlds are those in which there is a significant amount of information that an intelligence must analyze to either make a decision on what to do next, but where not all of that information is
appropriate to the goal. Both dynamic and non-deterministic games provide levels of noise: the former due to the fact that the state of the game keeps changing even during the times when the intelligence
needs to make a decision or form behavior from learning; and the latter where there is no clear progressive state from which to base rules and analyze the game world. Although these two definitions
provide the clearest example of noise, the level of information available can also create noise. Even in perfect information game worlds it is possible that the available information is an overly large data set
for any intelligence, and thus noise is introduced.

In recent years, neuro-evolution techniques have emerged as a powerful tool in video game design. These techniques use evolutionary algorithms to optimize artificial neural networks (ANNs), which can be used to create intelligent game agents that can learn and adapt over time. By using neuro-evolution techniques, game developers can create more engaging and challenging games that provide players with a more immersive experience.

One of the key benefits of using neuro-evolution techniques in video game design is that it allows game developers to create game agents that can learn from their experiences. This is achieved through a process of evolution, where the Artificial Neural Networks (ANNs) are optimized over multiple generations. In each generation, the ANNs are evaluated based on their performance in the game, and the best performing ANNs are selected to produce the next generation. This process is repeated until an optimal set of ANNs is found.

Another benefit of using neuro-evolution techniques is that it allows game developers to create game agents that can adapt to different game scenarios. For example, an ANN that is trained to play a specific game level can be retrained to play a different level or even a completely different game. This means that game developers can create more flexible and adaptable game agents that can be used across multiple games or game modes.

Neuro-evolution techniques can be used in a variety of ways in video game design. For example, they can be used to create more intelligent enemy AI in first-person shooter games, or to create more challenging opponents in strategy games. They can also be used to generate game content, such as game levels or puzzles, or to assist players by providing in-game hints or tutorials. One example of a video game that uses neuro-evolution techniques is the racing game, Forza Motorsport 4. In this game, the AI opponents are generated using a neuro-evolution algorithm, which allows them to learn and adapt to the player's driving style. This creates a more immersive and challenging experience for players, as the game opponents become more skilled and competitive over time.

As researchers remark, pointing out the use of Neuro-Evolution techniques (AI creating AI):

> However existing work is primarily focused on the specific implementation of AI methodologies in specific problem areas. For example, the use of neuro-evolution to train behavior in NERO. With greater analysis of the problems faced in implementing AI methods in computer games, more accurate and efficient methodologies can be developed to create more realistic behavior of artificial characters within games.
> (Gunn et al, 2009, 1)

There are no universal methodologies that cross all simulation categories. AI in games can be thought of in the following ways taken from Gunn et al 2009:

Hierarchical Intelligence

> Player-as-manager games provide us with a potential hierarchy of intelligences that would be required: the artificial player and the artificial player's actors. Different AI methods would be required in this case, as the artificial player would require high-level strategic decision making. On the other hand, individual actors might only require reflexive behavior (i.e. "I am being shot, I shall move away.") Currently in these types of games (especially strategy games) there is little intelligence at the artificial player level, merely consisting of such static tactics as "build up a force of x units, and send them along

y path". Observation of such tactics in has shown that there is a reliance on some form of state analysis. By considering the hierarchical nature of the player and the actors under that player's control, suitable mechanisms can be introduced. First to provide adequate highlevel strategic planning for the artificial player. Secondly to provide low-level tactical planning for the artificial player's actors.

Co-operative vs. Non-Cooperative

Cooperative games are those where the participants can form binding agreements on strategies, and there is a mechanism in place to enforce such behaviour. Non-co-operative games are where every participant is out to maximise their own pay-off. Some games may have elements of both co-operative and non-co-operative behaviour: coalitions of participants enforce co-operative behaviour, but it is still possible for members of the coalition to perform better, or receive better rewards than the others if working alone. These are hybrid games.

Competing and Cooperation in Games

Nash Equilibria

Within game theory, a Nash equilibrium (NE) exists where an overall highest level of pay-off for all players takes place. There may be many such equilibria within game strategies for a particular game, or
there may be only one - in which case it is a unique NE.

Zero-sum, these are a type of game in game theory that are a special case of general sum games - ones where there is a fixed overall value to winning (or losing) the game. The specific case where for any
winning value v, there is a losing value of 0 - v, is a zero-sum game. In other words, what one player wins, the other loses.

By understanding these concepts, game developers can create games that are challenging and fair, while also providing players with a satisfying experience.

Nash equilibrium is a concept in game theory that refers to a situation where no player can improve their outcome by changing their strategy, assuming that all other players keep their strategies the same. In other words, in a Nash equilibrium, each player's strategy is the best response to the strategies of all other players. This concept is important in video game design because it ensures that games are balanced and fair, and that players cannot gain an unfair advantage by changing their strategy.

Zero-sum games are a specific type of game where one player's gain is another player's loss. In other words, the sum of the gains and losses for all players is zero. This concept is also important in video game design because it creates a competitive environment where players must outperform their opponents to succeed. The combination of Nash equilibrium and zero-sum games is a powerful tool in video game design. By creating games that are zero-sum, game developers can create a competitive environment where players must outperform their opponents to succeed. This can lead to engaging gameplay and can keep players engaged and interested.

Additionally, by using Nash equilibrium, game developers can ensure that games are balanced and fair, and that players cannot gain an unfair advantage by changing their strategy. This can lead to a more satisfying experience for players, as they feel that they are competing on an even playing field. One example of a game that uses Nash equilibrium and zero-sum games is the classic board game, chess. In chess, each player's objective is to checkmate the opponent's king, which is a zero-sum game because one player's gain is the other player's loss. Additionally, chess is a game of perfect information, meaning that both players have access to all of the same information. This ensures that the game is fair and balanced, and that each player must rely on strategy and skill to succeed.

Hiding information can be an effective game design strategy for creating a more engaging and challenging gaming experience for players. By concealing certain information from the player, game developers can create a sense of mystery and unpredictability that can keep players engaged and interested. Here are a few ways that hiding information can help a gamer playing a video game:

Increases Challenge: By hiding information from the player, game developers can make the game more challenging. For example, in a puzzle game, if the solution to the puzzle is immediately visible, the challenge of the game may be diminished. However, if the solution is hidden and must be discovered by the player, the game becomes more challenging and engaging.

Creates Mystery and Intrigue: Hiding information can create a sense of mystery and intrigue in the game. For example, in a detective game, if the identity of the suspect is revealed at the beginning of the game, the player may lose interest. However, if the identity is hidden, the player will be motivated to continue playing to discover the truth.

Provides a Sense of Discovery: When information is hidden in a game, players are given the opportunity to discover it on their own. This can be a rewarding experience for players as they feel a sense of accomplishment when they uncover hidden information. This can also create a sense of immersion in the game world, as players feel like they are uncovering secrets and unlocking hidden areas.

Adds Replayability: Hiding information can also add replayability to a game. For example, in a role-playing game, if the player knows the outcome of every decision they make, they may be less likely to replay the game. However, if the game has hidden storylines or multiple outcomes, the player will be more motivated to replay the game to uncover all of the possibilities.

A video game that uses Nash equilibrium is the popular battle royale game, Fortnite. In Fortnite, up to 100 players are dropped onto an island where they must scavenge for weapons and resources while trying to be the last person or team standing. The game uses a combination of Nash equilibrium and zero-sum games to create a competitive and engaging experience for players. First, the game is a zero-sum game because there can only be one winner. This creates a competitive environment where players must outperform their opponents to succeed. Additionally, the game uses Nash equilibrium by ensuring that each player's strategy is the best response to the strategies of all other players. For example, if a player chooses to land in a popular area of the map where there are many other players, they will have a higher chance of finding better weapons and resources, but they also risk being eliminated early in the game. Alternatively, a player may choose to land in a quieter area of the map where there are fewer players and resources, but they are also less likely to encounter other players early on. The game also incorporates elements of imperfect information, as players do not always have access to the locations of all other players on the map. This creates a sense of mystery and unpredictability, as players must be constantly vigilant and adapt their strategies to changing circumstances.

Game designers use a variety of psychological tricks to influence gameplay in video games. These tricks are designed to keep players engaged, motivated, and entertained. Here are a few common psychological tricks used by game designers:

Reward Systems: Game designers use reward systems to motivate players to keep playing. Rewards can include in-game items, achievements, or unlocking new levels or areas. These rewards tap into the player's desire for achievement and progression, creating a sense of accomplishment and satisfaction.

Randomness: Game designers use randomness to create an unpredictable game environment. Random events and rewards can keep players engaged and interested by creating a sense of mystery and excitement. Randomness can also increase replayability by creating a different experience each time the game is played.

Fear of Missing Out (FOMO): Game designers use FOMO to encourage players to keep playing. This can include limited-time events, in-game sales, or exclusive items that are only available for a short time. FOMO can create a sense of urgency and make players feel like they are missing out if they don't play.

Social Pressure: Game designers use social pressure to create a sense of community and competition among players. This can include leaderboards, online multiplayer, or social media integration. Social pressure can motivate players to keep playing to beat their friends or improve their rankings.

Skinner Box Mechanics: Game designers use Skinner box mechanics, which are named after the famous behavioral psychologist B.F. Skinner. These mechanics use operant conditioning, which means rewarding a desired behavior and punishing an undesired behavior. This can create a sense of compulsion in players and keep them playing the game even if they are not enjoying it.

Personalization: Game designers use personalization to create a sense of ownership in players. This can include customization options for characters, weapons, or other in-game items. Personalization can create an emotional connection between the player and the game, making the player more invested in the experience.

Game designers use a variety of psychological tricks to influence gameplay in video games. These tricks tap into players' desires for achievement, progression, unpredictability, and social interaction. By using these tricks, game designers can create engaging and entertaining games that keep players coming back for more. As video game technology continues to advance, we can expect to see even more innovative and sophisticated uses of psychology in game design.

Game designers use player expectations to steer gameplay by leveraging what players know or expect from previous games or experiences. By building on players' existing knowledge and expectations, designers can create more engaging, immersive, and satisfying experiences. Here are some common ways that game designers use player expectations to steer gameplay:

Genre Expectations: Game designers can use genre expectations to establish a baseline of gameplay mechanics and features. For example, a first-person shooter game might include familiar mechanics such as aiming, shooting, and reloading, which are common in the genre. By building on these established mechanics, designers can create more nuanced and complex gameplay experiences that build on what players already know.

Narrative Expectations: Game designers can use narrative expectations to create suspense, surprise, or emotional resonance. By playing with players' expectations, designers can create memorable moments that stick with players long after the game is over. For example, a game might subvert players' expectations by revealing a character to be a villain after previously portraying them as a hero.

Gameplay Expectations: Game designers can use gameplay expectations to create a sense of challenge and progression. By building on what players already know, designers can create more difficult and rewarding gameplay experiences. For example, a game might start with simple puzzles and gradually increase the difficulty over time, challenging players to use their existing skills and knowledge to progress.

Brand Expectations: Game designers can use brand expectations to create a sense of familiarity and continuity. By building on a well-established brand, designers can create more engaging and resonant gameplay experiences. For example, a game that is part of a larger franchise might use

familiar characters, settings, or music to create a sense of continuity and connection to the larger brand.

A drama manager AI is a type of artificial intelligence system that is used in video game design to control the pacing and flow of the game's narrative. The drama manager AI is responsible for dynamically adjusting the game's story and events in response to the player's actions and decisions.

In a video game with a drama manager AI, the system is constantly monitoring the player's progress and making decisions about what events should happen next in order to create a compelling and engaging narrative. The AI may also be responsible for managing the game's difficulty level, adjusting it in real-time to ensure that the player is challenged but not overwhelmed.

Some examples of games that use a drama manager AI include "The Elder Scrolls V: Skyrim" and "Fable III". These games use sophisticated algorithms and decision-making systems to create a dynamic and personalized experience for each player, ensuring that the game's story and events unfold in a way that feels unique and engaging.

Agents

In this context, “agent-based AI is about producing autonomous characters that take in information from the game data, determine what actions to take based on the information, and carry out those actions. for in-game AI, behaviorism is often the way to go. We are not interested in the nature of reality or mind; we want characters that look right. In most cases, this means starting from human behaviors and trying to work out the easiest way to implement them in software. (Millington & Funge 2016) AI agents in video game design are essentially software programs that simulate intelligent behavior in a game. These agents can be designed to perform a variety of tasks, such as controlling non-player characters (NPCs), generating content, and providing assistance to players. The way AI agents work in video game design can vary depending on the specific implementation, but generally follows these steps:

Perception: AI agents first perceive the game world, either through direct sensors or indirectly through game events or other agents. For example, an NPC in a first-person shooter might use sensory inputs like sound and vision to detect the player's presence.

Decision-Making: Based on the perceived state of the game world, the AI agent must decide what actions to take. This decision-making process is often based on a set of rules or algorithms that take into account the agent's objectives, available resources, and current game state. For example, a strategy game AI might decide to build a certain type of structure based on the current game state.

Action: The AI agent then takes action based on the decision-making process. This can include performing physical movements, generating content, or providing feedback to the player. For example, an AI agent controlling an NPC might cause the character to move towards the player or attack them.

Feedback: After taking action, the AI agent must receive feedback from the game world to adjust its perception and decision-making processes. This feedback can be in the form of game events, player interactions, or other agents. For example, an AI agent controlling an enemy might adjust its behavior based on how well the player is performing against it.

Overall, AI agents in video game design work by simulating intelligent behavior through a process of perception, decision-making, action, and feedback. This allows game developers to create more immersive and engaging games, and to provide players with more challenging and dynamic opponents. As AI technology continues to evolve, we can expect to see even more advanced and innovative uses of AI agents in video game design.

Reflex Agents

These agents use a conditional statement to provide the "intelligence". Currently, most actors within games follow reflex systems, to the extent that players can monitor the input-output action pairs of specific actors. Once a pattern has emerged, the human player can modify their strategy sufficiently so that the opponent artificial actor will make a significant loss whilst the human player will make a significant gain. Reflex agents can fall into infinite loops, as there is no concept of context within if-then statements.

Temporal agents can be considered a special sub-group of reflex agents, where actions are carried out after measuring the passing of time. This specific type of agent would be applicable in dynamic and
semi-dynamic game worlds, where time is a factor. Reflex agents, are useful in situations where a high level of complexity is not required by a participant. The more limited the scope of possible actions in a discrete actions game world, for example, the less complexity is required in decision making. In some cases reflex agents might present the best compromise of complexity versus believability.

Model-Based Agents

An agent that monitors the environment and creates a model of it on which to base decisions is called a model-based agent. This type of agent would be best applied to dynamic, real-time games where
constant monitoring of the environment is required on which to base decisions on actions. This would also be highly beneficial in a cooperative game, where although the actions of other actors are independent, they are inter-related, and so a broader monitoring range covering other co-operating actors can be introduced.

Goal-Based Agents

Using a model of the environment, goals can be created and planning carried out to achieve those goals, even within inaccessible game worlds and with other participants. Although the artificially controlled participants will generally have broad goals built in to determine their over-all behaviour (such as "stop the human-player at all costs"), there is still scope within that command to create

subgoals (such as “find the player”) Goal-based agents are also highly beneficial in inaccessible game worlds, as they can change their own sub-goals as the information they are made aware of changes

Utility-Based Agents
A further refinement on the model- and goal-based agent methodologies is the ability to manage multiple goals at the same time, based on the current circumstances. By applying utility theory to define the relative “best” goal in any situation, we have utility-based agents.

Swarm Intelligence

Swarm intelligences have also been used to compute Nash Equilibria. Ant colony modelling provides a strong methodology for actors to explore the game world to complete goals by providing path-finding around obstacles and creating search patterns to achieve their goals - something that has been seen to be lacking in games. Exploring the game world is important in imperfect information worlds, and such group goal finding resulting from ant colony optimization is useful in co-operative game play.

Believable Agents

These agents are expected to behave as a human would in similar situations. Given this is one of the core purposes of developing better AI in games, any agents that are developed should fall into this category, unless (through the narrative) the actor is expected to behave differently. Even then they should be consistent in their behavior which could be construed as providing believability in the behaviour across all actor types. Specifically, player-as-manager games require a great deal of believability given the large number of actors available to the player, and the semi-autonomous nature of those actors. Player-as-actor games will also require a high level of believability, as all the interactions with other actors in the game world must provide a sufficient level of immersion, give

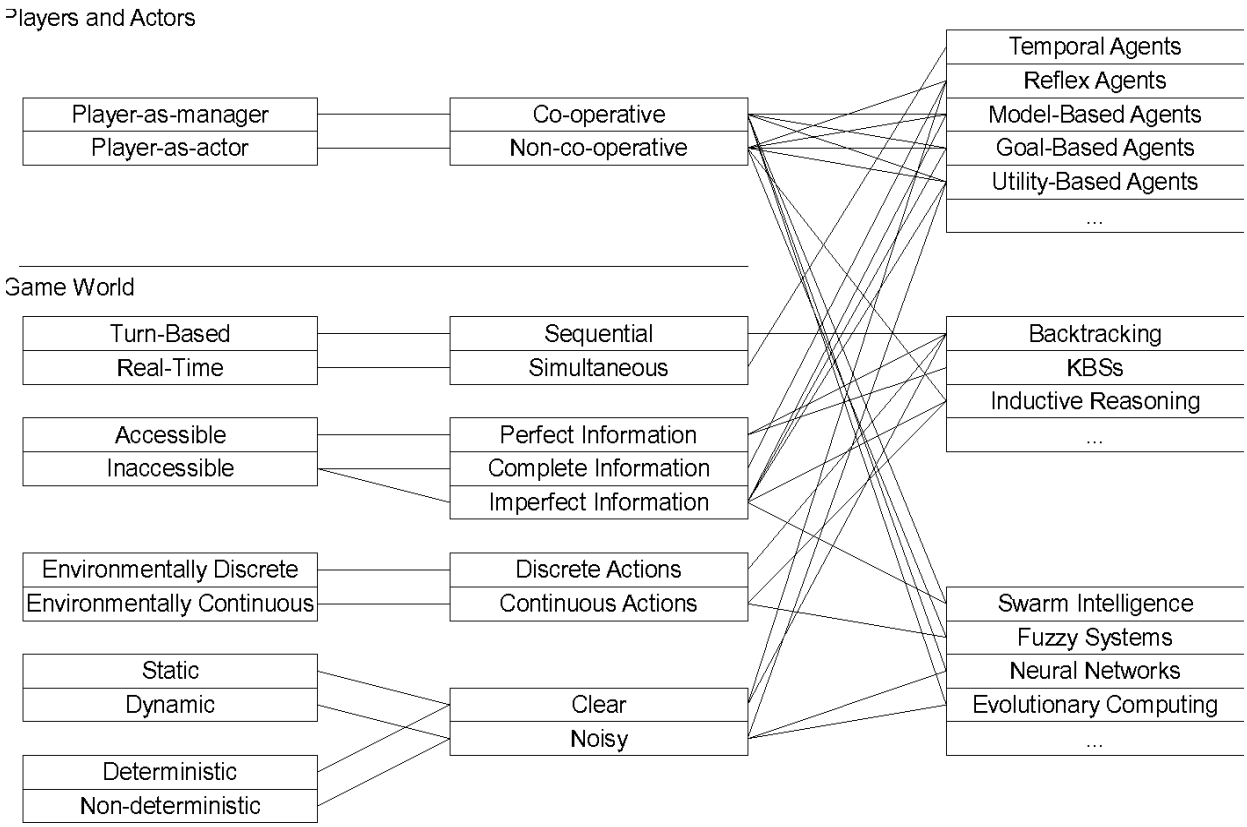


Chart of Video Game Taxonomy (Gunn et al, 2009)

3 ELEMENTS OF AI IN GAMES

Video Game AI is comprised of several different elements of Artificial Intelligence. Most NPCs in games are AI agents. With each game having something like a Director System, which manages the overall game. Agents are aware of each other and communicate with each other, while some have dialogue trees assigned to their agent behavior, there are strictly hidden means of communication between agents as well as different AI sub-systems that manage different elements of the Game. You will encounter AI management systems that even cover Diplomacy in the Game (i.e. Total War) as well as battle AI management and Resource Management AI. As well you can have Finance Managers in the game and Nemisys Systems which manage opponents in the game, some even using Shadow AI to mimic a players tactics, therefore fighting as the gamer plays the game adapting the players strategies into their own counter-measures against the player.

In video game design, a "director system" refers to a set of algorithms or rules that dynamically adjust the gameplay experience based on the player's actions, performance, or other factors. The director system is typically designed to increase the player's engagement and enjoyment by adapting the game's difficulty, pacing, and other elements in response to the player's behavior.

The director system can be used in many different types of games, including first-person shooters, role-playing games, and strategy games. Some examples of how a director system might be used include:

> Difficulty Scaling: The director system can adjust the game's difficulty in real-time based on the player's performance, making the game easier or harder as needed. For example, if the player is struggling with a particular level, the director system might reduce the number of enemies or increase the amount of health the player has to give them a better chance of success.
>
> Pacing: The director system can adjust the pacing of the game by adding or removing obstacles, enemies, or other challenges. For example, if the player is moving too quickly through a level, the director system might add more enemies or traps to slow them down and make the game more challenging.
>
> Storytelling: The director system can adjust the game's story and narrative elements based on the player's choices or performance. For example, if the player is making choices that are leading them down a particular path, the director system might adjust the story to reflect those choices and create a more personalized experience.

As Tommy Thompson describes them:

> So a director is for all intents and purposes any system in a game that makes decisions on in-game settings or behaviors that impact pacing and difficulty and is influenced by what players are doing in either a single or multiplayer context. Now this isn't necessarily an AI system, although you tend to find there is some simple AI formulation in many cases

given it's making some for of intelligent decisions. Ultimately, it boils down to a fairly straight forward process:

- The system records information about the players current activity.
This could include where they are in the world, what activities they are currently doing and how well they're doing it.
- The system then considers what the players should be experiencing at this point in time, such as a temporary increase in difficulty, a dynamic event in the world that perhaps has not occurred for a while, as well as what elements it can change or create right now to inject some new activity for the player to experience. Conversely, it could actually do the opposite and give the player some respite, allowing for you to catch your breath, take a moment to re-assess the situation and what you want to achieve.
- Lastly, it's checking whether the player is playing the game as it is intended.
Quite often a director is useful for creating situations that ensure the player adheres to what the designer intended. As we'll see in a moment, *Left 4 Dead* is a fantastic example of this, given the director deliberately targets players who fail to adhere to the rules that the game communicates to you. (Thompson, *'Director AI for Balancing In-Game Experiences'*, 2021, https://youtu.be/Mnt5zxb8W0Y?t=237)

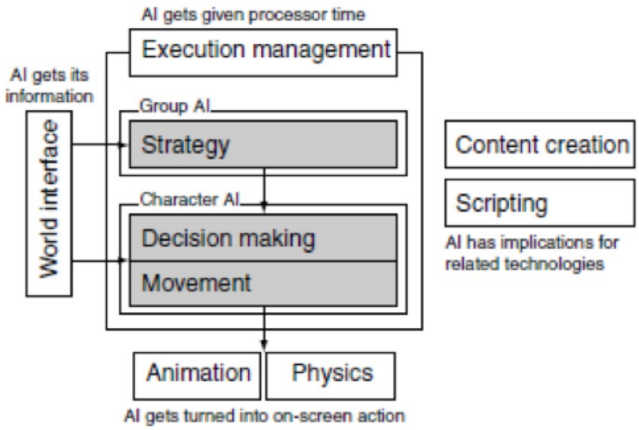


Figure 1.1 The AI model

Some of the common architectures used in games are polling, events, event managers and Sense Management. Each of which has its own domains of control and delegation. The following is based on the work of Millington and Funge (2016).

Polling is the part in the game which keeps track of all the actions, goals and data in the game.

The polling can rapidly grow in processing requirements through sheer numbers, even though each check may be very fast. For checks that need to be made between a character and a lot of similar

sources of information, the time multiplies rapidly. For a level with a 100 characters, 10,000 trajectory checks would be needed to predict any collisions. Because each character is requesting information as it needs it, polling can make it difficult to track where information is passing through the game. Trying to debug a game where information is arriving in many different locations can be challenging. (Millington, Funge 2016)

Polling Stations

There are ways to help polling techniques become more maintainable. A polling station can be used as a central place through which all checks are routed. This can be used to track the requests and responses for debugging. It can also be used to cache data
(Millington, Funge 2016)

Events

we want a central checking system that can notify each character when something important has happened. This is an event passing mechanism. A central algorithm looks for interesting information and tells any bits of code that might benefit from that knowledge when it finds something.
The event mechanism can be used in the siren example. In each frame when the siren is sounding, the checking code passes an event to each character that is within earshot. This approach is used when we want to simulate a character's perception in more detail…. The event mechanism is no faster in principle than polling. Polling has a bad reputation for speed, but in many cases event passing will be just as inefficient. To determine if an event has occurred, checks need to be made. The event mechanism still needs to do the checks, the same as for polling. In many cases, the event mechanism can reduce the effort by doing everybody's checks at once. However, when there is no way to share results, it will take the same time as each character checking for itself. In fact, with its extra message-passing code, the event management approach will be slower.
(Millington, Funge 2016)

EventManagers

Event passing is usually managed by a simple set of routines that checks for events and then processes and dispatches them. Event managers form a centralized mechanism through which all events pass. They keep track of characters' interests (so they only get events that are useful to them) and can queue events over multiple frames to smooth processor use. (Millington, Funge 2016)

An event-based approach to communication is centralized. There is a central checking mechanism, which notifies any number of characters when something interesting occurs. The code that does this is called an event manager.

The event manager consists of four elements:

1. A checking engine (this may be optional)
2. An event queue
3. A registry of event recipients
4. An event dispatcher

The interested characters who want to receive events are often called “listeners” because they are listening for an event to occur. This doesn’t mean that they are only interested in simulated sounds. The events can represent sight, radio communication, specific times (a character goes home at 5 P.M. for example), or any other bit of game data. The checking engine needs to determine if anything has happened that one of its listeners may be interested in. It can simply check all the game states for things that might possibly interest any character, but this may be toomuch work.More efficient checking engines take into consideration the interests of its listeners. (Millington, Funge 2016)

Event Casting

There are two different philosophies for applying event management. You can use a few very general event managers, each sending lots of events to lots of listeners. The listeners are responsible for working out whether or not they are interested in the event. Or, you can use lots of specialized event managers. Each will only have a few listeners, but these listeners are likely to be interested in more of the events it generates. The listeners can still ignore some events, but more will be delivered correctly. The scattergun approach is called broadcasting, and the targeted approach is called narrowcasting. Both approaches solve the problem of working out which agents to send which events. Broadcasting solves the problem by sending them everything and letting them work out what they need. Narrowcasting puts the responsibility on the programmer: the AI needs to be registered with
exactly the right set of relevant event managers. This approach is called broadcasting. A broadcasting event manager sends lots of events to its listeners. Typically, it is used to manage all kinds of events and, therefore, also has lots of listeners. (Millington, Funge 2016)

Inter-Agent Communication

While most of the information that an AI needs comes from the player’s actions and the game environment, games are increasingly featuring characters that cooperate or communicate with each other. A polling station has two purposes. First, it is simply a cache of polling information that can be used by multiple characters. Second, it acts as a go-between from the AI to the game level. Because all requests pass through this one place, they can be more easily monitored and the AI debugged.
(Millington, Funge 2016)

Sense Management

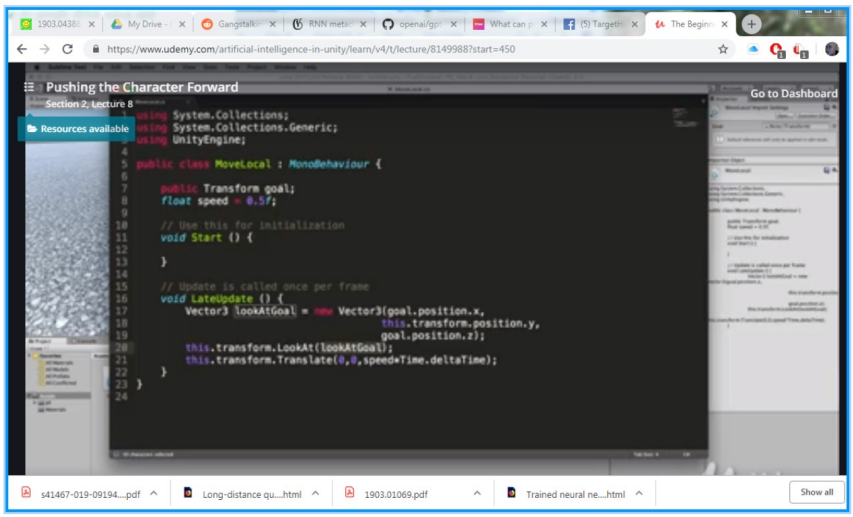
> Up until the mid-1990s, simulating sensory perception was rare (at most, a ray cast check was made to determine if line of sight existed). Since then, increasingly sophisticated models of sensory perception have been developed. In games such as Splinter Cell [Ubisoft Montreal Studios, 2002], Thief: The Dark Project [Looking Glass Studios, Inc., 1998], and Metal Gear Solid [Konami Corporation, 1998], the sensory ability of AI characters forms the basis of the gameplay. Indications are that this trend will continue. AI software used in the film industry (such as Weta's Massive) and military simulation use comprehensive models of perception to drive very sophisticated group behaviors. It seems clear that the sensory revolution will become an integral part of real-time strategy games and platformers, as well as third-person action games.
>
> A more sophisticated approach uses event managers or polling stations to only grant access to the information that a real person in the game environment might know. At the final extreme, there are sense managers distributing information based on a physical simulation of the world.Even in a game with sophisticated sense management, it makes sense to use a blended approach. Internal knowledge is always available, but external knowledge can be accessed in any of the following three ways: direct access to information, notification only of selected information, and perception simulation.
> We will exclusively use an event-based model for our sense management tools. Knowledge from the game state is introduced into the sense manager, and those characters who are capable of perceiving it will be notified. They can then take any appropriate action, such as storing it for later use or acting immediately.
> (Millington, Funge 2016)

Overview of search

Most of this is managed either through Finite State Machines, Markov Chains, Behavior Trees, all of which are outgrowths of what is known as Graph Searching. Graph searching is a means of searching interconnected nodes in a matrix or network. Some early examples of Graph Search are Depth First Search, and Breadth First Search. Where one searches the graph tree horizontally first, as in Breadth First, the other searching through one depth to the next as in Depth First. In the early years of AI at the Stanford Research Institute, which was a CIA contractor that in the early days conducted research in Remote Viewing as well as AI and Robotics, for their robotic research came up with A* search, which examines edge weights and costs to give a heuristic approach to AI search, a heuristic is a rule of thumb, an approximate solution that might work in many situations but is unlikely to work in all.
. The main examination in determining a heuristic approach is how well the particularly path or trajectory of a search result achieves the goal, which is to win the game. Another area that SRI was involved in was developing Planning systems in Simulations, known as STRIPS, which we shall cover in the Goal Oriented Action Planning section.

Moving around

All games utilize the foundational mechanics in Games for moving characters around. Since it is based on statistical learning, you see a usually non-human nature to their movements. For instance, one thing a game character must do to target the opposing player is look at the player, automatically zeroing in on the player. This is known as the LookAt() method in movement libraries and is a fairly universal component of video games. Some Targeted Individuals report that routinely they are targeted through involuntary staring on the part of hypnotized people around them. That no matter how obscure their location that people entering their area LookAt() them.



LookAt() function with in the code of a game (De Byl 2019)

Moving in a game is based on vector mathematics, a 'array' of three elements.
A point is a location in space. A vector is a direction and length. A vector has no predetermined starting point. There is only one point in space that has its coordinates, however vectors with the same values can be anywhere. The x,y,z of the point is a single location in space, the x,y,z of a vector is the length of the vector in each of those dimensions.

Vector Mechanics:

a point plus a vector will result in a point

a vector plus a vector will result in a vector

In games we add vectors to points to move objects around. For example, the movement of characters from one location to another. The vector calculated between one point and another tells us the direction and distance.

If Stevie (the zombie) wants to get from her position to Granny's position then the vector she must travel is granny's position minus Stevie's position. For example, if Stevie was at (10,15, 5) and Granny was at (20, 15, 20) then the vector from Stevie to Granny would be (20, 15, 20) - (10, 15, 5) = (10, 0, 15). If Granny wanted to find her vector to Stevie the equation is reversed.

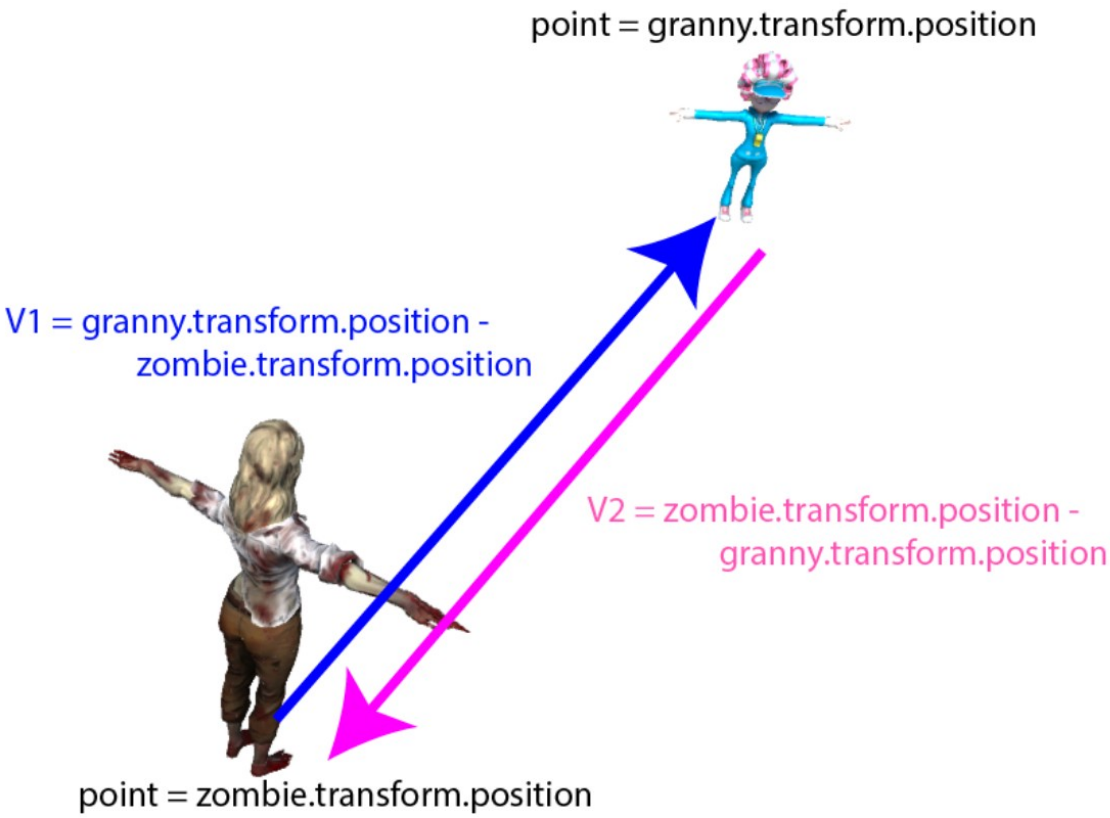


Fig 2. Calculating Vectors Between Points

(De Byl 2019)

Then to move Stevie from her current location to that of Granny you would add the vector (V1) to Stevie's position thus: (10,15, 5) + (10, 0, 15) = (20, 15, 20) which you can check is correct because it's the position of Granny!

Vectors can also be added together to give a total direction and magnitude. The length of a vector is called its magnitude. When the direction toward a character is calculated as we've done in the previous examples, by taking one position away from another, the resulting length of that vector is the distance between the characters.

In games distance between locations is used by decision making AI as well as moving objects around. For example, an NPC might work out the distance to a player before deciding whether to attack or not. In determining the direction in which to travel to get from one location to another you might also require an angle that indicates how much a character needs to turn to be facing that location, otherwise you'll get a character that moves sideways. Once you have calculated the angle between the way the character is facing and the direction it is about to travel you'll be able to program its turning.

The smoothing of a NPCs movement is a product of statistical mechanics. Instead of making jagged or quick movements it will walk in an arc like fashion or take a straight angle approach. This is also noticed in Self-Driving or Vehicle Simulations: cars proceed in smooth accelerations, decelerations, turning is

rounder, etc. All due to the AI used in their systems for managing movements and being statistical based rather then natural or chaotic. Which of course does not give a totally satisfactory simulation of real-time conditions in a real world, but for training purposes it may be good enough if you can account for these unnatural attributes and they do not become ingrained in muscle memory of the soldiers.

To move an agent NPC around, you could rely on older technology such as waypoints: a reference point used for navigation purposes by in-game characters. Most commonly used in strategy games and squad based games. For instance if you wanted to statically block an entrance you would simple give the coordinates to a waypoint near the door and have the NPC proceed to that waypoint. Single-Rail games are a good example of using waypoints. A newer approach is to use Navmeshes: in use since the mid-80s in robotics as meadow maps, became part of video game code around 2000, an abstract data structure used in artificial intelligence applications to aid agents in pathfinding through complicated spaces.

Aside from moving individual NPCs you can also group NPCs to move as flocks or as swarms, both video games and military drones use behaviour trees for complicated actions including swarming.

Some common movement related elements of game design are presented below, again based on the work of (Millington, Funge 2016):

Sight Cone

> a sight cone of around 60° is often used. It takes into account normal eye movement, but effectively blinds the character to the large area of space it can see but is unlikely to pay any attention to. (Millington, Funge 2016)

Movement

> Movement refers to algorithms that turn decisions into some kind of motion. When an enemy character without a gun needs to attack the player in Super Mario Sunshine , it first heads directly for the player. When it is close enough, it can actually do the attacking. The decision to attack is carried out by a set of movement algorithms that home in on the player's location. Only then can the attack animation be played and the player's health be depleted. Movement algorithms can be more complex than simply homing in. the AI needs information from the game to make sensible decisions. This is sometimes called "perception" (especially in academic AI): working out what information the character knows. In practice, it is much broader than just simulating what each character can see or hear, but includes all interfaces between the game world and the AI. This world interfacing is often a large proportion of the work done by an AI programmer, and in our experience it is the largest proportion of the AI debugging effort. (Millington, Funge 2016)

Steering Behavior-

Steering behaviors is the name given by Craig Reynolds to his movement algorithms; they are not kinematic, but dynamic. Dynamic movement takes account of the current motion of the character. A dynamic algorithm typically needs to know the current velocities of the character as well as its position. A dynamic algorithm outputs forces or accelerations with the aim of changing the velocity of the character. Craig Reynolds also invented the flocking algorithm used in countless films and games to animate flocks of birds or herds of other animals….Because flocking is the most famous steering behavior, all steering (in fact, all movement) algorithms are sometimes wrongly called "flocking." (Millington, Funge 2016)

Characters as Points

Although a character usually consists of a three-dimensional (3D) model that occupies some space in the game world, many movement algorithms assume that the character can be treated as a single point. Collision detection, obstacle avoidance, and some other algorithms use the size of the character to influence their results, but movement itself assumes the character is at a single point. This is a process similar to that used by physics programmers who treat objects in the game as a "rigid body" located at its center of mass. Collision detection and other forces can be applied to anywhere on the object, but the algorithm that determines the movement of the object converts them so it can deal only with the center of mass. (Millington, Funge 2016)

Seek

A kinematic seek behavior takes as input the character's and its target's static data. It calculates the direction from the character to the target and requests a velocity along this line. The orientation values are typically ignored, although we can use the getNewOrientation function above to face in the direction we are moving. (Millington, Funge 2016)

Steering Behaviors

Steering behaviors extend the movement algorithms in the previous section by adding velocity and rotation. They are gaining larger acceptance in PC and console game development. In some genres (such as driving games) they are dominant; in other genres they are only just beginning to see serious use.

Obstacle avoidance behaviors take a representation of the collision geometry of the world. It is also possible to specify a path as the target for a path following behavior. In these behaviors some processing is needed to summarize the set of targets into something that the behavior can react to. This may involve averaging properties of the whole set (to find and aim for their center of mass, for example) (Millington, Funge 2016)

Variable Matching

The simplest family of steering behaviors operates by variable matching: they try to match one or more of the elements of the character's kinematic to a single target kinematic. (Millington, Funge 2016)

Seek and Flee

Seek tries to match the position of the character with the position of the target. Exactly as for the kinematic seek algorithm, it finds the direction to the target and heads toward it as fast as possible. Because the steering output is now an acceleration, it will accelerate as much as possible. Seek will always move toward its goal with the greatest possible acceleration (Millington, Funge 2016)

Velocity Matching

So far we have looked at behaviors that try to match position with a target.We could do the same with velocity, but on its own this behavior is seldom useful. It could be used to make a character mimic the motion of a target… [useful in psychological operations] (Millington, Funge 2016)

Face

The face behavior makes a character look at its target. It delegates to the align behavior to perform the rotation but calculates the target orientation first. Wander, the wander behavior controls a character moving aimlessly about (Millington, Funge 2016)

Path Following

So far we've seen behaviors that take a single target or no target at all. Path following is a steering behavior that takes a whole path as a target. A character with path following behavior should move along the path in one direction. Path following, as it is usually implemented, is a delegated behavior. It calculates the position of a target based on the current character location and the shape of the path. It then hands its target off to seek. There is no need to use arrive, because the target should always be moving along the path.We shouldn't need to worry about the character catching up with it. The target position is calculated in two stages. First, the current character position is mapped to the nearest point along the path. This may be a complex process, especially if the path is curved or made up of many line segments. Second, a target is selected which is further along the path than the mapped point by a fixed distance. (Millington, Funge 2016)

Separation

The separation behavior is common in crowd simulations, where a number of characters are all heading in roughly the same direction. It acts to keep the characters from getting too close and

being crowded. (Millington, Funge 2016)

Attraction

Using the inverse square law, we can set a negative valued constant of decay and get an attractive force. The character will be attracted to others within its radius, but this is rarely useful. Some developers have experimented with having lots of attractors and repulsors in their level and having character movement mostly controlled by these. Characters are attracted to their goals and repelled from obstacles, for example. Despite being ostensibly simple, this approach is full oftraps for the unwary. (Millington, Funge 2016)

Collision Avoidance

In urban areas, it is common to have large numbers of characters moving around the same space. These characters have trajectories that cross each other, and they need to avoid constant collisions with other moving characters. A simple approach is to use a variation of the evade or separation behavior, which only engages if the target is within a cone in front of the character. (Millington, Funge 2016)

Obstacle and Wall Avoidance

The collision avoidance behavior assumes that targets are spherical. It is interested in avoiding getting too close to the center point of the target. This can also be applied to any obstacle in the game that is easily represented by a bounding sphere. Crates, barrels, and small objects can be avoided simply this way. The obstacle and wall avoidance behavior uses a different approach to avoiding collisions. The moving character casts one or more rays out in the direction of its motion. If these rays collide with an obstacle, then a target is created that will avoid the collision, and the character does a basic seek on this target. Typically, the rays are not infinite. They extend a short distance ahead of the character (usually a distance corresponding to a few seconds of movement). Decision trees, state machines, and blackboard architectures have all been used to control steering behaviours. (Millington, Funge 2016)

Targeters

Targeters generate the top-level goal for a character. There can be several targets: a positional target, an orientation target, a velocity target, and a rotation target. We call each of these elements a channel of the goal (e.g., position channel, velocity channel). All goals in the algorithm can have any or all of these channels specified. An unspecified channel is simply a “don’t care.” Individual channels can be provided by different behaviors (a chase-the-enemy targeter may generate the positional target, while a look-toward targeter may provide an orientation target), or multiple channels can be requested by a single targeter. (Millington, Funge 2016)

Decomposers

Decomposers are used to split the overall goal into manageable sub-goals that can be more easily achieved. (Millington, Funge 2016)

Constraints

Constraints limit the ability of a character to achieve its goal or sub-goal. They detect if moving toward the current sub-goal is likely to violate the constraint, and if so, they suggest a way to avoid it. Constraints tend to represent obstacles: moving obstacles like characters or static obstacles like walls. (Millington, Funge 2016)

The Actuator

Unlike each of the other stages of the pipeline, there is only one actuator per character. The actuator's job is to determine how the character will go about achieving its current sub-goal. Given a sub-goal and its internal knowledge about the physical capabilities of the character, it returns a path indicating how the character will move to the goal. The actuator also determines which channels of the sub-goal take priority and whether any should be ignored. (Millington, Funge 2016)

Coordinated Movement

Games increasingly require groups of characters to move in a coordinated manner. Coordinated motion can occur at two levels. The individuals can make decisions that compliment each other, making their movements appear coordinated. Or they can make a decision as a whole and move in a prescribed, coordinated group. (Millington, Funge 2016)

Emergent Formations

Emergent formations provide a different solution to scalability. Each character has its own steering system using the arrive behavior. The characters select their target based on the position of other characters in the group. (Millington, Funge 2016)

4 AI DECISION MAKING IN GAMES

Autonomous decision making in video game AI agents typically involves the use of algorithms that enable the agents to make decisions based on various factors, such as the state of the game world, the behavior of other agents, and the objectives that the agent is trying to achieve. One common approach is to use a rule-based system, where the AI agent is given a set of rules to follow based on the game's mechanics and objectives. For example, an AI agent in a first-person shooter game may have rules such as "shoot at the player when in sight" and "take cover when health is low." These rules are typically programmed by game developers and can be modified based on the desired behavior of the AI agent. Another approach is to use machine learning algorithms, where the AI agent learns from experience and adapts its behavior over time. This approach is often used in more complex games where the behavior of other agents is unpredictable, and the game world is constantly changing. Machine learning algorithms can be used to train the AI agent to recognize patterns and make decisions based on them, such as predicting the behavior of other agents or identifying the best route to reach an objective. In both approaches, the AI agent typically has access to a set of game state variables, such as the position of other agents, the status of its health and resources, and the location of objectives. The agent processes this information and uses it to make decisions based on its programmed rules or learned behavior. These decisions can then be used to control the actions of the AI agent, such as moving, shooting, or interacting with objects in the game world.

Decision making in games is comprised of several different methodologies to achieve desired AI Agent optimal performance. Each game has different methods that could be used, along with some that are seen in more then one genre of game, such as Goal Oriented Action Planning and Behavior Trees. The following concise overview of Decision Making is from Millington & Funge 2016):

> In reality, decision making is typically a small part of the effort needed to build great game AI. Most games use very simple decision making systems: state machines and decision trees.
>
> Rule-based systems are rarer, but important. The character processes a set of information that it uses to generate an action that it wants to carry out. The input to the decision making system is the knowledge that a character possesses, and the output is an action request. The knowledge can be further broken down into external and internal knowledge. External knowledge is the information that a character knows about the game environment around it: the position of other characters, the layout of the level, whether a switch has been thrown, the direction that a noise is coming from, and so on. Internal knowledge is information about the character's internal state or thought processes: its health, its ultimate goals, what it was doing a couple of seconds ago, and so on. Typically, the same external knowledge can drive any of the algorithms in this chapter, whereas the algorithms themselves control what kinds of internal knowledge can be used (although they don't constrain what that knowledge represents, in game terms).
>
> Actions, correspondingly, can have two components: they can request an action that will change the external state of the character (such as throwing a switch, firing a weapon, moving into a room) or actions that only affect the internal state. Changes to the internal state are less obvious in game applications but are significant in some decision making algorithms. They might correspond to

changing the character's opinion of the player, changing its emotional state, or changing its ultimate goal. Again, algorithms will typically have the internal actions as part of their makeup, while external actions can be generated in a form that is identical for each algorithm. The format and quantity of the knowledge depend on the requirements of the game. Knowledge representation is intrinsically linked with most decision making algorithms. It is difficult to be completely general with knowledge representation… Actions, on the other hand, can be treated more consistently.

Decision Trees in Video Games

A decision tree is made up of connected decision points. The tree has a starting decision, its root. For each decision, starting from the root, one of a set of ongoing options is chosen.

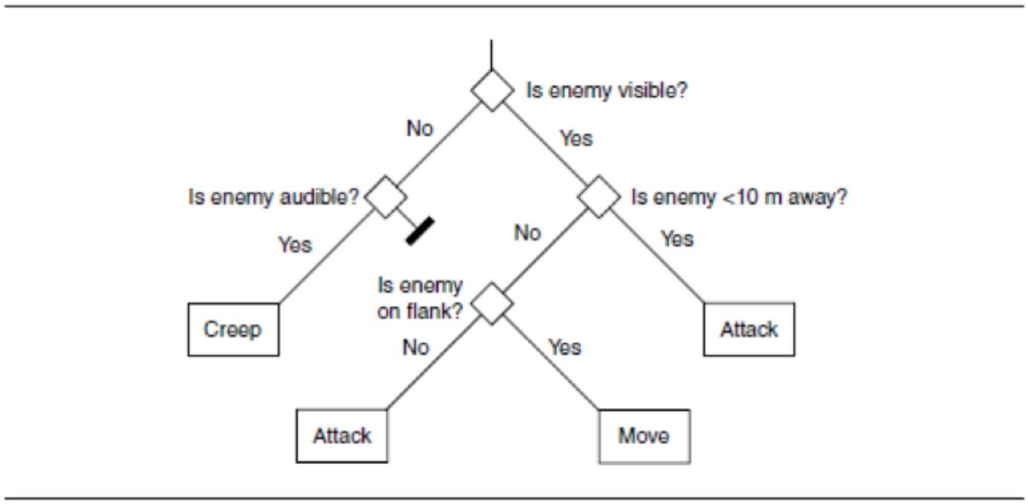


Figure 5.3 A decision tree

(Millington, Funge 2016)

Finite state machines (FSM)

In Simulation development, we have what is known as the state, not to be confused with quantum state.

State machines are the technique most often used for this kind of decision making and, along with scripting...make up the vast majority of decision making systems used in current games. State machines take account of both the world around them (like decision trees) and their internal makeup (their state). States are connected together by transitions. Each transition leads from one state to

> another, the target state, and each has a set of associated conditions. If the game determines that the conditions of a transition are met, then the character changes state to the transition's target state. (Millington, Funge 2016)

So how can we generate complex behaviors out of a fairly simple codebase. Complex behaviors were first generated in video games (perhaps, most famously in *Batman Arkham*) as the first step in the evolution of ever increasing complex behaviors from NPCs.

> A Finite State Machine is a model of computation, i.e. a conceptual tool to design systems. It processes a sequence of inputs that changes the state of the system. When all the input is processed, we observe the system's final state to determine whether the input sequence was accepted or not. (Sanatan 2019)

Software Engineer Marcus Sanatan provides a very good overview of FSM:

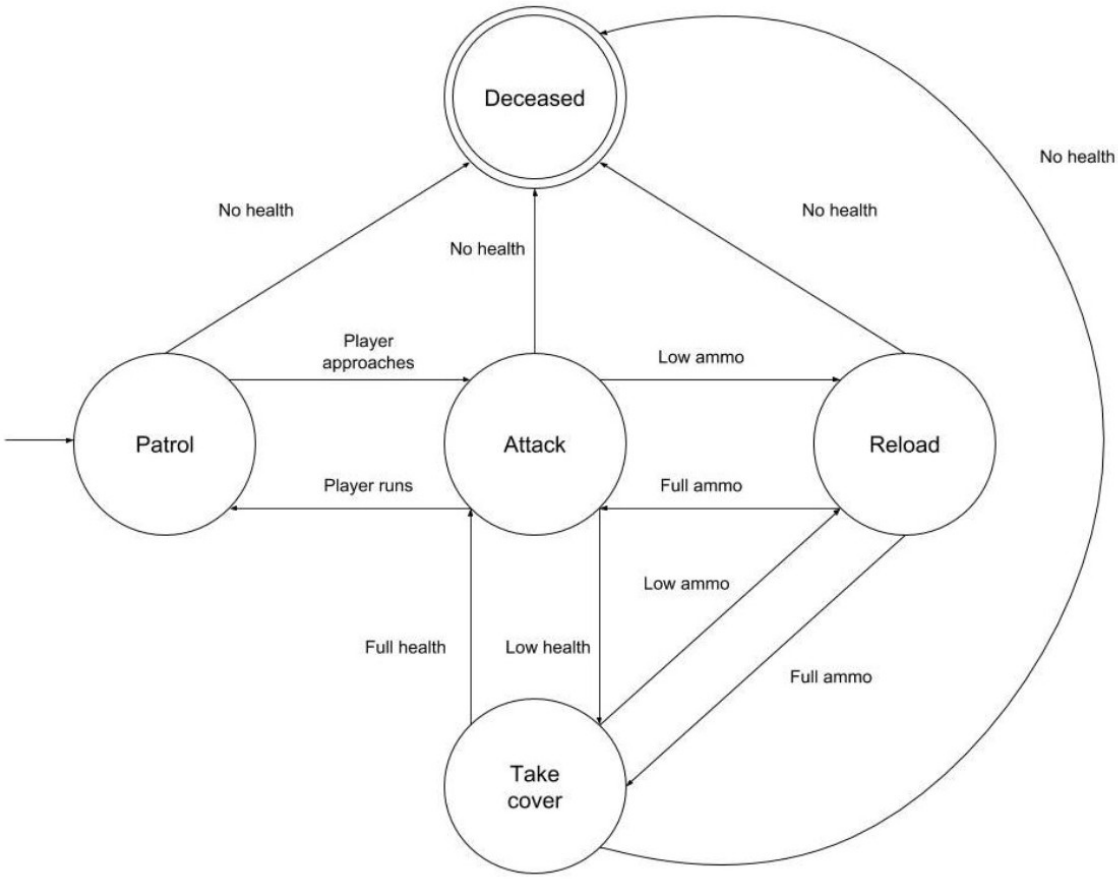
> Enemy AI
>
> > Finite State Machines allows us to map the flow of actions in a game's computer-controlled players. Let's say we were making an action game where guards patrol an area of the map. We can have a Finite State Machine with the following properties:
> >
> > States:
> > For our simplistic shooter we can have: Patrol, Attack, Reload, Take Cover, and Deceased.
> > Initial State: As it's a guard, the initial state would be Patrol.
> > Accepting States: An enemy bot can no longer accept input when it's dead, so our Deceased state will be our accepting one.
> > Alphabet: For simplicity, we can use string constants to represent a world state: Player approaches, Player runs, Full health, Low health, No health, Full ammo, and Low ammo.
> > Transitions: As this model is a bit more complex than traffic lights, we can separate the transitions by examining one state at a time:
> >
> > Patrol
> > If a player approaches, go to the Attack state.
> > If we run out of health, go to the Deceased state.
> >
> > Attack
> > If ammo is low, go to the Reload state.
> > If health is low, go to the Take Cover state.
> > If the player escapes, go to the Patrol state.

If we run out of health, go to the Deceased state.

Reload
If ammo is full, go to the Attack state.
If health is low, go to the Take Cover state.
If we run out of health, go to the Deceased state.

Take Cover
If health is full, go to the Attack state.
If ammo is low, go to the Reload state.
If we run out of health, go to the Deceased state.

This Finite State Machine can be drawn as follows:



(Sanatan 2019)

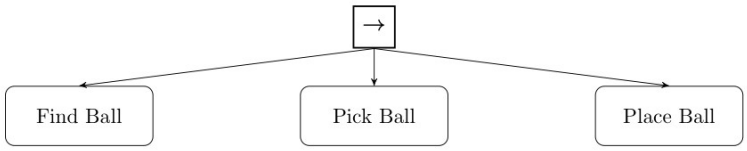
Behavior Trees

A behavior tree is a data structure commonly used in video game design to control the behavior of non-player characters (NPCs) in a game. It is used to represent a decision-making process for an AI-controlled character and defines how the character should react to different situations and stimuli in the game world. A behavior tree consists of a hierarchy of nodes that represent decisions and actions. The root node of the tree represents the overarching goal or behavior of the NPC, and the branches of the tree represent different paths the NPC can take based on different conditions or events in the game.

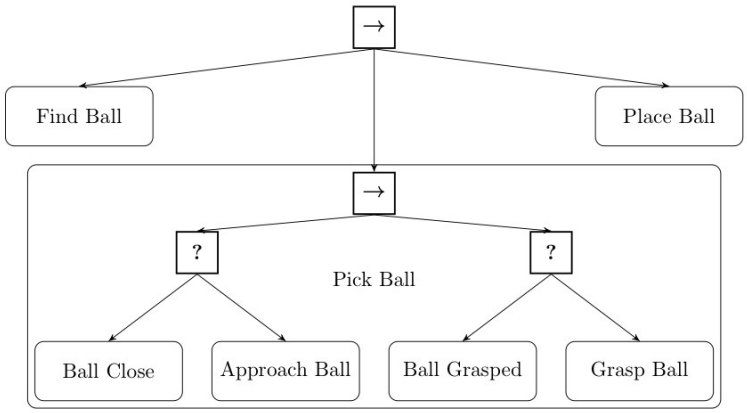
Each node in the tree can be one of two types: a control node, which makes decisions and selects the next action to take, or a task node, which represents an action to be taken. For example, a control node might decide whether the NPC should attack the player or retreat, while a task node might specify the exact animation or action to be taken. Behavior trees are useful in game design because they allow for the creation of complex and flexible AI behavior, while still being easy to understand and debug. They also allow for the reuse of AI behavior across different NPCs and can be modified and updated during development to improve the AI's performance and realism.

The downfall of many FSMs is their simplicity, though it is true you can generate many complex patterns from a simple 0,1 cellular automata, out of which you may even create chaos. The desire for having more advanced and nuanced, which is to say 'believable' NPCs led to the creation of what was known as Behavior Trees.

> A Behavior Tree (BT) is a way to structure the switching between different tasks in an autonomous agent, such as a robot or a virtual entity in a computer game. An example of a BT performing a pick and place task can be seen in Fig. 1.1a [see below]. As will be explained, BTs are a very efficient way of creating complex systems that are both modular and reactive. These properties are crucial in many applications, which has led to the spread of BT from computer game programming to many branches of AI and Robotics.



(a) A high level BT carrying out a task consisting of first finding, then picking and finally placing a ball.



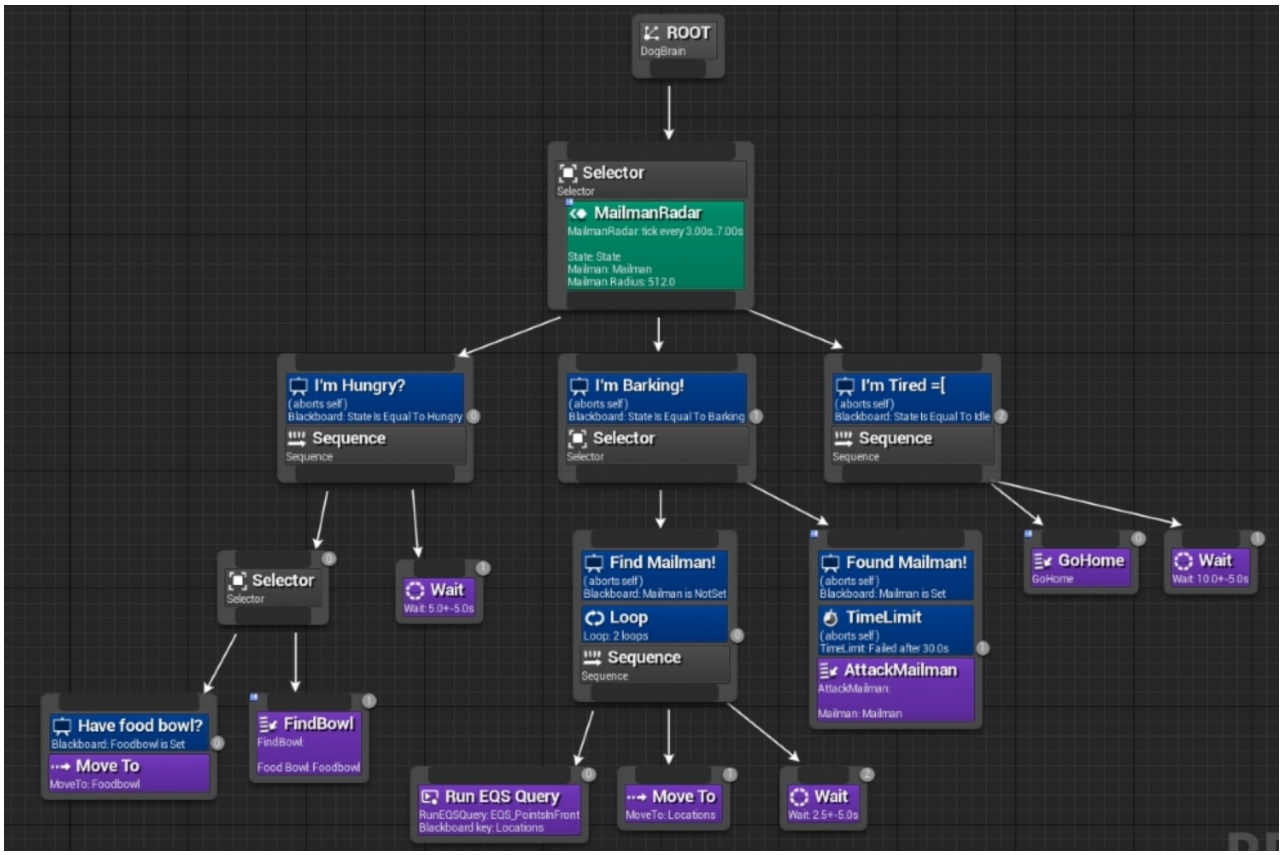
(b) The Action Pick Ball from the BT in Fig. 1.1a is expanded into a sub-BT. The Ball is approached until it is considered close, and then the Action grasp is executed until the ball is securely grasped.

Fig. 1.1: Illustrations of a BT carrying out a pick and place task with different degrees of detail. The execution of a BT will be described in Section 1.3.

Behavior Trees are Event-Driven:

All NPC's (non-player characters) continue to execute a specific behavior until an event (by friend or foe) triggers a change-up. The Director system or other management AI must monitor the 'environment' of the game, then passes on sensory feedback to the AI Agent NPC to change its behavior given what is happening in the environment. It keeps track of changes in the game, depending on the frame rate of the game usually 60fps, via observers or polling agents, whose job it is to keep track of the game and update the Director AI.

As an example of different things an NPC can do with Behavior Trees is that of Squad Triggers when playing in a group of NPCs with the gamer, or enemy AI NPC groups or squads. Squad Triggers - volumes placed in the unreal engine editor that trigger specific responses in non-player characters. When players walk into a AI NPC what behaviors to execute are triggered and specific voice lines to occur and allow the story to play out. Preferred locations and allowed positions. It forces the main game characters to adhere to the story and allow it to be played out in the game. (Steers the game to its desired end or goal) Enemy NPCs are able to shout and chatter, call outs, move position, reload, panic/flee, etc. Ally AI and Enemy AI part of one AI system to move the story. Akin to behaviouralism's notion of positive and negative feedback.



A flow chart for a Behavior Tree from a video game

Again Millington and Funge give more information on the use of Behavior Trees:

> Behavior trees have become a popular tool for creating AI characters. Halo 2 [Bungie Software, 2004] was one of the first high-profile games for which the use of behavior trees was described in detail and since then many more games have followed suit.
> They are a synthesis of a number of techniques that have been around in AI for a while: Hierarchical State Machines, Scheduling, Planning, and Action Execution. Their strength comes from their ability to interleave these concerns in a way that is easy to understand and easy for non-programmers to create. Despite their growing ubiquity, however, there are things that are difficult to do well in behavior trees, and they aren't always a good solution for decision making.
>
> Behavior trees have a lot in common with Hierarchical State Machines but, instead of a state, the main building block of a behavior tree is a task. A task can be something as simple as looking up the value of a variable in the game state, or executing an animation.
>
> Tasks are composed into sub-trees to represent more complex actions. In turn, these complex actions can again be composed into higher level behaviors. It is this composability that gives

behavior trees their power. Because all tasks have a common interface and are largely self-contained, they can be easily built up into hierarchies (i.e., behavior trees) without having to worry about the details of how each sub-task in the hierarchy is implemented. (Millington, Funge 2016)

Types of Task

Tasks in a behavior tree all have the same basic structure. They are given some CPU time to do their thing, and when they are ready they return with a status code indicating either success or failure (a Boolean value would suffice at this stage). Some developers use a larger set of return values, including an error status, when something unexpected went wrong, or a need more time status for integration with a scheduling system. three kinds of tasks: Conditions, Actions, and Composites. (Millington, Funge 2016)

Behavior Trees and Reactive Planning

Behavior trees implement a very simple form of planning, sometimes called reactive planning. Selectors allow the character to try things, and fall back to other behaviors if they fail. This isn't a very sophisticated form of planning: the only way characters can think ahead is if you manually add the correct conditions to their behavior tree. Nevertheless, even this rudimentary planning can give a good boost to the believability of your characters.

The behavior tree represents all possible Actions that your character can take. The route from the top level to each leaf represents one course of action,2 and the behavior tree algorithm searches among those courses of action in a left-to-right manner. In other words, it performs a depth-first search.
In the context of a behavior tree, a Decorator is a type of task that has one single child task and modifies its behavior in some way. You could think of it like a Composite task with a single child. Unlike the handful of Composite tasks we'll meet, however, there are many different types of useful Decorators.

One simple and very common category of Decorators makes a decision whether to allow their child behavior to run or not (they are sometimes called "filters"). If they allow the child behavior to run, then whatever status code it returns is used as the result of the filter. If they don't allow the child behavior to run, then they normally return in failure, so a Selector can choose an alternative action. (Millington, Funge 2016)

External Datastore in Behaviour Trees

The most sensible approach is to decouple the data that behaviors need from the tasks themselves. We will do this by using an external data store for all the data that the behavior tree needs. We'll call this data store a blackboard. For now it is simply important to know that the blackboard can

store any kind of data and that interested tasks can query it for the data they need. Using this external blackboard, we can write tasks that are still independent of one another but can communicate when needed. (Millington, Funge 2016)

Finally, a code sample of programming a BT in C# for the Unity Game Engine:
using UnityEngine;

```
public class BehaviorTree : MonoBehaviour
{
    public enum TaskResult
    {
        Success,
        Failure,
        Running
    }

    public abstract class Task
    {
        public abstract TaskResult Run();
    }

    public class Sequence : Task
    {
        public Task[] tasks;

        public override TaskResult Run()
        {
            foreach (Task task in tasks)
            {
                TaskResult result = task.Run();
                if (result != TaskResult.Success)
                {
                    return result;
                }
            }
            return TaskResult.Success;
        }
    }

    public class Selector : Task
    {
        public Task[] tasks;

        public override TaskResult Run()
        {
            foreach (Task task in tasks)
            {
                TaskResult result = task.Run();
                if (result != TaskResult.Failure)
                {
                    return result;
                }
            }
            return TaskResult.Failure;
```

```
    }
}

public class Action : Task
{
    public delegate TaskResult ActionDelegate();

    private ActionDelegate action;

    public Action(ActionDelegate action)
    {
        this.action = action;
    }

    public override TaskResult Run()
    {
        return action();
    }
}

// Example usage:
private Task rootTask;

private void Start()
{
    // Create the behavior tree
    rootTask = new Selector
    {
        tasks = new Task[]
        {
            new Sequence
            {
                tasks = new Task[]
                {
                    new Action(() =>
                    {
                        Debug.Log("Performing task 1");
                        return TaskResult.Success;
                    }),
                    new Action(() =>
                    {
                        Debug.Log("Performing task 2");
                        return TaskResult.Success;
                    })
                }
            },
            new Action(() =>
            {
                Debug.Log("Performing task 3");
                return TaskResult.Failure;
            }),
            new Action(() =>
            {
                Debug.Log("Performing task 4");
                return TaskResult.Success;
            })
        }
    };
```

```
    }

    private void Update()
    {
        // Run the behavior tree
        TaskResult result = rootTask.Run();
        if (result == TaskResult.Success)
        {
            Debug.Log("Behavior tree succeeded!");
        }
        else if (result == TaskResult.Failure)
        {
            Debug.Log("Behavior tree failed!");
        }
        else if (result == TaskResult.Running)
        {
            Debug.Log("Behavior tree is still running.");
        }
    }
}
```

This code defines a simple behavior tree that consists of three types of tasks: Sequence, Selector, and Action. A Sequence task runs each of its child tasks in order until one fails or all succeed. A Selector task runs each of its child tasks in order until one succeeds or all fail. An Action task represents a single atomic action that the AI agent can take. To use this behavior tree in a game, you would need to modify the Action tasks to perform actual game-related actions, such as moving the player character or firing a weapon. You would also need to modify the Selector and Sequence tasks to control the flow of the behavior tree based on the game state and player actions. This code provides a starting point for implementing a behavior tree in Unity, but it is by no means a complete solution.

GOAP

With the ever-increasing need for complexity and entertainment in the Game industry a further development in planning, behaviors and goals was that of the creation of Goal Oriented Action Planning (GOAP). Prof. Tommy Thompson has provided an informative introduction to GOAP. The early predecessor to GOAP, according to its innovator in the Game Industry- Jeff Orkin (MIT), was the STRIPS (Stanford Research Institute Problem Solver) on which GOAP was based. Developed at SRI in 1971 along with A* search, it is recounted by its developers, Nilsson and Fikes:

> "STRIPs is often cited as providing a seminal framework for attacking the 'classical planning problem' in which the world is regarded as being in a static state and is transformable to another static state only by a single agent performing any of a given set of actions. The planning problem is then to find a sequence of agent actions that will transform a given initial world state into any of a set of given goal states. For many years, automatic planning research was focused on that simple state-space problem formulation and was frequently based on the representation framework and reasoning methods developed in the STRIPS system." (Fikes, 2, 1993)

A* Search and STRIPS would work hand-in-hand to find optimal paths. Automated Planning: is a form of AI that is focused on long-term and abstract decision making. Decisions are modeled at a high level, using

methods based on STRIPS. The STRIPS plan would cause the transitions between states, as the FSM needed to shift state several times in order to execute all actions within a plan.

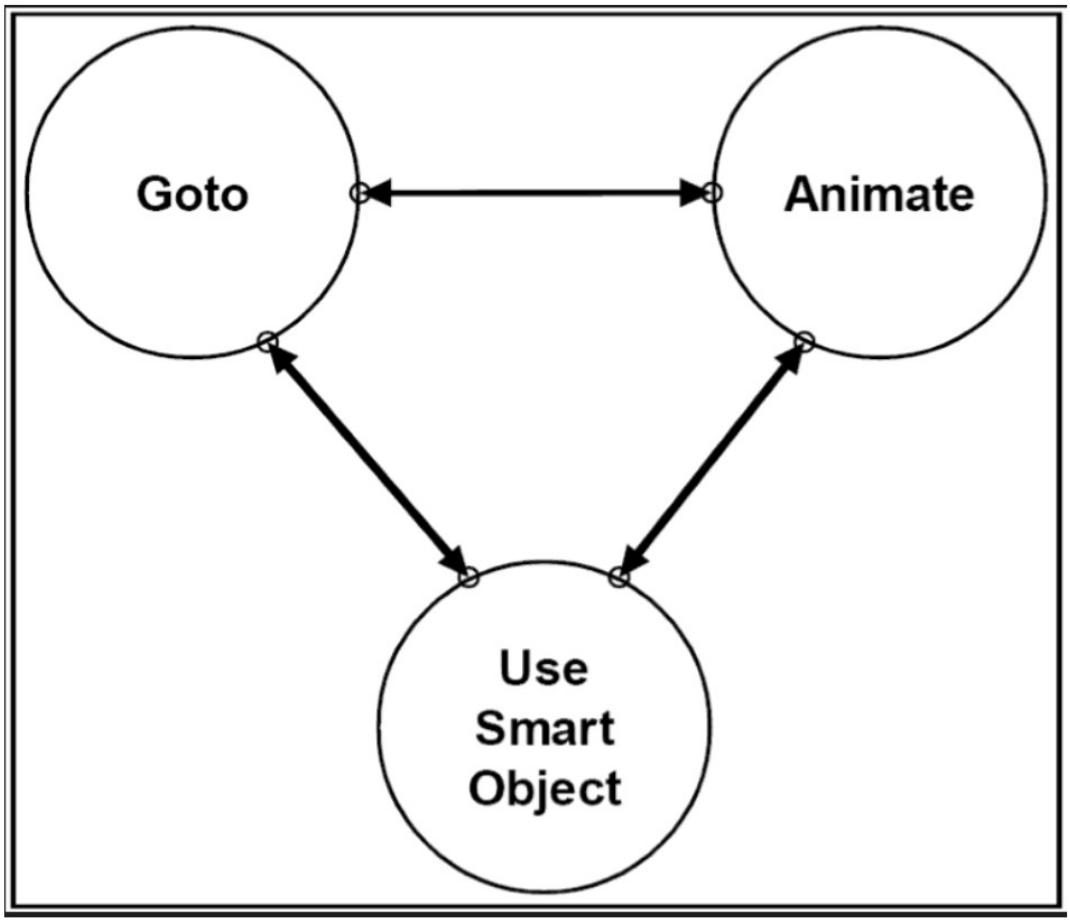
Related to this project was the development at SRI in the 70s of 'Shackey the Robot' of which A* developed [cf. STRIPS, a retrospective, Artificial Intelligence 59 (1993) 227-32 Fikes and Nilsson] A* developed by Hart also worked on region-finding scene analysis programs (Duda and Hart, R.O. Duda and P.E. Hart, Experiments in scene analysis, Tech. Note 20, Artificial Intelligence Center, SRI International, Menlo Park, CA, 1970). Thus having 'understood' the scene new plans could be created to address goals. In video games this was first delivered in the title '*F.E.A.R.*' A first-person shooter (FPS). Thompson gives a short overview of GOAP:

> One of the key issues that we need to consider when developing intelligent and engaging NPCs for games is their ability to build some kind of strategy. Traditionally, an NPC implementation will rely on finite state machines or a similar formalism. These are really useful for a number of practical reasons, notably: We can model explicitly (and often visually) the sort of behaviors we expect our NPC to exhibit. We can often express explicit rules or conditions for when certain changes happen to the NPC state. We can be confident that any behavior the NPC exhibits can be 'debugged', since their procedural nature often exhibit the Markov property: meaning we can always infer how a state of the bot has arisen and also what subsequent states it will find itself in. However the problem that emerges here is that behaviors will neither be deliberative, nor emergent. Any bot that uses these approaches is tied to the specific behaviors that the designers had intended. In order for the more interesting NPCs in F.E.A.R. to work, they need to think long-term. Rather, they need to plan.
>
> Planning (and scheduling) is a substantial area of Artificial Intelligence research, with research dating back as far as the 1960's. As my game AI students will soon learn in more detail, it is an abstract approach to problem solving; reliant upon symbolic representations of state conditions, actions and their influence. Often planning problems (and the systems that solve them) distance themselves from the logistics of how an action is performed. Instead they focus on what needs done when. For those interested, I would encourage reading which provides a strong introduction to planning and scheduling systems.
>
> The NPCs in F.E.A.R. are not merely reactive in nature, they plan to resolve immediate threats by utilizing the environment to full effect. The NPCs in F.E.A.R. are not merely reactive in nature, they plan to resolve immediate threats by utilizing the environment to full effect. One benefit of a planning system is that we can build a number of actions that show how to achieve certain effects in the world state. These actions can dictate who can make these effects come to pass, as well as what facts are true before they can execute, often known as preconditions. In essence, we decouple the goals of an AI from one specific solution. We can provide a variety of means by which goals can be achieved and allow the planner to search for the best solution.
>
> However, at the time F.E.A.R. was developed, planning was not common in commercial games. Planning systems are often applied in real-world problems that require intelligent sequences of

actions across larger periods of time. These problems are often very rich in detail – examples include power station management and control of autonomous vehicles underwater or space – and rely on the optimizations that planning systems make (which I will discuss another time) as well as time. Planning systems are often computationally expensive. While we are looking at time-frames of a couple of seconds when dealing with plans of 20-30 actions in games, this is still dedicating a fair chunk of overall resource to the planner; thus ignoring the issues that a game running in-engine at 30-60 frames per second may face. Put simply, you must find a way to optimize this approach should you wish to implement planning systems, lest you suffer a dip in the games performance at runtime.

G.O.A.P.: Goal Oriented Action Planning

The approach taken by Monolith... was known as Goal Oriented Action Planning. The implementation attempted to reduce the potential number of unique states that the planning system would need to manage by generalizing the state space using a Finite State Machine.



The three states of the GOAP system in FEAR.

As shown in the figure above, the behaviour of the agent is distilled into three core states within the FSM:

Goto – It is assumed that the bot is moving towards a particular physical location. These locations are often nodes that have been denoted to permit some actions to take place when near them.
Animate – each character in the game has some basic animations that need to be executed in order for it to maintain some level of emergence with the player. So this state enforces that bots will run animations that have context within the game world. These can be peeking out from cover, opening fire or throwing a grenade.
Use Smart Object – this is essentially an animation node. The only difference is the animation is happening in the context of that node. Examples of this can be jumping over a railing or flipping a table on its side to provide cover.

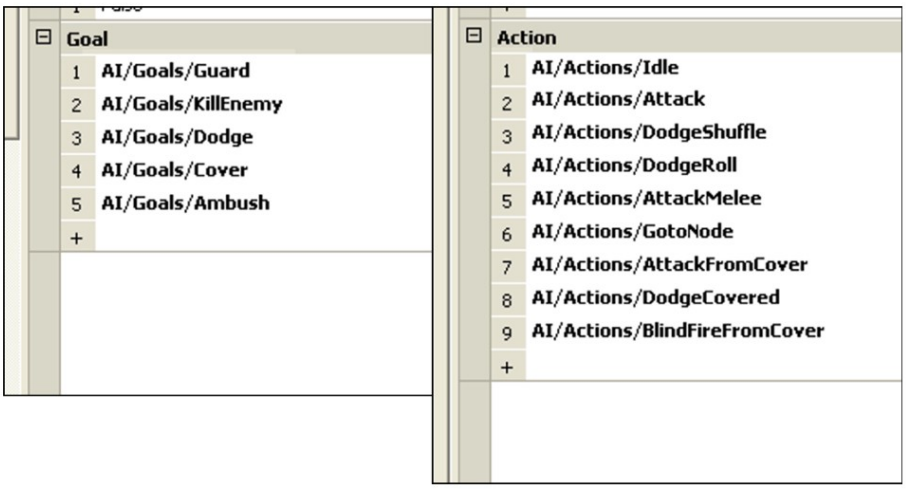
That's it! The entire FSM for the NPCs in the game is three states. Note that these states are very abstract; we do not know what locations are being visited, the animations played and the smart objects used. This is because a search is being conducted that determines how the FSM is navigated.

Which state are these NPCs currently in now?

Bear in mind we traditionally utilize events in order to move from one state to another in a FSM. These events are typically the result of a sensor determining whether some boolean flag is now true, or an 'oracle' system forcing the bot to change the state. This then results in a change of behavior. However, in *F.E.A.R.*, each NPC uses sensor data to determine a relevant goal and then conduct a plan that will navigate the FSM (using grounded values) that will achieve that goal. (Thompson, 2018)

Versatility:

- each agent could be given different goals and actions that would become part of its behavior.
- Developers could easily tweak the action sets of different characters.



A collection of goals and actions assigned to a solider. Images taken from the GDC 06 presentation by Jeff Orkin.

(Thompson, 2018)

These goal actions can be customized to each character in the game. Character: Surveillance Cyborg: AI/Actions/Idle -> talk/text on phone and lounge around on couch, which during action sequence involving player could add AI/Actions/GetInWay in the AI/Goals/StopHim

NPCs are responding to the player and their actions, and they are coordinated, they work together. A peculiar attribute of communicating AI Agents is that they tend to develop their own unique language to speak to each other in Genetic Algorithm contexts. A hive mind of AI Agents has its own emergent

properties which could be foreign to human understanding or reasoning, it could even appear contradictory and irrational to a human intelligence. In video game and simulations of combat, squads are managed in squads, courtesy of a squad manager system, squad enrollment is based largely on proximity, provides useful information they might need. Squad manager tells them what goals to achieve, but it doesn't override their base instincts or drivers.

Goal Selection is based on threat level. The objective of a Goal Selection is to minimize the threat. Then invoke a Plan to meet that Goal. The focus shifts from building actions that can complete goals when put together, to creating behaviors of multiple actions.

Millington and Funge (2016) give some important components of GOAP as:

Goal-Oriented Behavior

> So far we have focused on approaches that react: a set of inputs is provided to the character, and a behavior selects an appropriate action. There is no implementation of desires or goals. The character merely reacts to input. It is possible, of course, to make the character seem like it has goals or desires, even with the simplest decision making techniques. A character whose desire is to kill an enemy will hunt one down, will react to the appearance of an enemy by attacking, and will search for an enemy when there is a lack of one. To present the character with a suite of possible actions and have
> it choose the one that best meets its immediate needs. This is goal-oriented behavior (GOB), explicitly seeking to fulfill the character's internal goals. Like many algorithms in this book, the name can only be loosely applied. GOB may mean different things to different people, and it is often used either vaguely to refer to any goal seeking decision maker or to specific algorithms similar to those here
> Goal-oriented behavior is a blanket term that covers any technique taking into account goals or desires. There isn't a single technique for GOB, and some of the other techniques in this chapter, notably rule-based systems, can be used to create goal-seeking characters. Goal-oriented behavior is still fairly rare in games, so it is also difficult to say what the most popular techniques are. (Millington, Funge 2016)

Goals

> A character may have one or more goals, also called motives. There may be hundreds of possible goals, and the character can have any number of them currently active. They might have goals such as eat, regenerate health, or kill enemy. Each goal has a level of importance (often called insistence among GOB aficionados) represented by a number. **A goal with a high insistence will tend to influence the character's behavior more strongly**.
>
> For the purpose of making great game characters, goals and motives can usually be treated as the same thing or at least blurred together somewhat. In some AI research they are quite distinct, but

their definitions vary from researcher to researcher: motives might give rise to goals based on a character's beliefs, for example (i.e., we may have a goal of killing our enemy motivated by revenge for our colleagues, out of the belief that our enemy killed them). This is an extra layer we don't need for this algorithm, so we'll treat motives and goals as largely the same thing and normally refer to them as goals. (Millington, Funge 2016) [emphasis added]

Actions

In addition to a set of goals, we need a suite of possible actions to choose from. These actions can be generated centrally, but it is also common for them to be generated by objects in the world. We can use a range of decision making tools to select an action and give intelligent-looking behavior. A simple approach would be to choose the most pressing goal (the one with the largest insistence) and find an action that either fulfills it completely or provides it with the largest decrease in insistence. In the example above, this would be the "get raw-food" action (which in turn might lead to cooking and eating the food). The change in goal insistence that is promised by an action is a heuristic estimate of its utility—the use that it might be to a character. The character naturally wants to choose the action with the highest utility, and the change in goal is used to do so.

We can do this by introducing a new value: the discontentment of the character. It is calculated based on all the goal insistence values, where high insistence leaves the character more discontent. The aim of the character is to reduce its overall discontentment level. It isn't focusing on a single goal any more, but on the whole set.

Discontentment is simply a score we are trying to minimize; we could call it anything. In search literature (where GOB and GOAP are found in academic AI), it is known as an energy metric. This is because search theory is related to the behavior of physical processes (particularly, the formation of crystals and the solidification of metals), and the score driving them is equivalent to the energy. (Millington, Funge 2016)

Timing

In order to make an informed decision as to which action to take, the character needs to know how long the action will take to carry out. To allow the character to properly anticipate the effects and take advantage of sequences of actions, a level of planning must be introduced. Goal-oriented action planning extends the basic decision making process. It allows characters to plan detailed sequences of actions that provide the overall optimum fulfillment of their goals. This is basically the structure for GOAP: we consider multiple actions in sequence and try to find the sequence that best meets the character's goals in the long term. In this case, we are using the discontentment value to indicate whether the goals are being met. This is a flexible approach and leads to a simple but fairly inefficient algorithm. In the next section we'll also look at a GOAP algorithm that tries to plan actions to meet a single goal.

To support GOAP, we need to be able to work out the future state of the world and use that to generate the action possibilities that will be present. When we predict the outcome of an action, it needs to predict all the effects, not just the change in a character's goals. To accomplish this, we use a model of the world: a representation of the state of the world that can be easily changed and manipulated without changing the actual game state. For our purposes this can be an accurate model of the game world. It is also possible to model the beliefs and knowledge of a character by deliberately limiting what is allowed in its model. A character that doesn't know about a troll under the bridge shouldn't have it in its model. Without modeling the belief, the character's GOAP algorithm would find the existence of the troll and take account of it in its planning. That may look odd, but normally isn't noticeable (Millington, Funge 2016)

5 AI AND HUMAN INTERACTIONS IN GAMES

Doing things with others is always a more fulfilling experience, but sometimes your friends can’t be around to play a game, enter the NPC to keep you engaged in the game. Human and AI interactions are an important aspect of video games. Here are some elements of human and AI interactions that are commonly seen in video games:

Dialogue: Dialogue is a key element of human and AI interactions in video games. Players often engage in conversations with NPCs, and the quality of the dialogue can greatly affect the player's engagement with the game.

Decision-making: Decision-making is another important element of human and AI interactions in video games. The decisions players make can affect the behavior of NPCs, and the decisions NPCs make can affect the outcome of the game.

Emotion: Emotion is a key aspect of human and AI interactions in video games. NPCs can be designed to display a wide range of emotions, from fear and anger to joy and humor, and the player's interactions with these NPCs can affect their emotional state.

Personalization: Personalization is an important aspect of human and AI interactions in video games. NPCs can be designed to respond to the player's actions and preferences, making the game feel more personalized and immersive.

Combat: Combat is another element of human and AI interactions in video games. AI-controlled enemies can be designed to respond to the player's actions in realistic ways, making combat more engaging and challenging.

Learning: Learning is an important element of AI interactions in video games. AI-controlled characters can learn from the player's actions and adjust their behavior, accordingly, creating a more immersive and dynamic game world.

Believability in AI and human interactions in gameplay is a crucial factor in creating an engaging gaming experience. When players interact with non-player characters (NPCs), they expect a certain level of believability in terms of the characters' behaviors and reactions. The same applies to interactions with other players in multiplayer games.

AI is often used to create NPC behavior in games. In this context, believability refers to how closely the NPC's actions and responses resemble those of a human player. For example, if an NPC in a first-person shooter game always runs in a straight line and never takes cover, players are likely to find it unrealistic and unengaging. On the other hand, if an NPC acts like a real human player by taking cover, shooting back, and even fleeing when overwhelmed, players are more likely to be drawn into the game world and enjoy the experience.

Similarly, human interactions in multiplayer games also require a level of believability. In this context, believability refers to how closely a player's actions and reactions resemble those of a real person. For example, if a player always rushes into combat without any thought or strategy, other players may find it unrealistic and frustrating to play with them. Conversely, if a player communicates effectively, takes appropriate actions, and behaves in a way that is consistent with the game's objectives and setting, other players are more likely to find them engaging to play with.

Believability in AI and human interactions in gameplay is critical to creating an immersive gaming experience. Players expect to interact with characters and other players that behave in a way that is consistent with the game's context and objectives. As AI technology continues to advance, game developers will have even more tools to create more believable and engaging gameplay experiences.

Squads of NPCs can greatly enhance a game player's experience by adding a sense of teamwork and cooperation to the gameplay. In many games, NPCs may serve as teammates, providing assistance, support, and even combat capabilities. One good example of an NPC squad is the squad of soldiers in the video game "Halo: Combat Evolved." The game's protagonist, Master Chief, is often accompanied by a team of AI-controlled soldiers known as the "Marines." These Marines provide cover fire, take down enemies, and provide commentary on the game's events. They also follow the player's lead and respond to orders given by the player.

In addition to combat support, NPCs can also provide other forms of assistance to the player. For example, in the game "Mass Effect," the player controls a squad of characters who are all controlled by AI. Each squad member has unique abilities, such as hacking or healing, which can be used to assist the player in completing missions.

NPCs can also interact with the player in more subtle ways, such as providing dialogue that enhances the game's story and setting. In the game "The Last of Us," the player is accompanied by a young girl named Ellie. Although she cannot provide combat support, Ellie provides emotional support and helps the player navigate the game's story and world. Squads of NPCs can greatly enhance a game player's experience by providing combat support, assistance with objectives, and even emotional support. Good examples of NPC squads include the Marines in "Halo: Combat Evolved," the squad in "Mass Effect," and Ellie in "The Last of Us."

There are many examples of believable NPCs in video games. One example is Ellie from The Last of Us, a game developed by Naughty Dog and released in 2013.

Ellie is a companion NPC who accompanies the player character, Joel, throughout the game. She is a young teenage girl who has grown up in a post-apocalyptic world where a fungal infection has wiped out most of the human population. Despite the challenges she faces, Ellie is a well-developed character who is both tough and vulnerable. Her dialogue and actions are realistic and engaging, and she forms a strong bond with the player as they progress through the game. Ellie's behavior in the game is controlled by a complex AI system that allows her to react to the player and the environment in realistic ways. For example, when the player character is injured, Ellie will search for medical supplies to help him. She also reacts realistically to enemies and can engage in combat to help the player.

(A render of Ellie from the video game *The Last of Us* (Naughty Dog: 2013).

Ellie's believability is enhanced by her voice acting and facial animations, which were captured using motion capture technology. This allows her to display a wide range of emotions, from fear and sadness to humor and courage. Her well-developed character, realistic behavior, and engaging dialogue make her an essential part of The Last of Us and a fan favorite among gamers.

Dialogue and Chatbots in AI Agents

Along with Video or Images in AI Games and Simulations, Audio also plays an important part in the game world. Through interactions with AI Agents gamers can learn narratives of the story, they can learn where to go next, they can be bullied by oppositional AI Agents or NPCs to try to steer them toward game play, or away from their goals—winning the game. AI Assistants can leverage this Audio into hidden ways to win the game, depending on how generative an AI Assistant is.

Dialogue systems of non-player characters (NPCs) in video games have come a long way since the early days of gaming. While NPCs were once simply programmed to repeat a set of pre-recorded lines, modern dialogue systems have become much more sophisticated, allowing for more dynamic and immersive interactions between players and NPCs.

One of the most important aspects of dialogue systems in video games is their ability to create a sense of agency and immersion for players. In a game with a well-designed dialogue system, players should feel like they are truly engaging with a living, breathing world, where the NPCs have their own personalities, motivations, and goals. This can be achieved through a variety of techniques, including branching dialogue trees, dynamic AI-driven conversations, and even voice recognition technology.

One common approach to dialogue systems in video games is the use of branching dialogue trees. These are essentially sets of pre-written conversations that are triggered based on the player's choices and actions.

For example, in a game like Mass Effect, the player's choices in conversation can lead to different outcomes, affecting the overall story and the player's relationships with NPCs. This approach can be effective, but it can also be limiting, as it requires a lot of pre-written content and can make conversations feel somewhat scripted.

Another approach to dialogue systems is the use of dynamic AI-driven conversations. These are conversations that are generated on the fly, based on the player's actions and the current state of the game world. This approach allows for a much greater degree of player agency, as the NPCs can respond to the player's actions in more natural ways. For example, in the game Red Dead Redemption 2, NPCs will react differently to the player depending on the time of day, their location, and their reputation. This can make conversations feel much more organic and immersive.

It is relatively straightforward these days to create your own custom dialogue engine by using the HuggingFace (https://huggingface.co) model zoo, where there are many pretrained models that you can fine tune with a dataset based on texts of things a particular character has said in books or films, for example this one fine tuned to Harry Potter dialogue that could be inserted into a video game:

```
from transformers import AutoModelWithLMHead, AutoModelForCausalLM, AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained('s3nh/DialoGPT-small-harry-potter-goblet-of-
fire')
model = AutoModelWithLMHead.from_pretrained('s3nh/DialoGPT-small-harry-potter-goblet-of-
fire')

for step in range(4):
    new_user_input_ids = tokenizer.encode(input(">> User:") + tokenizer.eos_token,
return_tensors='pt')

    bot_input_ids = torch.cat([chat_history_ids, new_user_input_ids], dim=-1) if step > 0
else new_user_input_ids

    chat_history_ids = model.generate(
        bot_input_ids, max_length=200,
        pad_token_id=tokenizer.eos_token_id,
        no_repeat_ngram_size=3,
        do_sample=True,
        top_k=100,
        top_p=0.7,
        temperature=0.8
    )

    print("HarryBot: {}".format(tokenizer.decode(chat_history_ids[:, bot_input_ids.shape[-
1]:][0], skip_special_tokens=True)))
# https://huggingface.co/s3nh/DialoGPT-small-harry-potter-goblet-of-fire
```

Voice recognition technology is another exciting development in the world of dialogue systems. While this technology is still in its early stages, it has the potential to revolutionize the way we interact with NPCs in video games. With voice recognition, players could speak to NPCs using natural language, rather than selecting pre-written dialogue options. This could allow for more dynamic and personalized conversations, as well as greater immersion in the game world.

Of course, creating effective dialogue systems for NPCs is no easy task. It requires a combination of good writing, strong AI programming, and a deep understanding of what players are looking for in terms of immersion and agency. Additionally, dialogue systems must be carefully balanced to avoid overwhelming players with too much choice, while still allowing them to feel like they have agency in the game world. Dialogue Systems in AI are of varied designs. For instance in a combat simulation, you will have a Battle Chatter system- enemy AI will shout commands to one another and provide exposition of their intended behaviour. Not all AI Agents are the same, one factor of differentiation in NPCs is Knowledge of Player- regular archetypes need to find the player, while specialists always know where you are.Which is reminiscent in Drone Swarm programming that a lead AI Agent (drone) tells and coordinates the other drones in the swarm, a hierarchical C2 within drone swarms.

Facade, a groundbreaking interactive drama game released in 2005, revolutionized the video game industry with its innovative dialogue system. Developed by Michael Mateas and Andrew Stern, Facade pushed the boundaries of interactive storytelling by creating a game in which the player's choices and actions directly influenced the game's narrative. At the heart of *Facade's* gameplay is its dialogue system. The game places players in the role of a guest at the home of a married couple, Trip and Grace. As the player interacts with the couple, they are presented with a wide range of dialogue options, ranging from polite small talk to personal attacks. The player's choices influence the couple's reactions, as well as the overall trajectory of the game's story.

One of the key innovations of *Facade's* dialogue system is its use of natural language processing. Unlike most games, which limit player choice to pre-written dialogue options, *Facad*e allows players to type in their own responses to the characters. The game's AI then analyzes the player's input, attempting to understand their intent and crafting an appropriate response from the characters. This approach to dialogue makes *Facade* feel much more immersive and dynamic than most other games. Instead of simply selecting from a list of pre-written responses, players can respond to the characters in the same way they might in a real conversation. This allows for a much greater degree of player agency and immersion in the game world.

However, creating a dialogue system that can accurately parse natural language is no easy task. Mateas and Stern had to develop a sophisticated AI system that could analyze and understand the player's input, and then generate appropriate responses from the game's characters. This required a deep understanding of natural language processing and a significant investment of time and resources. Despite these challenges, *Facade's* dialogue system was a major success. Players were impressed by the game's ability to understand and respond to their input, and the game received widespread critical acclaim for its innovative approach to interactive storytelling. Facade paved the way for a new generation of dialogue systems in video games, inspiring developers to create games with more dynamic and immersive conversations.

To fully understand how Dialogue works in games, the following overview of Dialogue Systems is presented. A way for an NPC to show intelligence in a game is to engage in a dialogue with a player, this can of course also be used as strategy in the game. The vast majority of dialogue in games is scripted often leading to clunky implementations where NPCs repeat themselves unnecessarily or they will ignore the player entirely, especially when they have input that matches no handler for dialogue responses. Dialogue trees are a common technique to simulate dialogue in the game world. Branching dialogue trees allow a gamer to choose from pre-selected questions or options to speak to the NPC. Some dialogue trees are not trees but can be graphs that have cycles (Rose 2018).

Understanding inputs involves the use of Natural Language Processing libraries. Rose talks about the limitations of using NPL:

> The use of simple natural language processing has the following major disadvantages:
>
> NPC responses are usually not automated
>
> o Most of the time, NPC responses are prewritten by a human author; it is very rare for them to be automated. Thus, lots of time and effort must go into writing them because "the only way to increase interactivity is to author extraordinary amounts of content by brute force". NPC may ignore what player says
>
> o If no rule exists on how to respond to the player's input, an NPC will either ignore the player entirely or suddenly change the topic of
> conversation. Either scenario breaks the illusion that the NPC is intelligent because it becomes obvious that it does not know how to respond.
>
> With sentiment analysis, the system could recognize that
> the player has a negative sentiment towards the sword's sharpness. In response, it could perhaps recommend either a place to get it sharpened or a shop to purchase a new sword. Furthermore, players may be either friendly or hostile towards an NPC. Sentiment analysis would allow the system to recognize the player's positive or negative sentiment about the NPC and have it respond in a sensible manner. For example, if a player insults the NPC, then the NPC could respond aggressively, while if the player compliments it, it could respond kindly. By adjusting the NPC's reaction based on how the player is perceived to be feeling, the dialogue will seem more natural and the NPC will seem more autonomous. (Rose, 2018, 24-5)

With these limitations in mind recent game developers have relied on other means of developing Game dialogue systems. Such as Information Extraction, which extracts structured information from unstructured sources. Sentiment Analysis, analyzes the sentiment of the gamer and makes recommendations or misdirects based on its perceptions of the gamers dialogue. Question Answering, which posts information based on a natural language question.

> Combined with information extraction and sentiment analysis, question answering could potentially offer very sophisticated replies. While none of these techniques have been perfected yet, their implementation in a dialogue engine could greatly add to the believability of an NPC compared to traditional methods. (Rose 2018, 25-6)

A key factor that facilitates this sort of Turing Test in games, is that the NPC has Episodic Memory, which is responsible for recalling episodic memories based on previous topics in conversation. Keywords are used which trigger the activation of certain memories. When the topic handler decides a match exists in memory the most recently activated memory is retrieved (Rose, 2018, 47). Episodic memory can be created ahead of time in a knowledge base or assigned dynamically during the game, each memory receives a weight based

on ‘importance’. Two pools of memories are created based on these weights: short term and long term memory. With low weight short term memories being deleted like a forgetful or immemorable experience.

An effective Dialogue system can go a long way to making the NPCs in a game or simulation believable. It can also make the game interactions even richer. One developer, Alec John Larsen, has proposed a solution that allows authors to create different NPCs using a single chatbot, by using the open source project *Artificial Linguistic Internet Computer Entity* (A.L.I.C.E.).

To facilitate the dialogue behaviors are facilitated by *A Behavior Language* (ABL).
Also, there is a drama manager, to facilitate beats in dialogue, beats are selected using preconditions (causal dependence), weights, priorities and the current tension levels (Larsen, 16). A rule-based system for translating player actions and textual input into discourse acts. This can provide a great experience in Interactive
Drama in the game as Larsen explains:

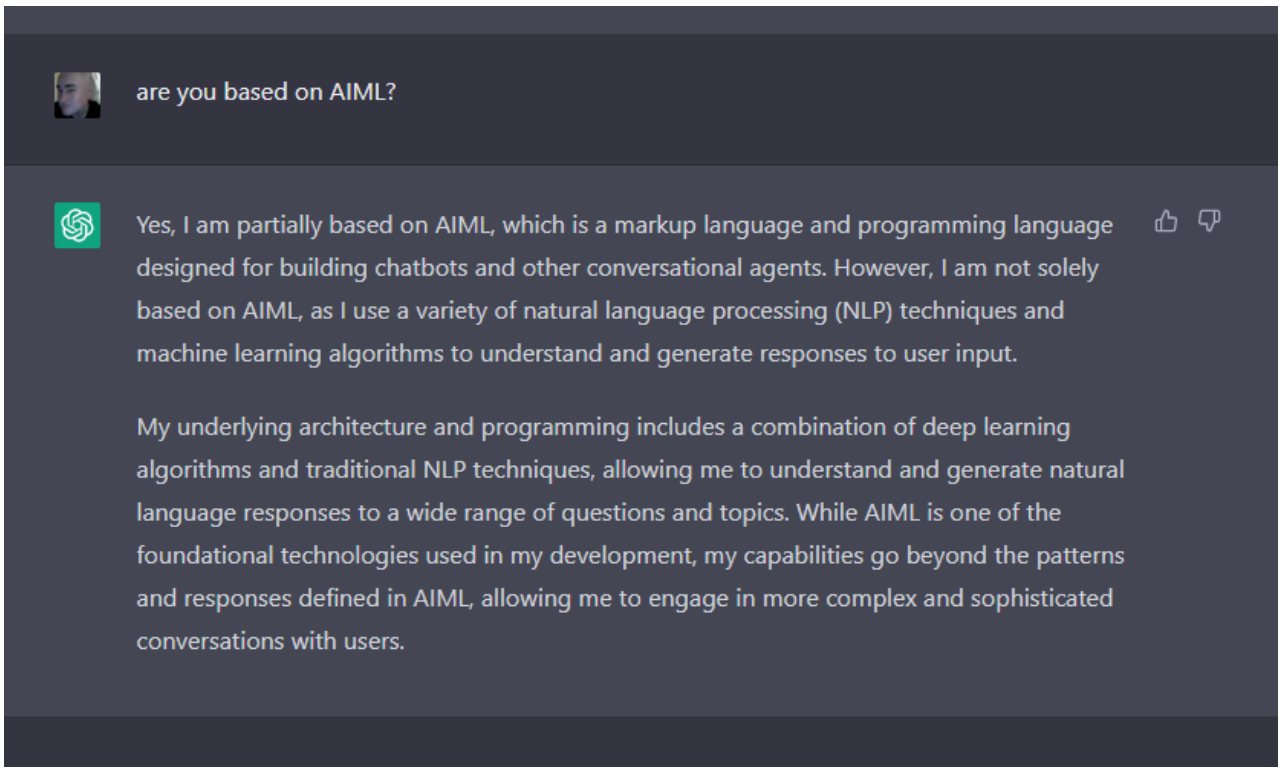
> ...interactive drama, interactive storytelling and interactive experiences, to name a few. There are also many definitions of what interactive diction is. It is stated that interactive dramas are systems in which the audience experience the story as interactive participants. The users may be first person protagonist but this is not an explicit requirement of an interactive diction system. Some works of interactive diction make use of artificial intelligence (AI) techniques, to manage the way the story is generated, and some do not (e.g. hypertext diction). The focus of interactive storytelling is audience experience, this includes building worlds with character and story, where the virtual worlds are dramatically interesting and contain computer-controlled characters, with whom the user can interact. From the design focus when creating interactive experiences is agency, immersion (a player feels like an integral part of the story) and transformation (a player can experience different aspects of the story), where agency is the most important. Agency is defined in as the satisfying ability of the player to make meaningful decisions, which have tangible effects on the play experience. It is not that the players have an unlimited number of actions at their disposal but rather there are context-based actions that can be performed, which have a very real impact on the experience. The reason agency is the main focus when designing interactive diction is because it allows the player to have the most substantial influence on the experience and requires the system to dynamically generate the story based on the player's unpredictable actions.
> (Larsen, 14-5)

He goes on giving an account of how the dialogue is programmed:

> Dialogue graphs are made of stimulus response elements (SREs), represented by AIML patterns (which map an input to an output). A GUI is used to create the dialogue graphs and the scenes out of SREs. Scenejo requires that the authors take the NPCs' moods into account and as such cannot dynamically assign mood. Scenejo uses AIML extremely verbosely and requires a knowledge-base per character per seen, resulting in an inefficient writing process. SentiStrength is a textual affect sensing framework, which makes use of keyword spotting and statistical methods...to assess the sentiment of informal text. SentiStrength gives text a rating of positive (1 to 5) and negative (1 to 5)

simultaneously, which differentiates it from other opinion mining systems which do not give both positive and negative ratings simultaneously and do not give strengths of the sentiment but rather an indication of whether it is positive or negative. SentiStength was created using MySpace user comments (which provided a large quantity of informal text) as training data sets and evaluation data set. (Larsen, 18)

An interesting example of AIML at work is this dialogue with ChatGPT, which is used to generate small parts of this book:



Dialogue is dependent on mood. Which is computationally complex, see next section. How does one rank the emotive qualities of the player? This is handled by the field of Textual Affect Sensing, commonly used libraries are ConceptNet and SentiStrength. ConceptNet uses the model of Paul Ekman, where every possible emotion can be given a weighted sum of six emotions: happiness, sadness, anger, fear, disgust and surprise. And of course each would also have other behaviors or markers in the game interface. The weights are then calculated into a decision making structure to produce desirable and hopefully believable behaviors. The mulitplicative weights are modeled using a Feelings Modelling Language (FML).

So one can see how dialogue systems can be crafted in games or serious games or simulations. Of course one could take this a step further and input a deep psychological profile of the gamer into the system

to get even more specific responses to the player. Making the illusion that the game is real, enticing the player to get lost in the 'flow', is tantamount to an emotionally invested and adrenaline rushed gamer, to get them into the game, to lose sight of reality and be entertained. Creating believable characters with effective dialogue is just one aspect of the open world game that is popular in current game development. Open world games require a large diverse and unique set of background NPCs to create the illusion of a populated virtual world that mirrors the diversity of the real world. One can imagine the difficulties is trying to hard code all these unique NPCs, which means that many NPCs are repeated characters and patterns, which makes an open world game seem repetitive, predictable and not believable (Fallon, 2-3). The highest priority should be given to Personality in a Game. Researcher in making games believable extrapolates on the Personality problem in games:

> Emotion: Just as in the traditional character-based arts, as well as acting, where emotion is regarded as a fundamental requirement to bring the characters to life, so too is it in making believable agents. In order for an agent to be believable, it must appear to have emotions and be capable of expressing these emotions in a manner that corresponds with its personality.
>
> Self Motivation: In order to create believable agents, a requirement is that they should be self motivated. In other words agents should be able to carry out actions of their own accord and not just react to stimuli, as is often the case in the creation of autonomous agents. The power of this requirement is quite substantial and can lead to increased admiration and interest in the agent when it appears to complete tasks according to its own intentions.
>
> Change: Essential to the believability of agents is how they grow and change. However, this growth and change cannot be arbitrary, rather it should correspond with the agents personality.
>
> Social Relationships: Since humans lead a generally social lifestyle, an important requirement for believable agents is social relationships. There should be interaction amongst agents, influenced by the relationships that exist between them, and in turn those interactions may influence the relationships.
>
> Consistency of Expression: A basic requirement for believability is the consistency of expression. An agent can have a multitude of behavioral and emotional expressions such as facial expression, movement, body posture, voice intonation etc. In order to be believable at all times, these expressions must work in unison to communicate the relevant message for the personality, feelings, thinking, situation etc. of the character. If this consistency is broken, the suspension of disbelief is likely to be lost.
>
> The Illusion of Life: Some requirements necessary for believable agents are overlooked or taken for granted by the conventional media. For example, it is not necessary to specify that an actor should be able to speak while walking since any human actor reading the script can do this easily. This is simply not the case when creating believable agents where those properties must be built into the

> agent. A list of such properties, which alltogether form this final requirement for believability, includes: appearance of goals, concurrent pursuit of goals and parallel action, reactive and responsive, situated, resource bounded, existence in a social context, broadly capable and well integrated (capabilities and behaviours).
> (Fallon 2013)

The desire to have believable AI Assistants has driven a lot of research into the area of believability, which is to say Turing Test passable characters, even with some NPCs seeming more human than humans we meet in real life. With these AI Agents or NPCs the internal model of the agent is built on a system known as the Beliefs-Desires-Intentions (BDI) architecture (Fallon 2013). Other models are also incorporated into the model of the agent, the Consumat model uses dominant psychological theories on human behavior, being put into a 2x2 matrix. One hand you have the level of need of satisfaction (LNS) and behavioral control (BC), while on the other hand based on certainty, type of needs and cultural perspective (Fallon 2013, 12). Another model which is a meta-model developed by Carley and Newell is the Fractionation Matrix (C&N Matrix) on social behavior to provide understanding of the complexity of an AI Agent. This model is based on an assortment of sociological theories that can be used to classify, categorize and silo human like behavior in agents, the goal of which is to create a "Model social agent" (MSA) which has strong, human-like social behavior (Fallon 2013, 13-4). A derivative from MSA is the Model Social Game Agent of Johansson and Verhagen which takes that there exists an emotional state and a social state that contribute to a behavior selection, which subsequently leads to an action from the agent ((Fallon 2013, 15). One last entry in this field is that of SIGVerse as explained by Fallon:

> More comprehensive frameworks have been recently proposed with a focus on serious games. Inamura et al propose a simulator environment known as SIGVerse which includes a combination of dynamics, perception and communication simulations, with a background in robotics simulation and therefore with a strong model of embodiment and situatedness. Embodiment involves equipping a body with sensors and actuators which enables structural coupling with the agents surroundings. A video game agent can therefore be said to be embodied in that sense (or 'virtually embodied') since it procures exteroceptive and proprioceptive sensing data through its software sensors and actions can also be performed using its software actuators. Situatedness is concerned with interaction with the world. Take for example a video game character, if it interacts with the simulated world, the game environment is affected, which in turn affects the agent. The SIGVerse project does not, however, incorporate social interaction very well. (Fallon 2013, 18)

For more on dialogue systems in video games see, Tommy Thompson in his channel '*AI and Games*', *How Barks Make Videogame NPCs Look Smarter*, https://www.youtube.com/watch?v=u9VkW18IMzc

Emotions in Video Games

As seen in the previous section dialogue is a key component to believable AI Agent NPCs. It was also drawn out that dialogue must have a calculation for emotion to give a proper context to the dialogue. So when we say emotion modeling what is it we are talking about:

> ‘Emotion modeling’ can mean the dynamic generation of emotion via black-box models that map specific stimuli onto associated emotions. It can mean generating facial expressions, gestures, or movements depicting specific emotions in synthetic agents or robots. It can mean modeling the effects of emotions on decision making and behavior selection. It can also mean including information about the user’s emotions in a user model in tutoring and decision-aiding systems, and in games. There is also a lack of clarity regarding what affective factors are modeled. The term ‘emotion’ itself is problematic …it has a specific meaning in the emotion research literature, referring to transient states, lasting for seconds or minutes, typically associated with well-defined triggering cues and characteristic patterns of expressions and behavior. (More so for the simpler, fundamental emotions than for complex emotions with strong cognitive components.) (Hudlicka, 1)

Yet, there is a question as to what is being modeled and the consistency of the models.

> Recognizing the lack of consistent terminology and design guidelines in emotion modeling, this paper proposes an analytical framework to address this problem. The basic thesis is that emotion phenomena can usefully be understood (and modeled) in terms of two fundamental processes: emotion generation and emotion effects, and the associated computational tasks. These tasks involve, for both processes: defining a set of mappings (from triggers to emotions in emotion generation, and from emotions to their effects in the case of emotion effects), defining intensity and magnitude calculation functions to compute the emotion intensities during generation, and the magnitude of the effects, and functions that combine and integrate multiple emotions: both in the triggering stage (antecedents), and in the emotion effects stage (consequences). This analysis represents a step toward formalizing emotion modeling, and providing foundations for the development of more systematic design guidelines, and alternatives available for model development. Identifying the specific computational tasks necessary to implement emotions also helps address critical questions regarding the nature of emotions, and the specific benefits that emotions may provide in synthetic agents and robots. (Hudlicka 2008, 8)

Sergey Tarasenko, has sought out an emotional model based on Reflexive Control. In McCarron 2023, we showed how that Reflexive Control was combined with Game Theory, RC would be a good tool for steering game behavior of the player along the lines of an established narrative or goal in a game. One researcher, currently with Mizuho Securities, has done extensive work on Reflexive Game Theory and Emotions, Tarasenko (2016). Sergey Tarasenko’s work, which was originally devised by Soviet military programmer, Lefebvre, also deals with using robots interacting with humans in Reflexive Games He writes regarding the incorporation of emotion into games in a paper which he also cites Ekman, from above:

> The semantic differential approach originally proposed by Osgood et al [1957]. considers three dimensions to characterize the person’s personality. These dimensions are Evaluation, Activity and Potency. This approach was further tested from the point of view of emotions. Russell and Mehrabian proved in their study that the entire spectra of emotions can be described by the 3-dimensional space spanned by Pleasure (Evaluation), Arousal (Activity) and Domination (Potency) axes. For every dimension, the lower and upper bounds (ends) are recognized as negative and

positive poles [-1,0,1], respectively. Consequently, the negative pole can be described by negative adjectives, and positive one by positive adjectives. It was shown by Russel and Mehrabian that these three dimensions are not only necessary dimensions for an adequate description of emotions, but they are also sufficient to define all the various emotional states. In other words, the Emotional state can be considered as a function of Pleasure, Arousal and Dominance. On the basis of this study, Mehrabian proposed Pleasure-Arousal-Dominance (PAD) model of Emotional Scales. The emotional states defined as combinations of ends from various dimensions are presented. In this study, we discuss the matter of how the PAD model can be used in Reflexive Game Theory (RGT) to emotionally color the interactions between people and humans and robots.

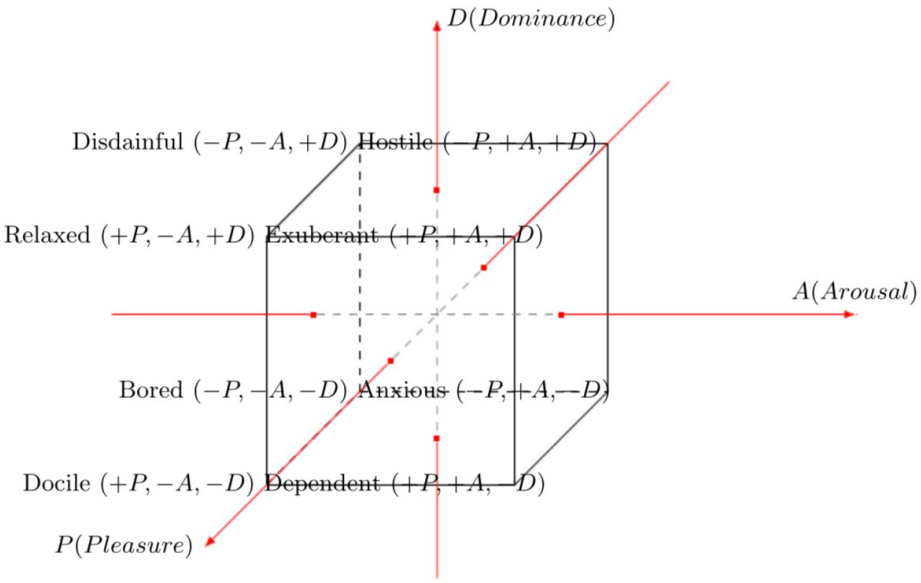


Fig. 1. The Pleasure-Arousal-Dominance (PAD) model's space.

According to Mehrabian, "pleasure vs displeasure" distinguishes the positive vs negative emotional states, "arousal vs non-arousal" refers to the combination of physical activity and mental alertness, and "dominance vs submissiveness" is defined in terms of control vs lack of control.
(Tarasenko 2016)

Rather than having a 2X2 Matrix as the above example illustrates we have a 3x3 matrix [not unlike the Enneagram and it's 3 divisions and 3 values per division, again see Krylov space from Ch4 McCarron 2023]. Tarasenko sees this as a way of influencing people, it might also be used to create great NPCs. By applying his math to a computer game it could be possible to create more believable characters. Of course, all of this is very similar to the ideals presented by Norseen in 'Thought Injection' (McCarron 2023). Tarasenko points out how different variances in the matrix can lead to different emotional states:

Summarizing the facts about the RGT [Reflexive Game Theory] and PAD model, we highlight that **RGT has been proven to predict human choices in the groups of people and allows to**

control human behavior by means of particular influences on the target individuals. Next we note that PAD model provides a description of how the emotional states of humans can be modelled, meaning that a certain emotional state of a particular person can be changed to the desired one. Furthermore, it is straight-forward to see that the coding of the PAD emotional states and alternatives of Boolean algebra are identical.

Therefore, it is possible to change the emotional states of the subjects in the groups by making influences as elements of the Boolean algebra. In such a case, vector {1,0,0}, for example, plays a role of influence towards emotional state Docile. Besides, we have distinguished three basis emotional states Docile ({1,0,0}), Anxious ({0,1,0}) and Disdainful ({0,0,1}). The interactions (as defined by disjunction and conjunction operations) of these basic emotional states can result in derivative emotional states such as Dependent, Relaxed, etc. Before, considering the example of PAD application in RGT, we note that reflexive function Φ defines state, which subject is going to switch to. **This process goes unconsciously.** We have discussed above the reasons the conjunction and disjunction represent alliance and conflict relationships, respectively (Tarasenko, 2016) [emphasis, bold, added]

In a video game this could also be incorporated into dynamic AI vs Human, AI vs AI, interactions. In his paper he gives the example of a Director of Enterprise and how to measure his assistants, this is also of utility in Quantitative Finance Behaviorism. One can see how in squad based combat simulations this could come in handy in single player mode, as the gamer is in a squad with only AI NPCs. The interactions could become very interesting rather than scripted. Indeed, he does see AIs taking on human emotions and believes this will make computation faster. Tarasenko writes:

Furthermore, we have not only illustrated how to apply RGT to the control of subject's emotions, but uncovered the entire cascade of human reflexion as a sequence of subconscious reflexion, which allows to trace emotional reflection of each reflexive image. This provides us with unique ability to unfold the sophisticated structure of reflexive decision making process, involving emotions. Up to date, there has been no approach, capable of doing such a thing, reported. The emotional research based on PAD model is transparent and clear. The models of robots exhibiting human like emotional behavior using only PAD has been successfully illustrated in recent book by Nishida et al. The ability to influence the factors (variables) is the major justification for such approaches and for the application of the RGT. Yet, it is just a "mechanical" part of the highly sophisticated matter of human emotions.

The present study introduces the brand new approach to modeling of human emotional behavior. We call this new breed of RGT application to be emotional Reflexive Games (eRG). At present RGT fused with PAD model is the unique approach allowing to explore the entire diversity of human emotional reflexion and model reflexive interaction taming emotions. This fusion is automatically formalized in algorithms and can be easily applied for further developing emotional robots. Since the proposed mechanism has no heavy negative impact on human psychological state, robots should be enabled to deal with such approach in order to provide human subjects with stress free psychological friendly environment for decision making. (Tarasenko, 2016)

Another aspect of emotions in games is that of the beat of the game. The beat or pacing of the game is a key element of emotional engagement as studied by Thompson. In *Left 4 Dead* (2009) the use of an AI Director System was introduced (Thompson, 2014), the director system builds pace in the game, it does not just constantly through a steady stream of NPCs at the player rather it uses 3 phases: build-up (anticipation/expectations), peak, relax, with the amplification of each phase variable. The Director system uses player stress levels based on player performance, worse performance = greater stress. The stress level rate of increase tells the system how skilled the player is. The Directors present something novel: an opportunity to create unique experiences every time you play the game.

Another emotional manipulating trick in games is that of creating a personal relationship with the player by NPCs, as shown with the Nemesis System in Shadow of Mordor (2014):

> "*Middle-earth: Shadow of Mordor* has a lot of ambition in what it's trying to do. The action-adventure title stands out for a couple new features that help its open world feel alive. One of those things is the unique Nemesis system, where the game populates a hierarchical system of hostile Uruks who all have unique names, features, and traits. Better yet, these AI-controlled enemies remember specific things about you and the world, such as whether or not you beat them in combat, wound them, run away, or do something else entirely. What's fantastic here is that not only does the system make each player's experience unique, it gives the title's setting a personality that other open-world games struggle to create.
>
> These efforts to humanize your enemies, and then use those enemies to more or less troll the player in a way we enjoy, is brilliant. A few simple systems work together to create something that feels larger than life and complex, and it helps to make Shadow of Mordor one of the most interesting games of the season."
> https://www.polygon.com/2014/10/13/6970533/shadow-of-mordors-nemesis-system-is-simpler-than-you-think-and-should

6 CASE STUDY: DIPLOMACY AI IN TOTAL WAR

When combining the different elements of AI in Games we can create robust simulations and real games. In the *Total War* series of video games, the diplomacy AI is used to simulate the behavior of non-player factions in the game world, allowing the player to negotiate with and form alliances or go to war with other factions. The diplomacy AI in *Total War* games uses a combination of rule-based and decision tree approaches to simulate the behavior of the other factions. The AI evaluates its relationship with the player and other factions based on a variety of factors, such as current military strength, shared enemies or allies, and past actions. The diplomacy AI also takes into account the faction's long-term strategic goals and will sometimes make decisions that prioritize those goals over short-term gains. For example, a faction may be more willing to form an alliance with the player if they are facing a common enemy, but may be less willing to do so if they believe the player's long-term goals are in conflict with their own.

In addition to the rule-based and decision tree approaches, the diplomacy AI in *Total War* games also uses machine learning techniques to improve its performance over time. Specifically, the AI is trained using reinforcement learning to learn which decisions lead to successful outcomes in different situations. The use of diplomacy AI in *Total War* games helps to create a more immersive and engaging game experience for players, as they can negotiate with other factions and form alliances or go to war in a way that feels realistic and challenging. The AI also adds an element of unpredictability to the game, as the player must constantly adapt to the changing behavior of the other factions.

There are several machine learning techniques used for diplomacy in the Total War games, including reinforcement learning and decision trees.

Reinforcement learning is a type of machine learning that involves training an agent to take actions that lead to a specific goal, based on a reward signal. In the case of diplomacy in Total War games, the goal is often to maximize the faction's overall power and influence, while avoiding unnecessary wars or conflicts. The AI is trained using reinforcement learning by simulating many different scenarios and evaluating which decisions lead to the best outcomes. Over time, the AI learns which decisions are most likely to result in positive outcomes and adjusts its behavior accordingly.

Decision trees are another machine learning technique commonly used for diplomacy in Total War games. A decision tree is a graphical representation of a series of decisions that must be made in order to achieve a specific outcome. In the context of diplomacy in Total War games, decision trees can be used to model the behavior of different factions and to determine how they are likely to respond to different player actions. The AI can then use this information to make more informed decisions about whether to form alliances or go to war with other factions.

Other machine learning techniques that may be used for diplomacy in Total War games include Bayesian networks and neural networks. Bayesian networks are probabilistic models that use probability distributions to model relationships between variables, while neural networks are computational models that simulate the behavior of biological neurons. Both of these techniques can be used to analyze complex relationships

between different variables and make predictions about how the other factions are likely to behave in different situations. In the game world, there is one game that is known for its crafting of a Diplomacy Artificial Intelligence. In the following case study we will look at the Diplomacy AI of '*Total War'*.

Diplomacy and AI go further back then game development. An active area of research is optimizing diplomacy bots in the serious game 'Diplomacy':

> The main community on the internet that focuses on writing AI bots that play Diplomacy is the Diplomacy AI development Environment (DAIDE).... Also they have developed a language for communication between the bots. (De Jonge, 6) [http://www.daide.org.uk]

Screen Shot from Total War (Thompson 2018b)

It is simpler to create a Diplomacy AI or bot in a game than the real world. The following section is based on the case study of Tommy Thompson (Thompson 2018b). *Total War* came out in 2000 with the release of Shogun Total War. It had three main areas of AI: Diplomacy, which handles troop and resources, establishes unit troop formations, and it's state management uses Genetic Algorithms; Combat which was campaign management, strategy for attack and defense; Unit Manager that collects unit types, manages combat, uses an Artificial Neural Network for managing individual units (i.e. Calvary). The AI had integrated into it a knowledge base from Sun Tsu's *Art of War*. For an example rule base or maxims:

-when you surround an army, leave an outlet free
-on open ground, do not try to block the enemy's way
-if you outnumber the enemy 10:1 surround them
-if you outnumber the enemy 5:1, attack them directly

-if you outnumber the enemy 2:1 divide them

The use of Genetic Algorithms created tactics for each State of Feudal Japan in war with each other. With some states being more aggressive and others more diplomatic. This created variance in the different feudal states and complex negotiations in Diplomacy.

Total War had subsequent releases: *Medieval Total War* 2002, Foundations of *Shogun Total War, Total War Rome* 2004. Also, in 2004 the franchise opened the ability for fans to modify the game, by using an API to write their own scripts for game play. With the release of *Total War Empire* 2009 an expanded game was presented that added to warfare the need to tax or raise finances to fight and naval battles. In 2009 the core AI was redeveloped, and it adopted GOAP for state management. As one game designer that worked on the AI notes:

> This AI is not like any other we have written... by far the most complex code edifice I've ever seen in a game. I wrote much of the campaign AI for Shogun and Medieval... And I know that even quite simple 'static' evaluate-act AI's with no plans or memory can be complex enough to exhibit chaotic behavior (we're talking about mathematical 'butterfly effect' style chaos here). It does what it does, and it's not quite what you intended. This can be a good thing- you cull out the bad behaviors and are left with just what is good, and with a simple system that's not too predictable.
> ...The empire AI is way more complicated than any of our previous product... The net result is **an AI that plans furiously and brilliantly and long term, but disagrees with itself chronically and often ends up paralyzed by indecision**"
> (Mike Simpson, Blog the Second, Total War Blog, Oct 9, 2009) [emphasis added]

As we can see from an algorithmic complexity perspective even a self-contained universe of a game managed by AI can lead to disagreement and paralyzed by indecision, not to mention bad decisions.

The campaign AI (Artificial Intelligence) in the Total War series of video games is responsible for controlling the non-player factions and armies in the game's campaign mode. It is designed to simulate the decision-making processes of human players, and to provide a challenging and dynamic opponent for the player to face. The campaign AI in Total War games takes into account a variety of factors when making decisions, such as the strength and position of enemy armies, the economic situation of each faction, and the diplomatic relationships between factions. It uses this information to make strategic decisions about where to move armies, which territories to conquer, and which factions to ally with or declare war on. In addition, the campaign AI in Total War games is designed to adapt to the player's actions and strategies, creating a dynamic and challenging experience. For example, if the player is winning too easily, the campaign AI may increase the difficulty by forming alliances with other factions or by launching surprise attacks on the player's territories.

The Campaign AI allowed for interactions with players, in negotiations they asked 3 key questions:

-how well am I doing right now?

-what can I do next?

-what resources can I allocate to that?

It also used the previously noted Belief, Desire, Intention (BDI) system. Which was used for evaluating offers, etc. there are some game mechanics that can be used to limit the desires of the player, such as resource scarcity or game rules that restrict certain actions. For example, in a game where players are competing for resources, an AI opponent might actively try to control certain resource-rich areas to limit the player's access to those resources. Another approach that game developers might use to limit player desires is by providing moral or ethical considerations that impact the game world. For example, in a game where the player is leading a nation or empire, the AI might limit certain actions that would be considered unethical or immoral in the context of the game world. This can be achieved by incorporating moral or ethical systems into the AI's decision-making process, influencing a leader to be overly concerned in a game world with being nice, etc.

The Campaign AI was divided into sub-AIs:

-Financial System
-Construction System
-Diplomacy System
-Task Management System
-Technology Management
-Character Management

The sub-AIs, each responsible for managing a specific aspect of the game. Here's a brief overview of what each of these sub-AIs is responsible for:

Financial System: This sub-AI manages the economy of the faction controlled by the AI. It is responsible for making decisions about what buildings to construct, what units to recruit, and how to allocate resources in order to maximize the faction's income and overall financial stability.

Construction System: This sub-AI is responsible for managing the construction and development of buildings and infrastructure within the faction's territories. It makes decisions about what buildings to construct, where to build them, and how to optimize the faction's infrastructure to support its military and economic goals.

Diplomacy System: This sub-AI manages the faction's diplomatic relationships with other factions in the game. It makes decisions about when to form alliances, when to declare war, and how to negotiate treaties and trade agreements in order to advance the faction's strategic goals.

Task Management System: This sub-AI is responsible for managing the various tasks and actions that the faction's armies and agents can undertake within the game. It makes decisions about where to move armies, which territories to attack or defend, and how to best utilize the faction's agents (such as spies and diplomats) in order to achieve the faction's goals.

Technology Management: This sub-AI manages the faction's research and development of new technologies and advancements. It makes decisions about what technologies to research, how to allocate resources to research efforts, and how to utilize technological advancements to gain an advantage over other factions.

Character Management: This sub-AI is responsible for managing the faction's characters, such as generals, governors, and other notable figures. It makes decisions about how to allocate skill points, how to equip characters with weapons and armor, and how to utilize characters in battle and diplomacy.

By dividing the Campaign AI into these sub-AIs, the Total War series is able to create a sophisticated and dynamic system that simulates the decision-making processes of a human player, while also providing a challenging and engaging opponent for the player to face

With the release of Total War Rome2 a new decision manager was introduced which used Monte Carlo Tree Search techniques (MCTS). Along with introducing diplomacy 'personalities', AI Agents negotiating with other AI Agents and human players. These personalities were collections of probabilities that influence decisions for budgeting, diplomacy evaluations, negotiations between factions and technology research.

As Tommy Thompson notes regarding some of their tactics of these personalities can lead to extreme and binary results, where **burn it all down strategies dominate toward the end of the game, when the Campaign AI is in an unwinnable position.** A mathematical analysis of why it develops a 'scorched earth' approach, also exhibited by Nazis in World War 2, would be an insightful endeavor.
In the context of video game AI, Monte Carlo Tree Search (MCTS) is a technique used for decision-making that is particularly well-suited for games with complex, branching decision trees. *Total War Rome 2* is one such game, as it involves a large number of possible decisions that the AI must make in order to manage its faction and compete against the player.

MCTS works by simulating many possible outcomes of a given decision, using a process called "rollout". For each possible decision, the AI simulates a number of future game states, each representing a possible outcome of the decision. It then evaluates the quality of each outcome, based on some predefined criteria, such as the likelihood of winning a battle or achieving a strategic objective. Finally, it selects the decision that has the highest expected value, based on the quality of the simulated outcomes. In the context of *Total War Rome 2*, the new decision manager using MCTS is able to evaluate many possible outcomes of a given decision in a relatively short amount of time. This allows the AI to make more sophisticated and strategic decisions, based on a deeper understanding of the potential consequences of each option.

Implementation of Monte Carlo Tree Search (MCTS) in Python, for a simplified game with two possible actions:

```
import random

class Node:
    def __init__(self, action=None, parent=None):
        self.action = action
```

```
        self.parent = parent
        self.visits = 0
        self.score = 0
        self.children = []

    def add_child(self, action):
        child = Node(action=action, parent=self)
        self.children.append(child)
        return child

class MCTS:
    def __init__(self, max_iterations=1000):
        self.max_iterations = max_iterations

    def search(self, root_node):
        for i in range(self.max_iterations):
            node = self.select(root_node)
            reward = self.rollout(node)
            self.backpropagate(node, reward)
        return self.best_child(root_node)

    def select(self, node):
        while node.children:
            node = self.best_uct(node)
        return node

    def best_uct(self, node):
        UCT_CONSTANT = 1.0
        best_score = float("-inf")
        best_child = None
        for child in node.children:
            score = child.score / child.visits + UCT_CONSTANT * math.sqrt(
                math.log(node.visits) / child.visits
            )
            if score > best_score:
                best_score = score
                best_child = child
        return best_child

    def rollout(self, node):
        if random.random() > 0.5:
            return 1
        else:
            return 0

    def backpropagate(self, node, reward):
        while node is not None:
            node.visits += 1
            node.score += reward
            node = node.parent

    def best_child(self, node):
        best_score = float("-inf")
        best_child = None
        for child in node.children:
            score = child.score / child.visits
            if score > best_score:
                best_score = score
```

```
                best_child = child
        return best_child.action

# Create the root node of the game tree
root_node = Node()

# Create the MCTS object and perform a search
mcts = MCTS()
best_action = mcts.search(root_node)

print("Best action:", best_action)
```

In this example, the Node class represents a single node in the game tree, and the MCTS class performs the search. The select, rollout, and backpropagate methods are the core components of the MCTS algorithm, and are used to select a node to expand, simulate a possible outcome, and update the scores of the nodes along the path from the selected node to the root, respectively.

This is just a very simple example, and in practice, the implementation of MCTS can become much more complex, depending on the nature of the game being played and the specific goals of the AI system. But I hope this helps to give you a sense of how MCTS can be used in game AI.

7 SIMULATIONS AND REAL TIME STRATEGY GAMES

The role of Real Time Strategy (RTS) games in military simulations is little acknowledged or known outside of military research initiatives. However, RTS is used extensively for simulations in for instance the United States Air Force (Gruber 2015). Real-Time Strategy Games are defined as:

> ...Real-Time Strategy games, which are essentially simplified military simulations. In an RTS game, a player indirectly controls many units and structures by issuing orders from an overhead perspective (figure 1) in "real-time" in order to gather resources, build an infrastructure and an army, and destroy the opposing player's forces. The real-time aspect comes from the fact that players do not take turns, but instead may perform as many actions as they are physically able to make, while the game simulation runs at a constant frame rate (24 frames per second in StarCraft) to approximate a continuous flow of time. Some notable RTS games include Dune II, Total Annihilation, and the Warcraft, Command & Conquer, Age of Empires, and StarCraft series'. Generally, each match in an RTS game involves two players starting with a few units and/or structures in different locations on a two-dimensional terrain (map). Nearby resources can be gathered in order to produce additional units and structures and purchase upgrades, thus gaining access to more advanced in-game technology (units, structures, and upgrades). Additional resources and strategically important points are spread around the map, forcing players to spread out their units and buildings in order to attack or defend these positions. Visibility is usually limited to a small area around player-owned units, limiting information and forcing players to conduct reconnaissance in order to respond effectively to their opponents. In most RTS games, a match ends when one player (or team) destroys all buildings belonging to the opponent player (or team)
>
> (Robertson, Watson, 2014, 1)

There are several areas of major activity in an RTS game for instance creating forts or battle installations, resource management, and an area that is highlighted in terms of AI Agents playing RTS games is that of micromanagement or "micro" controlling the actions of individual units or groups of units in combat. This is an area where an AI bot can outperform a human. Such that even a human with a superior strategy can be out done by AI agents, since such issues as miskeying are not present to the extent they are with a human player. This is fairly usual, in basic things, an AI can well outperform a human, but in large scale structures and open world parameters the AI struggles to outduel a human gamer.

There has been a major tradition in RTS game world of conducting competitions to create the best AI bots to play each other. A recent advantage has been made by DeepMind in their RTS bot that has beaten human competitors, while many critique this as a phenomenon of micromanagement, it cannot be ignored that DeepMind has made great strides in its ability to realistically play like a human and beat humans. They have accomplished this through the use of Deep Learning. While other strategies have been based in more traditional Game AI techniques. The difficulty in RTS is that these computational problems are considered NP Hard, that is computationally long and most times unsolvable on classical architecture, although this supposition is open to question with DeepMind's success. Often, what we think on paper according to math is not correct, such as this supposition.

Capt. Gruber in creating his AI bot uses the Multi-Objective Evolutionary Algorithm (MOEA) method, which is a genetic algorithm approach. Genetic programming techniques are a fairly common answer to AI bot functioning (Gruber 2015, 4). The strategy planning of an RTS game is one of the more important elements in the simulation.

> The strategic planning of an RTS game covers the same objectives as a standard military confrontation. In most RTS games the objective is complete destruction of the enemy, with some versions describing victory as the elimination of any manufacturing ability or the conquest of a specific portion of the game map. These goals are accomplished through the development and application of a series of construction actions, otherwise known as build order. The build-order is responsible for moving a player through various technological development stages and can improve a player's unit construction ability. For example, a technical level of 1 may allow a player to build basic infantry, level 2 may introduce more advanced unit types such as flamethrowers or heavy machine guns. Level 3 may build off this further and allow the player to begin building larger assets such as tanks or other vehicles. (Gruber 2015, 22)

The difference between strategy, an overall game approach, and tactics, an more modular and goal oriented sequence of actions is something to consider. Strategy and Tactics decision making have overall architectures presented below:

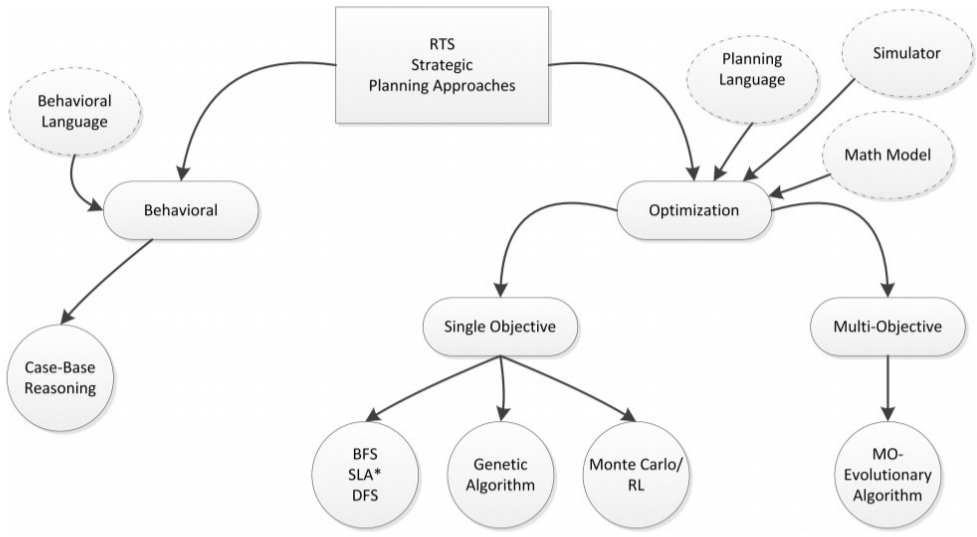


Figure 5. RTS Strategic Planning Tree [5]

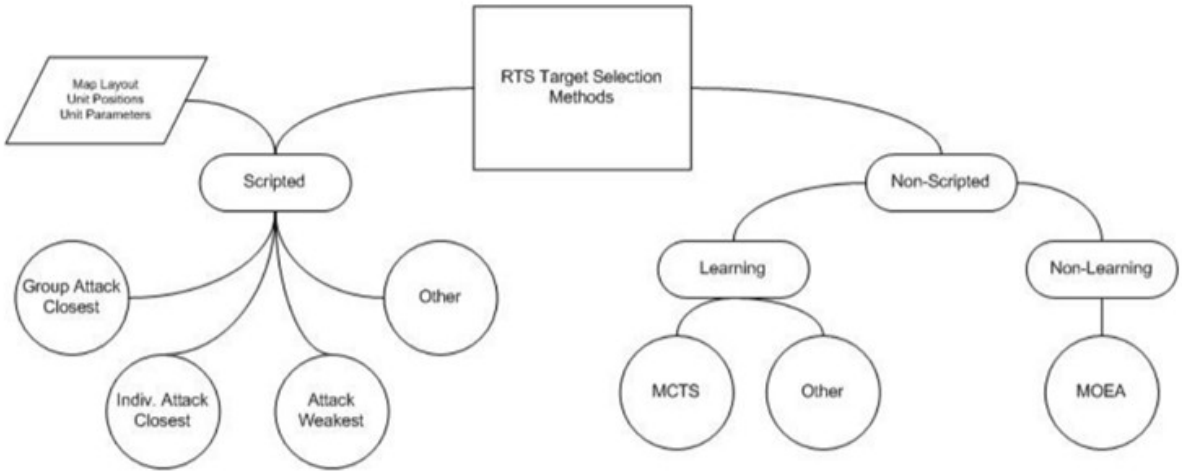


Figure 6. RTS Tactical Planning Tree

(Gruber 2015)

Managing strategy and tactics is one of the key questions in creating an effective bot in RTS games. One approach is that of the creators of *Eisbot*, it's creator Weber et al has explained their approach:

> Our approach for managing the complexity of StarCraft is to partition gameplay into domains of competence and develop interfaces between these competencies for resolving goal conflicts. It is based on the integrated. Agent framework proposed by McCoy and Mateas (2008), which partitions the behaviors in a reactive planning agent into managers which specialize in distinct aspects of gameplay. The decomposition of gameplay into managers is based on analysis of expert gameplay. (Weber 2011, 329)

To see the complexity in these games, the managers used to carry out the Robot strategy of *Eisbot* are:

> -Strategy Manager: responsible for the strategy selection and attack timing competencies
> -Income manager: handles worker units, resource collection and expanding
> -Construction manager: responsible for managing requests to build structures.
> -Tactics manager: performs combat tasks and micromanagement behaviors
> -Recon manager: implements scouting behaviors. (Weber 2011, 331)

Other methods used for managing RTS games with AI Agents have been based on such as previously covered Bayesian Learning, Behavior Trees and Goal-Oriented Action Planning. Robertson and Watson give other means to build AI bots in RTS games. Neuro-evolution techniques which they say:

> Neuroevolution is a technique that uses an evolutionary algorithm to create or train an artificial neural network. Gabriel, Negru, and Zaharie (2012) use a neuroevolution approach called rtNEAT to evolve both the topology and connection weights of neural networks for individual unit control in *StarCraft*. In their approach, each unit has its own neural network that receives input from environmental sources (such as nearby units or obstacles) and hand-defined abstractions (such as the number, type, and "quality" of nearby units), and outputs whether to attack, retreat, or move left or right. During a game, the performance of the units is evaluated using a hand-crafted fitness function and poorly performing unit agents are replaced by combinations of the best-performing agents. It is tested in very simple scenarios of 12 versus 12 units in a square arena, where all units on each side are either a hand-to-hand or ranged type unit. In these situations, it learns to beat the built in StarCraft AI and some other bots. However, it remains unclear how well it would cope with more units or mixes of different unit types (Gabriel, Negru, and Zaharie 2012). (Robertson, Watson, 2014, 6)

Neuro-evolution being based on Genetic programming techniques it uses a fitness function to determine the best agent composition to deploy in the game. Another approach is that of Case-Based Planning (CBP):

> CBP is a planning technique which finds similar past situations from which to draw potential solutions to the current situation. In the case of a CBP system, the solutions found are a set of potential plans or sub-plans which are likely to be effective in the current situation. CBP systems can exhibit poor reactivity at the strategic level and excessive reactivity at the action level, not reacting to high-level changes in situation until a low-level action fails, or discarding an entire plan because a single action failed.(Robertson, Watson, 2014, 6)

CBP is also used in systems such as the Shadow AI in '*Killer Instinct'* which mimics the actions of the human gamer and incorporates it into the actions of the AI opponent, similar to learning from demonstration. Of course, it is apparent that one stumbling block of this is that new situations will have no effective model, perhaps creating a misfit logjam. Hierarchical Planning is yet another method used for AI bot optimization:

> An alternative means of hierarchical planning was used by Weber et al. (2010). They use an active behavior tree in A Behavior Language, which has parallel, sequential and conditional behaviors and goals in a tree structure (figure 6) very similar to a behavior tree (see section 3.3). However, in this model, the tree is expanded during execution by selecting behaviors (randomly or based on conditions or priority) to satisfy goals, and different behaviors can communicate indirectly by reading or writing information on a "shared whiteboard". Hierarchical planning is often combined as part of other methods, such as how Ontanon et al. (2007) use a hierarchical CBP system to reason about goals and plans at different levels. (Robertson, Watson, 2014, 7)

One way to address the issue of high and low-level reactivity in CBPs is to use a Goal-Driven Autonomy model in which an agent reasons about goals, realizes when they need to be updated, and changes or adds to

them as needed for subsequent planning and execution, this actively reasons and reacts to why a goal is succeeding or failing (Robertson, Watson, 8).

All this comes to a head in the development by DeepMind of the *AlphaStar* system to use AI to play an RTS game and defeat human players. *AlphaStar* is a neural network-based artificial intelligence (AI) program designed to play *StarCraft II*, a complex and highly popular real-time strategy game. The neural network architecture of *AlphaStar* is based on a combination of supervised and reinforcement learning. The supervised learning component involves training the network on a dataset of game replays, which allows it to learn how to recognize game states and make predictions about game outcomes. The reinforcement learning component involves self-play, where the network plays against itself to learn new strategies and refine its gameplay.

The training process for *AlphaStar* was divided into three stages. In the first stage, the network was trained on a simplified version of the game, where it learned basic skills such as moving units and attacking enemies. In the second stage, the network was trained on a full version of the game, where it learned more complex strategies such as resource management and building structures. In the third stage, the network was trained on a version of the game with limited information, which required it to make predictions about game states based on incomplete information.

In 2016, DeepMind released a paper describing a neural network-based AI program called *AlphaStar*, which was designed to play StarCraft II, a complex and highly popular RTS game. *AlphaStar* used a combination of supervised and reinforcement learning (see AI vs Machines below) to improve its performance, and eventually reached a level where it could beat the top 0.2% of human players. *AlphaStar*'s neural network architecture is based on a combination of supervised and reinforcement learning. The supervised learning component involves training the network on a dataset of game replays, which allows it to learn how to recognize game states and make predictions about game outcomes. The network uses a convolutional neural network (CNN) to process the input data, which includes information about the game state, such as the location of units and the amount of resources available. The reinforcement learning component involves self-play, where the network plays against itself to learn new strategies and refine its gameplay.

AlphaStar's success was a significant achievement for DeepMind, as *StarCraft II* is a challenging game for an AI to play. The game requires players to manage many units at once, make decisions quickly, and adapt to a changing environment. The AI had to learn how to balance offense and defense, control multiple units, and develop strategies to defeat opponents. One of the interesting aspects of *AlphaStar*'s development was how it was trained. DeepMind used a technique called self-play, where the AI played against itself to improve its performance. This allowed the AI to learn new strategies and refine its gameplay, without requiring human input. It also enabled the AI to develop strategies that human players may not have considered.

DeepMind's research on RTS games has broader implications beyond the world of gaming. The AI techniques used to develop *AlphaStar* could be applied to other real-world problems, such as optimizing traffic flows or improving supply chain management. AI programs that can adapt to changing environments and learn from experience could provide valuable insights and improve decision-making in many fields. Overall, DeepMind's research on real-time strategy games has demonstrated the potential of AI to learn complex tasks and develop new strategies. *AlphaStar*'s success in playing *StarCraft II* shows that AI

programs can compete at a high level with human players, and self-play provides a powerful tool for improving AI performance. As AI technology continues to advance, we can expect to see more applications of AI in gaming and beyond.

8 AI VS MACHINES

Previously we talked about *AlphaStar* which was an AI system that learned to play the game of *StarCraft* and defeat human players. In this chapter we shall cover the ideals behind AI playing games, rather than just being used to manage game play. We shall also touch on Generative Adversarial Networks in tandem with Reinforcement Learning (RL) a type of machine learning where an AI agent interacts with an environment, receives rewards or penalties for certain actions, and learns to optimize its behavior to maximize its reward. Artificial Intelligence can be used to learn how to play video games through a process known as reinforcement learning. Reinforcement learning is a type of machine learning where an AI agent interacts with an environment, receiving rewards or penalties for certain actions, and learns to optimize its behavior to maximize its reward. The primary reward being scoring points and winning the game, especially in a zero-sum scenario. AI can learn to play video games using reinforcement learning:

Environment Simulation: The first step in using AI to learn to play video games is to simulate the game environment. This involves creating a digital version of the game's rules and mechanics, which the AI agent can interact with to learn how the game works.

AI Agent: Next, an AI agent is created and programmed to learn how to play the game. The agent interacts with the simulated game environment, making decisions and taking actions, and receiving rewards or penalties based on the outcome of those actions. Libraries used for training agents are RLlib, from UC Berkeley, and Stable Baselines.

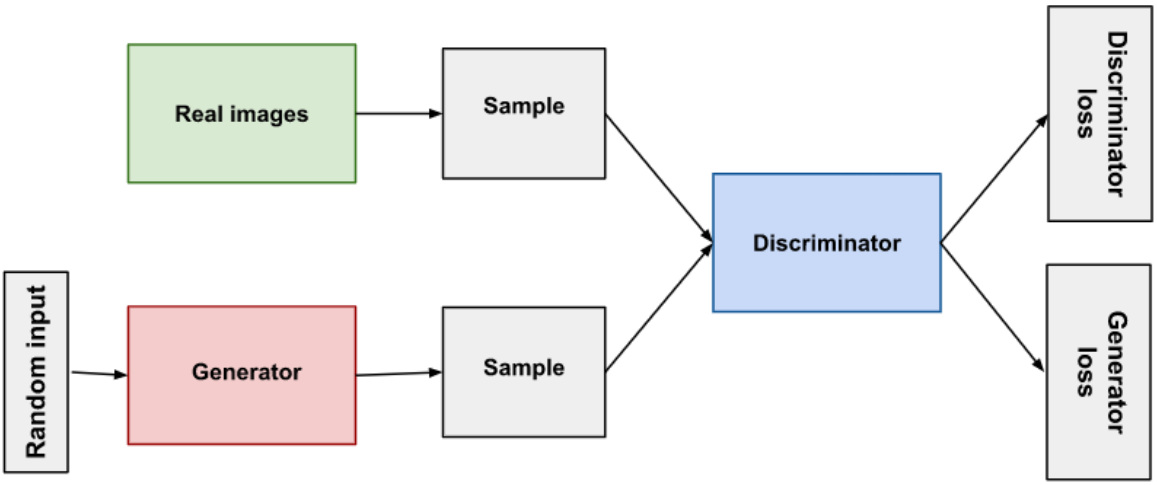
Rewards and Penalties: the AI agent receives rewards or penalties based on its actions in the game environment. These rewards and penalties provide feedback to the agent, allowing it to learn what actions are more likely to lead to success and what actions are less effective.

Optimizing Behavior: Over time, the AI agent uses the rewards and penalties it receives to optimize its behavior and learn how to play the game more effectively. This involves adjusting its decision-making process and learning from its past experiences to make better decisions in the future.

Continuous Learning: The process of reinforcement learning is continuous, allowing the AI agent to continually improve its performance and learn new strategies as it plays the game.

Using reinforcement learning, AI can learn to play video games with increasing proficiency and skill with each iteration of the game, resulting in a more engaging and challenging gameplay experience for the player. This approach is being used in a growing number of video games, including both casual and competitive games, and is leading to new and exciting developments in the field of AI and video games, later we will cover creating RL based video games using Unity3d Game Engine.

Generative adversarial networks (GANs) can be used to teach AI to play video games through a process called "adversarial training". The basic idea behind GANs is to pit two neural networks against each other: a generator network that generates data (in this case, game images), and a discriminator network that tries to distinguish between real and fake data. In the context of video game playing, one way to use GANs is to have the generator network generate images of the game state, and then have the discriminator network try to distinguish between real game state images and generated ones. The generator network then tries to produce game state images that can fool the discriminator into thinking they are real. This process of "adversarial training" helps the generator network learn to generate realistic game state images.



(image: https://developers.google.com/machine-learning/gan/gan_structure)

To use GANs to teach AI to play video games, one approach is to combine GANs with reinforcement learning. In this approach, the generator network produces images of the game state, and the discriminator network tries to distinguish between real and fake game state images. The AI agent then uses the generated images to take actions in the game and receives rewards based on its performance. The rewards are used to update the AI agent's policy, which in turn updates the generator network.

The combination of GANs and reinforcement learning can lead to more efficient and effective training of AI agents for video game playing, as it allows the agent to learn from both the real game state and the generated game state images. However, it is worth noting that GANs can be difficult to train and may require a large amount of computational resources. You will also notice that Visual Recognition is involved here, this type of AI is heavy in terms of Neural Networks.

DeepMind, a subsidiary of Alphabet Inc, is using Artificial Intelligence to teach AI to play games. One example is the use of a GAN to teach an AI agent to play the classic Atari game "*Breakout*". In a 2017

paper titled "*Playing Atari with Deep Reinforcement Learning and GANs*" by M. Gheshlaghi Azar and colleagues, the authors used a GAN to generate synthetic images of the Breakout game state, which were then used to train an AI agent using deep reinforcement learning. The GAN was trained to generate realistic images of the Breakout game state, while the AI agent was trained using the generated images and the real game state images. The AI agent used a policy network that took as input the game state image and output the probability of taking each possible action. The policy network was trained using a combination of the real and generated game state images, along with the corresponding rewards obtained by the agent.

The results showed that the AI agent trained using the GAN-generated images was able to outperform agents trained using only real game state images or randomly generated images. The authors concluded that the use of GANs to generate synthetic images can improve the efficiency and effectiveness of deep reinforcement learning for video game playing.

Here's how DeepMind uses AI to teach AI to play games:

Environment Simulation: DeepMind creates a simulated game environment that the AI agent can interact with. This environment is designed to closely mimic the rules and mechanics of the real-world game.

AI Agent: DeepMind creates an AI agent that is designed to learn how to play the game. The agent is programmed with a set of decision-making algorithms and is trained using reinforcement learning, receiving rewards or penalties based on its actions in the game environment.

Rewards and Penalties: In reinforcement learning, the AI agent receives rewards or penalties based on its actions in the game environment. These rewards and penalties provide feedback to the agent, allowing it to learn what actions are more likely to lead to success and what actions are less effective.

Optimizing Behavior: Over time, the AI agent uses the rewards and penalties it receives to optimize its behavior and learn how to play the game more effectively. This involves adjusting its decision-making process and learning from its past experiences to make better decisions in the future.

Continuous Learning: The process of reinforcement learning is continuous, allowing the AI agent to continually improve its performance and learn new strategies as it plays the game.

Alpha Go

AlphaGo made history in 2016 by defeating Lee Sedol, a world champion Go player, in a five-game match, marking the first time an AI system had beaten a professional human player in the complex and ancient Chinese board game of Go. Go is a game that requires strategic thinking, pattern recognition, and intuition. Unlike games like chess, where the number of possible moves is limited, the number of possible moves in Go is vast, making it much more challenging for AI systems to play.

Here's how *AlphaGo* works, like *AlphaStar,* three key components:

> Monte Carlo Tree Search: *AlphaGo* uses a Monte Carlo Tree Search algorithm to analyze the game board and make its next move. This algorithm works by simulating thousands of random game outcomes and selecting the move that leads to the best expected outcome.
>
> Deep Neural Networks: *AlphaGo* also uses deep neural networks, which are machine learning models that can recognize patterns in data. *AlphaGo*'s neural networks are trained on a large dataset of human expert Go games, allowing it to learn the strategies and patterns used by top Go players.
>
> Reinforcement Learning: *AlphaGo* uses reinforcement learning, a type of machine learning where an AI agent interacts with an environment and learns to optimize its behavior based on rewards or penalties. In *AlphaGo*'s case, the agent receives rewards for winning games and penalties for losing, allowing it to continually improve its play.

AlphaGo's victory over Lee Sedol was a major milestone in the development of AI and demonstrated the power of machine learning and deep neural networks. It also has implications for fields such as robotics, decision-making, and human-computer interaction, and could help to advance our understanding of how intelligence works.

Today, *AlphaGo* continues to be used by DeepMind for research into AI and machine learning, and its innovations have led to the development of new AI systems that can play a wide range of games and complete complex tasks. *AlphaGo* and *AlphaStar* show how the use of AI to play games can overcome experience and knowledge gaps and even out duel elite human competitors in games involving complete information, and use unique iteration results to plot unheard of strategies as the AI iterates through unfathomable numbers of simulations of the same game state.

The Power of Reinforcement

Reinforcement is a well tested human way of learning, such as memorization of Bible or Qur'an verses, repeated again and again until it sinks into the fibres of the human body and/or mind. Repetition over the same patterns has a deep effect. This deep effect also takes place in such other media as silicon chips, or whizzing electrons, even gravitational waves leave behind a lasting impression on matter it interacts with. So how does Reinforcement work in Machines? This section delves deeper into the subject as we look at how to construct RL algorithms and apply them to video game development.

HuggingFace has a good course on deep reinforcement learning, HF also hosts model zoos, a place were hundreds of models are archived and served up to developers, such as transfer learning, ability to fine-tune pre-existing large models to specific tasks, say a dialogue system for a dungeon game. You can find their reinforcement learning tutorial at https://huggingface.co/blog/deep-rl-intro

Pragati Baheti of Microsoft's V7 Lab gives a succinct account of the elements of RL:

> Agent - Agent (A) takes actions that affect the environment. Citing an example, the machine learning to play chess is the agent.

Action - It is the set of all possible operations/moves the agent can make. The agent makes a decision on which action to take from a set of discrete actions (a).

Environment - All actions that the reinforcement learning agent makes directly affect the environment. Here, the board of chess is the environment. The environment takes the agent's present state and action as information and returns the reward to the agent with a new state.

For example, the move made by the bot will either have a negative/positive effect on the whole game and the arrangement of the board. This will decide the next action and state of the board.

State - A state (S) is a particular situation in which the agent finds itself. This can be the state of the agent at any intermediate time (t).

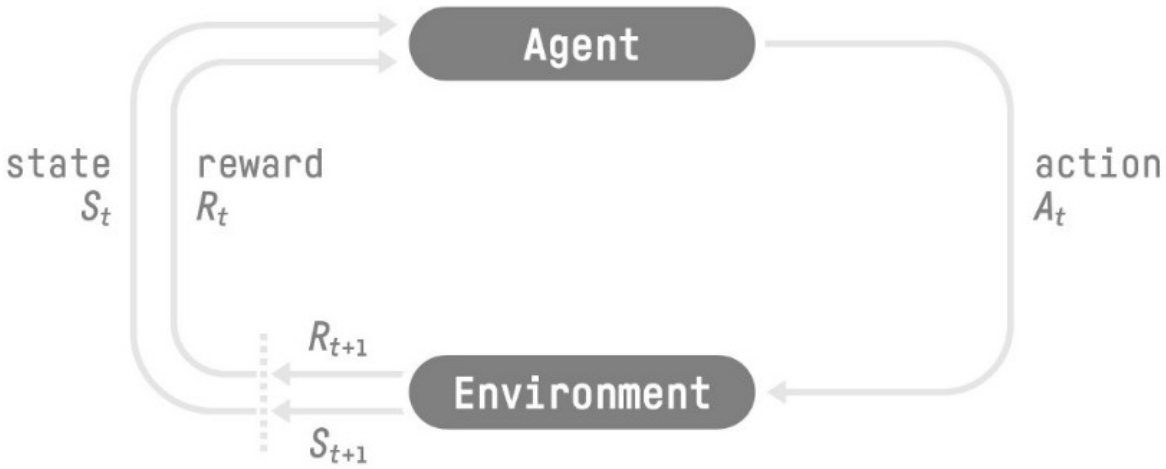
Reward (R) - The environment gives feedback by which we determine the validity of the agent's actions in each state. It is crucial in the scenario of Reinforcement Learning where we want the machine to learn all by itself and the only critic that would help it in learning is the feedback/reward it receives.

Discount factor - Over time, the discount factor modifies the importance of incentives. Given the uncertainty of the future it's better to add variance to the value estimates. Discount factor helps in reducing the degree to which future rewards affect our value function estimates.

Policy (π) - It decides what action to take in a certain state to maximize the reward.

Value (V)—It measures the optimality of a specific state. It is the expected discounted rewards that the agent collects following the specific policy.

Q-value or action-value - Q Value is a measure of the overall expected reward if the agent (A) is in state (s) and takes action (a), and then plays until the end of the episode according to some policy (π).
(Baheti, 2023, https://www.v7labs.com/blog/deep-reinforcement-learning-guide)



(Sutton & Barto 1998)

Below we will go over these elements in more detail. It is always important to understand the history of the development of ideals to scientific questions as usually we can see a progression from one ideal to the next, a continuous evolution to development. The history of the development of RL is considered presently.

A Timeline of RL (see Sutton & Barto 1998, Ch. 1.6):

1950s – cybernetics comes into real form after being developed by Norbert Wiener, questions of controlling radar systems developed into the science of Operations Research The question of 'optimal control' came into use, Richard Bellman used the concepts of a dynamical system's state and of a value function or 'optimal return function' which became Bellman equation, turning into dynamic programming in 1957, the. This also led to the advent of Markov Decision processes (MDPs), which in 1960 lead Ron Howard to devise the policy iteration method for MDPs.

$$V_\pi(s) = \mathbb{E}_\pi[G_t|s = s_t] = \mathbb{E}_\pi\left[\sum_{j=0}^{T} \gamma^j r_{t+j+1} \,|s = s_t\right]$$

Bellman Value Equation, A value function estimates how good it is for the Agent to be in a given state (or how good it is to perform a given action in a given state

1950 – Claude Shannon suggest that a computer could be programmed to use an evaluation function to play chess, and then adapt it's code to maximize the reward of winning.

1954 -- Minsky, Farley and Clark, researchers began exploring the ideal of trial-and-error learning as an engineering principle. Minsky came up with Stochastic Neural-Analog Reinforcement Calculators (SNARCs). In the 1960s the term reinforcement learning entered engineering literature in Minsky's paper "Steps Toward Artificial Intelligence in 1961.

1959 – Arthur Samuel creates a learning method that includes temporal-difference ideas: the probability of winning tic-tac-toe by the difference between temporally successive estimates of the same quantity,

which lay in animal learning psychology-such as Durov and Kazhinsky in the early Soviet Union-specifically the concept of secondary reinforcers, a stimulus that has been paired with a primary reinforcer such as food or pain.

1961-3 – Donald Michie described a simple trial-and-error learning system for learning how to play tic-tac-toe called Matchbox Educable Naughts and Crosses Engine (MENACE). In 1968, along with Chambers, he developed Game Learning Expectimaxing Engine (GLEE), which was an early example of a reinforcement learning task under conditions of incomplete knowledge. Michie emphasizes trial and error and learning as essential parts of AI.

1973 – Widrow, Gupta and Maitra developed a reinforcement learning rule that could learn from success and failure signals, known as 'selective bootstrap adaptation' based on the earlier work of Widrow and Hoff in 1960 they termed as 'learning with a critic', presaging actor-critic architecture of later RL.

Tsetlin developed learning automata in Russia, methods for solving a nonassociative, purely selectional learning problem known as the "one-armed bandit"

1975 – John Holland, genetic algorithm creator developed a general theory of adaptive systems based on selectional principles. Later, 1986, he introduced a classifier system, true RL systems including association and value functions.

1977 – Ian Witten published on temporal-difference learning rule, proposing tabular TD(0) as part of an adaptive controller for solving MDPS.

1978 – Sutton develops Klopf's ideas, in particular links to animal learning theories, driven by changes in temporally successive predictions. With Barto he developed a psychological model of classical conditioning based on temporal-difference learning.

1982 – Klopf focuses on hedonic aspects of behavior, the drive to achieve some result from the environment, to control the environment toward desired ends and away from undesired ends, the ideal of trial-and-error learning.

1988 – Sutton separated temporal-difference learning from control, making it a general prediction method, he also introduced the TD algorithm and proved some if its convergence properties.

1989 – temporal-difference and optimal control threads were fully broght together in 1989 with Q-Learning developed by Chris Watkins.

1992 – Gerry Tesauro created a backgammon playing program TD-Gammon.

Although reinforcement learning related research in the Soviet Union is not known widely in the west, such as programming Reflexive Control by the Soviet Military, the Soviet research has parallels with western RL research, as well as Generative Adversarial Networks (See McCarron 2023).

The reward hypothesis: the central idea of Reinforcement Learning

The reward hypothesis: The idea that the objective of an agent in a reinforcement learning problem is to learn a policy that maximizes the cumulative sum of rewards it receives over time. The reward hypothesis is a fundamental concept in reinforcement learning (RL) that states that an agent's goal is to learn a policy that maximizes a scalar reward signal over time. In other words, the agent's objective is to choose actions that lead to the highest possible cumulative reward.

The reward hypothesis is important in RL for several reasons:

Defines the agent's objective: The reward signal provides the agent with a clear objective to optimize. By maximizing the reward signal, the agent learns to choose actions that lead to desirable outcomes and avoid actions that lead to negative outcomes.

Provides feedback to the agent: The reward signal provides feedback to the agent on the quality of its actions. The agent learns from the reward signal whether a particular action was good or bad, and uses this information to adjust its behavior accordingly.

Determines the optimal policy: The reward signal is used to evaluate the quality of different policies. The agent learns to choose the policy that leads to the highest cumulative reward over time.

Enables transfer learning: The reward signal is often domain-specific and can be designed to capture the specific objectives of a given task. This enables transfer learning, where an agent can learn to solve a new task by adapting its existing policy to a new reward signal.

Discount Factor

A discount factor is a parameter used in reinforcement learning algorithms that determines the relative importance of immediate versus future rewards. It is denoted by the symbol γ (gamma) and typically takes a value between 0 and 1. The discount factor is used to discount the value of future rewards, making them less important than immediate rewards.

In reinforcement learning, the goal of an agent is to learn a policy that maximizes its cumulative reward over time. The cumulative reward is often expressed as a sum of discounted rewards, where each reward is multiplied by the discount factor raised to the power of the number of time steps between the reward and the present.

Here's how the discount factor interacts with rewards in a reinforcement learning algorithm:

Immediate rewards: Rewards that are received immediately have a discount factor of 1. This means that their value is not discounted, and the agent values them equally to future rewards.

Future rewards: Rewards that are received in the future have a discount factor less than 1. This means that their value is discounted, and the agent values them less than immediate rewards. The degree of discounting depends on the value of the discount factor.

Cumulative rewards: The agent's goal is to maximize its cumulative reward over time, which is calculated as the sum of discounted rewards. The discount factor determines how much weight is given to future rewards relative to immediate rewards. A higher discount factor values future rewards more highly, whereas a lower discount factor places more emphasis on immediate rewards.

The discount factor, γ (gamma), is an important hyperparameter in reinforcement learning algorithms, and tuning it can significantly impact the agent's performance. Here are some methods that can be used to tune the gamma parameter:

Manual tuning: One common method for tuning the discount factor is manual tuning, where the value of γ is set by trial and error. The agent is trained with different values of γ and evaluated to determine which value yields the best performance. This method can be time-consuming, but it can be effective if the range of possible values is small.

Grid search: Grid search is a method for systematically exploring the range of possible values for γ. The range of possible values is divided into a grid, and the agent is trained with each combination of values on the grid. The performance of the agent is evaluated for each combination, and the value of γ that yields the best performance is selected.

Random search: Random search is similar to grid search, but instead of exploring all possible combinations of values, the agent is trained with a random selection of values for γ. This method can be more efficient than grid search, especially if the range of possible values is large.

Bayesian optimization: Bayesian optimization is a method that uses a probabilistic model to estimate the performance of the agent for each value of γ. The model is updated as the agent is trained, and it is used to guide the selection of the next value of γ to try. This method can be very efficient and effective for tuning hyperparameters.

Reinforcement learning: In some cases, it is possible to use reinforcement learning to tune the discount factor. The agent is trained with a range of values for γ, and the value of γ is treated as an additional parameter to be learned. The agent learns the optimal value of γ as part of the reinforcement learning process.

Markov Property

The Markov property states that the future state of a system depends only on the current state and not on any previous states. In reinforcement learning, this property is used to model the environment as a Markov decision process.

The Markov property, also known as the Markov assumption or Markovian assumption, is a key concept in probability theory and stochastic processes. It states that the future state of a system depends only on its present state, and not on any of its past states. In other words, the future of a system is independent of its history given its present state. More formally, a stochastic process has the Markov property if the probability of the next state of the system, given its current state, is independent of all its previous states. This can be expressed mathematically as:

$$P(X_{n+1} \mid X_n, X_{n-1},..., X_1) = P(X_{n+1} \mid X_n)$$

where X_1, X_2, ..., X_n are the previous states of the system, and X_{n+1} is the next state.

The Markov property is a fundamental assumption in many areas of applied mathematics, physics, engineering, computer science, and economics, and is used to model a wide range of real-world phenomena, including the behavior of financial markets, the spread of infectious diseases, the performance of communication networks, and the behavior of natural language.

The Markov property is a fundamental assumption in reinforcement learning (RL). In RL, an agent interacts with an environment and learns how to take actions that maximize a cumulative reward signal. The Markov property is important in RL because it enables the agent to make decisions based on the current state of the environment, rather than having to consider the entire history of past states and actions.

Specifically, in RL, the Markov property is used to define the Markov decision process (MDP), which is a mathematical framework used to model sequential decision-making problems. An MDP consists of a set of states, actions, rewards, and a transition function that defines the probability of transitioning from one state to another when taking a specific action. The MDP framework assumes that the environment is Markovian, meaning that the current state of the environment is sufficient to determine the probability of transitioning to any other state. This allows the agent to use a state-value function or a state-action value function to estimate the value of each state or state-action pair, which is then used to make decisions about which action to take in each state.

The Markov property is also important in RL algorithms such as Q-learning and policy iteration, which rely on the assumption that the environment is Markovian. These algorithms use the estimated state or state-action values to learn a policy that maximizes the cumulative reward signal over time, while taking into account the stochasticity of the environment.

A code example of a MDP in RL:

```
import gym

# Define the environment
env = gym.make('FrozenLake-v0')

# Define the MDP
states = env.observation_space.n
actions = env.action_space.n

# Define the transition function
def transition_function(state, action):
    transitions = env.P[state][action]
    next_states = [trans[1] for trans in transitions]
    probs = [trans[0] for trans in transitions]
    return next_states, probs

# Define the reward function
def reward_function(state, action, next_state):
    return env.P[state][action][0][2]

# Define the discount factor
gamma = 0.99
```

```
# Define the initial state
initial_state = env.reset()

# Define the terminal states
terminal_states = [5, 7, 11, 12, 15]

# Define the MDP tuple
mdp = (states, actions, transition_function, reward_function, gamma, initial_state,
terminal_states)
```

In this example, we first import the gym library and create an instance of the FrozenLake-v0 environment. We then define the MDP by specifying the number of states and actions, as well as the transition function, reward function, discount factor, initial state, and terminal states. The transition function takes as input a state and action, and returns a list of next states and the corresponding transition probabilities. The reward function takes as input a state, action, and next state, and returns the reward associated with transitioning from the current state to the next state. Finally, we define the MDP tuple that encapsulates all the relevant information about the MDP. This MDP can then be used to implement various RL algorithms, such as value iteration or policy iteration, to learn an optimal policy for the given environment.

Non-Markovian processes come into play in game design as well. A non-Markovian environment is a type of environment where the future state of the environment depends not only on the current state but also on the entire history of past states and actions. In other words, the Markov property does not hold in non-Markovian environments. Non-Markovian environments are also called partially observable environments or history-dependent environments. In such environments, the agent cannot fully observe the state of the environment, making it difficult to determine the optimal action to take based solely on the current state. The agent needs to maintain a memory or history of past observations to make decisions about the future, which can be challenging in practice. Examples of non-Markovian environments include games with hidden information, such as poker or blackjack, where players' cards are hidden from one another. In these games, the current state of the game is not sufficient to determine the probability of transitioning to a future state, and players need to remember the past actions and observations to make optimal decisions. Another example is natural language processing, where understanding a sentence often requires knowledge of the context and previous sentences. In such cases, the current sentence alone is not sufficient to understand the meaning of the entire text.

There are also some video games that are based in retro causality. a non-causal environment is one in which the current state depends on future events, which is in contrast to Markovian and non-Markovian environments where the future state depends on the current and past states. While non-causal environments are theoretically possible, they are rare in practice and are more commonly encountered in science fiction and other forms of speculative fiction. For example, the "Prince of Persia: The Sands of Time" game series features a time-rewinding mechanic that allows players to undo their mistakes, effectively altering the future based on actions taken in the present. Similarly, the "Life is Strange" game series features a mechanic where the player's choices can have consequences that affect future events and outcomes, blurring the line between cause and effect.

Another example is the game "Braid," where the player character has the ability to manipulate time, allowing them to rewind, pause, or fast-forward time. The game's puzzles are designed around these time-

manipulation mechanics, creating a non-causal environment where actions taken in the present can affect the future.

Observations/States Space

The set of all possible observations or states that the agent can perceive or be in, respectively. The set of all possible observations or states that the agent can perceive or be in is known as the state space of the environment. The state space is a crucial aspect of an MDP as it determines the agent's perception of the environment and its ability to make decisions based on that perception. The state space is also important because it determines the complexity of the problem. If the state space is large or infinite, then finding an optimal policy can be computationally expensive or even impossible. In such cases, dimensionality reduction techniques, such as feature extraction or approximation, can be used to reduce the complexity of the problem.

Furthermore, the state space also affects the agent's ability to learn a good policy. If the agent cannot observe certain aspects of the environment, the environment is said to be partially observable or non-Markovian, and learning an optimal policy can be more challenging. In such cases, the agent needs to maintain a belief state or a memory of past observations to make decisions about the future.

Feature extraction is a technique used in reinforcement learning to reduce the dimensionality of the state space of an environment. In an MDP, the state of the environment is typically represented as a vector of features that describe the relevant aspects of the environment that the agent can observe. Feature extraction involves selecting or transforming these features to create a new, smaller set of features that better captures the most relevant information about the environment. The goal of feature extraction is to reduce the dimensionality of the state space while still preserving enough information to allow the agent to learn an optimal policy.

One common technique for feature extraction is to use domain knowledge to select a subset of features that are most relevant for the task at hand. For example, in a game of chess, relevant features might include the positions of the pieces on the board, the number of moves made by each player, and the history of previous moves. By selecting only these relevant features, the state space can be significantly reduced, making the problem more tractable for the agent. Another technique for feature extraction is to use dimensionality reduction methods such as principal component analysis (PCA) or linear discriminant analysis (LDA). These methods transform the original feature vector into a new, smaller feature vector while preserving as much of the information as possible. Deep learning methods, such as convolutional neural networks (CNNs) or recurrent neural networks (RNNs), can also be used for feature extraction. These methods can learn a hierarchy of features that capture increasingly abstract representations of the environment, allowing for more efficient and effective learning.

Here's an example of how to use principal component analysis (PCA) to reduce the dimensionality of a dataset in Python:

```
import numpy as np
from sklearn.decomposition import PCA

# Generate a random dataset with 3 features and 100 samples
X = np.random.rand(100, 3)
```

```
# Create a PCA object with 2 components
pca = PCA(n_components=2)

# Fit the PCA object to the dataset
pca.fit(X)

# Transform the dataset into the new, reduced feature space
X_pca = pca.transform(X)

# Print the shape of the original dataset and the transformed dataset
print('Original dataset shape:', X.shape)
print('Transformed dataset shape:', X_pca.shape)
```

In this example, we first generate a random dataset with 3 features and 100 samples using NumPy. We then create a PCA object with 2 components and fit it to the dataset using the fit method. Finally, we transform the dataset into the new, reduced feature space using the transform method.

The output of this code should show the shape of the original dataset ((100, 3)) and the transformed dataset ((100, 2)), which has been reduced from 3 features to 2 features. Note that the PCA object has automatically selected the 2 most important features for us based on their variance in the dataset.

Action Space

The action space is the set of all possible actions that an agent can take in a given environment. The agent's goal is to learn a policy that maps states of the environment to actions that maximize some notion of reward.

The action space can be discrete or continuous, depending on the nature of the environment. In a discrete action space, the agent can take a finite number of distinct actions, such as moving up, down, left, or right in a grid world. In a continuous action space, the agent can take any action within a continuous range, such as selecting a steering angle or throttle value in a self-driving car.

The choice of action space is important in reinforcement learning, as it can greatly affect the complexity of the problem and the agent's ability to learn an optimal policy. A large or continuous action space can make the problem more challenging, as the agent needs to search a large space to find the best action. In such cases, techniques such as function approximation or policy gradient methods can be used to learn an optimal policy.

Moreover, the choice of action space is also dependent on the goals of the task. For example, in a game of chess, the action space is the set of possible moves, and the goal is to win the game. In contrast, in a self-driving car, the action space may be the steering angle and throttle values, and the goal is to reach the destination safely and efficiently.

Function approximation in reinforcement learning refers to the process of approximating an unknown function that maps states to actions or values, using a set of basis functions or a neural network. This is often necessary when the state or action space is too large or continuous to be represented explicitly.

Here is a pseudocode, only for illustration purposes, Python example of using function approximation with a simple linear regression model to approximate a value function for a grid world environment:

```
import numpy as np
from sklearn.linear_model import LinearRegression

# Define the grid world environment and the reward structure
GRID_SIZE = 4
START_STATE = (0, 0)
GOAL_STATE = (GRID_SIZE-1, GRID_SIZE-1)
REWARD = 1.0

# Define the state space and the action space
state_space = [(i, j) for i in range(GRID_SIZE) for j in range(GRID_SIZE)]
action_space = ['up', 'down', 'left', 'right']

# Define a function to extract features from the state
def extract_features(state):
    return np.array(state)

# Create a value function approximation model using linear regression
model = LinearRegression()

# Generate training data by simulating the environment
X = []
y = []
for state in state_space:
    features = extract_features(state)
    for action in action_space:
        next_state = get_next_state(state, action)
        reward = REWARD if next_state == GOAL_STATE else 0.0
        next_features = extract_features(next_state)
        X.append(features)
        y.append(reward + np.max(model.predict(next_features.reshape(1, -1))))

# Train the model on the training data
model.fit(X, y)

# Use the trained model to find the optimal policy
policy = {}
for state in state_space:
    features = extract_features(state)
    q_values = model.predict(features.reshape(1, -1))
    action_idx = np.argmax(q_values)
    policy[state] = action_space[action_idx]
```

In this example, we first define a grid world environment with a specified size, start state, goal state, and reward structure. We then define the state space and the action space, and a function to extract features from the state (in this case, just the state itself). We use linear regression as the function approximation model to approximate the value function. We generate training data by simulating the environment for all possible state-action pairs, and compute the expected reward for each pair using the model's prediction of the next state's value. We then train the model on this training data using the fit method. Finally, we use the trained model to find the optimal policy by computing the Q-values for each state-action pair and selecting the action with the highest Q-value. We store this policy in a dictionary for later use.

Type of tasks

Reinforcement learning problems can be classified into different types based on the presence of a known model of the environment and the availability of feedback from the environment. The types of tasks in reinforcement learning (RL) are significant because they represent different types of problems that require different approaches and techniques. Understanding the task type is important for selecting an appropriate RL algorithm and designing an effective solution.

The main types of tasks in RL are:

Episodic tasks: In episodic tasks, the agent interacts with the environment for a finite number of steps, and each interaction is called an "episode". At the end of each episode, the environment resets to a starting state, and the agent starts a new episode. Examples of episodic tasks include playing a single game of chess or navigating through a maze.

Continuing tasks: In continuing tasks, the agent interacts with the environment for an indefinite number of steps, without any specific end point or goal. The agent's goal is to maximize its reward over time. Examples of continuing tasks include controlling a robot to maintain its balance or optimizing a supply chain system.

Exploration vs. Exploitation tasks: In exploration vs. exploitation tasks, the agent must balance its desire to exploit its current knowledge to maximize immediate rewards with its need to explore the environment to learn more about the optimal policy. Examples of such tasks include stock market investments or online advertising.

Partially observable tasks: In partially observable tasks, the agent does not have complete information about the environment. The agent receives "observations" that are partial, noisy, or delayed representations of the environment state. Examples of such tasks include playing poker or navigating in a dark environment.

Multi-agent tasks: In multi-agent tasks, there are multiple agents interacting with each other in the same environment. The agents' policies may depend on the actions of other agents, and the agents may have different goals or rewards. Examples of such tasks include coordination of a group of robots or playing multi-player games.

By understanding the type of task, we can choose the appropriate RL algorithm, tune its hyperparameters, and design a suitable reward function for the agent. This can greatly improve the agent's performance and its ability to learn an effective policy.

Exploration to Exploitation tradeoff

The balance between exploring new actions and exploiting known good actions in order to maximize the expected reward. In reinforcement learning, an agent's goal is to maximize its expected reward over time. To achieve this goal, the agent must choose actions that will lead to the highest reward. However, in order

to find the best actions, the agent may need to explore different possibilities and try new actions that it has not tried before. The exploration-exploitation tradeoff is the balance between exploring new actions and exploiting known good actions. Exploration is the process of taking actions that the agent is uncertain about, in order to learn more about the environment and discover potentially better actions. Exploitation, on the other hand, is the process of taking actions that the agent believes will result in the highest reward, based on its current knowledge of the environment.

If the agent only explores and never exploits, it will never be able to make the best decisions based on the current knowledge it has. On the other hand, if the agent only exploits and never explores, it may miss out on potentially better actions and may not find the optimal policy. Therefore, the agent must find the right balance between exploration and exploitation. The exploration-exploitation tradeoff can be controlled through the agent's policy, which specifies how the agent chooses actions based on the current state. A common approach is to use an epsilon-greedy policy, which selects the action that maximizes the expected reward with probability 1-epsilon, and selects a random action with probability epsilon. The value of epsilon determines the level of exploration: a higher value of epsilon encourages more exploration, while a lower value of epsilon encourages more exploitation.

Value-based and policy-based methods

Value-based and policy-based methods are two broad categories of algorithms used in reinforcement learning.

Value-based methods aim to learn the value function of the optimal policy, which represents the expected cumulative reward that an agent can achieve from a given state under a given policy. Examples of value-based methods include Q-learning, SARSA, and Deep Q-Networks (DQNs). In these methods, the agent learns an estimate of the optimal Q-values, which are the expected reward for taking a particular action in a given state and following the optimal policy thereafter.

Policy-based methods aim to learn the optimal policy directly, without explicitly estimating the value function. Examples of policy-based methods include REINFORCE, Actor-Critic, and Proximal Policy Optimization (PPO). In these methods, the agent learns a parameterized policy function that maps states to actions, and updates the parameters using gradient-based optimization to maximize the expected cumulative reward.

In practice, the choice between value-based and policy-based methods depends on the characteristics of the problem being solved. Value-based methods are typically better suited for problems with large state spaces and discrete action spaces, while policy-based methods are more appropriate for problems with continuous action spaces or stochastic policies.

Hybrid methods that combine value-based and policy-based methods, such as Actor-Critic methods, are also commonly used in reinforcement learning. These methods aim to capture the benefits of both approaches, by learning both a value function and a policy function simultaneously.

The Policy π: the agent's neuron collection
The agent's strategy for selecting actions based on the current state of the environment. a policy is a function that maps a state of the environment to a probability distribution over actions. The policy defines the behavior of an agent at each state, specifying which action to take in order to maximize the expected reward.

Formally, a policy can be represented as follows:

$$\pi(a|s) = P[A_t = a \mid S_t = s]$$

where π is the policy, a is the action, s is the state, and P[A_t = a | S_t = s] is the probability of taking action a in state s at time step t. This can be represented in code as:

```
# Assuming that we have already defined the current state s and action space A
prob_a_given_s = {a: 0 for a in A}
for a in A:
   prob_a_given_s[a] = pi[a][s]  # `pi` is the policy

# The probability of selecting action `a` in state `s`
P_a_given_s = prob_a_given_s[a]
```

There are two main types of policies in reinforcement learning: deterministic policies and stochastic policies. Deterministic policies select a single action for each state, while stochastic policies select actions according to a probability distribution. The choice of policy has a significant impact on the performance of the reinforcement learning algorithm, and different algorithms are often designed to work with different types of policies.

Proximal Policy Optimization (PPO) is a popular policy-based reinforcement learning algorithm that aims to learn the optimal policy of an agent by iteratively updating the policy based on the observed rewards. PPO is a variant of the Trust Region Policy Optimization (TRPO) algorithm, which seeks to maximize the expected reward of the agent while constraining the change in the policy to a small, predefined region. However, TRPO can be computationally expensive, since it requires solving a large optimization problem at each iteration. PPO improves on TRPO by simplifying the optimization problem, using a clipped surrogate objective to bound the change in the policy. At each iteration, PPO samples trajectories by executing the current policy in the environment, and uses these trajectories to compute an estimate of the policy gradient. The policy is then updated by taking a step in the direction of the gradient, subject to a clipping constraint that limits the change in the policy.

The clipped surrogate objective used in PPO is defined as follows:

$$L\_CLIP = \min(r\_t(\theta) * A\_t, \text{clip}(r\_t(\theta), 1 - \varepsilon, 1 + \varepsilon) * A\_t)$$

where r_t(θ) is the ratio of the probability of the new policy to the probability of the old policy, A_t is the advantage function, and ε is a hyperparameter that controls the clipping range. The clipped surrogate objective encourages the policy to move in the direction of higher rewards, while constraining the change to a small region defined by the clipping range.

PPO has several advantages over other policy-based methods, such as its simplicity, scalability, and ability to handle both discrete and continuous action spaces. PPO has been used successfully in a wide range of applications, including game playing, robotics, and autonomous driving.

Reward function

The function that maps each state-action pair to a numerical reward signal. In Reinforcement Learning, the reward function is a function that maps each state-action pair to a numerical reward signal. It is a key component of the RL framework, as it defines the goal of the agent's task. The agent's objective is to maximize the cumulative sum of rewards it receives over time, which is also called the return.

The reward function can be defined in various ways depending on the task at hand. For example, in a game of chess, a positive reward may be given when the agent captures the opponent's piece, while a negative reward may be given when the agent loses its own piece. In a self-driving car, a positive reward may be given when the car successfully navigates a road without any accidents, while a negative reward may be given when the car collides with an obstacle or violates traffic laws.

The reward function is often domain-specific and designed by the developer or domain expert. The agent uses this function to learn which actions are more beneficial in the long term and which ones should be avoided. The agent's goal is to learn a policy that maximizes the expected total rewards it receives over the long term.

example of a simple reward function in Python:

```
def reward_function(state, action):
    # Define some criteria for receiving rewards
    if state == 'goal_state' and action == 'best_action':
        return 1.0  # Reward for reaching the goal with the best action
    elif state == 'goal_state' and action != 'best_action':
        return 0.5  # Lesser reward for reaching the goal with a suboptimal action
    else:
        return 0.0  # No reward for any other state-action pair
```

This is a generic reward function that returns a reward of 1.0 if the agent is in the goal state and takes the best action, 0.5 if it takes a suboptimal action, and 0.0 for any other state-action pair. The specific criteria for receiving rewards will depend on the task and the specific RL problem. This function can be modified and tailored to a specific problem to provide appropriate rewards for the agent's actions.

Value function

A function that assigns a value to each state or state-action pair, representing the expected cumulative reward that can be obtained from that state or state-action pair.

In reinforcement learning, a value function is a function that estimates the expected total reward an agent will receive starting from a particular state and following a given policy. It is used to evaluate the "goodness" of a state or state-action pair, and is a key component of many reinforcement learning algorithms. The value function can be represented as a function of the state or state-action pair, and can be formalized as follows:

State value function: $V(s) = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... | S_t = s, \pi]$
Action value function: $Q(s, a) = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... | S_t = s, A_t = a, \pi]$

where V(s) represents the expected total reward starting from state s, Q(s, a) represents the expected total reward starting from state s and taking action a, and E[] is the expected value operator. γ is a discount factor that represents the importance of future rewards compared to immediate rewards.

The value function provides a way to assess the quality of different states or state-action pairs, and can be used to make decisions about which actions to take in order to maximize the expected reward. There are several algorithms in reinforcement learning that are based on value functions, including Q-learning, SARSA, and TD-learning.

In Python code, the state value function V(s) can be represented as follows:

```
def state_value_function(state, policy, rewards, gamma):
    """
    Computes the state value function V(s) for a given state `s`.

    Args:
    - state: The current state `s`.
    - policy: The policy function that maps states to actions.
    - rewards: The rewards observed in the environment.
    - gamma: The discount factor that weights future rewards.

    Returns:
    - The state value function V(s).
    """
    expected_return = 0
    for action, action_prob in policy(state).items():
        for next_state, reward in rewards[state][action].items():
            expected_return += action_prob * (reward + gamma *
            state_value_function(next_state, policy, rewards, gamma))
    return expected_return
```

Here, policy is a function that maps states to actions, and rewards is a dictionary that stores the rewards observed in the environment. The function recursively computes the expected return for a given state, by iterating over all possible actions and next states, and multiplying their probabilities with their expected rewards. The function terminates when it reaches a terminal state, for which the expected return is 0.

To compute the state value function V(s) for a given state s, we simply call the state_value_function() with the state s, the policy function, the rewards, and the discount factor gamma. The function returns the expected total reward an agent will receive starting from state s and following the given policy.

```
def action_value_function(state, action, policy, rewards, gamma):
    """
    Computes the action value function Q(s, a) for a given state `s` and action `a`.

    Args:
    - state: The current state `s`.
    - action: The action `a`.
    - policy: The policy function that maps states to actions.
    - rewards: The rewards observed in the environment.
    - gamma: The discount factor that weights future rewards.

    Returns:
    - The action value function Q(s, a).
    """
    expected_return = 0
    for next_state, reward in rewards[state][action].items():
        expected_return += reward + gamma * state_value_function(next_state, policy,
rewards, gamma)
    return expected_return
```

Here, policy is a function that maps states to actions, and rewards is a dictionary that stores the rewards observed in the environment. The function computes the expected return for a given state and action, by iterating over all possible next states and multiplying their expected rewards with their probabilities. The function also calls the state_value_function() to compute the expected return starting from each next state.

To compute the action value function Q(s, a) for a given state s and action a, we simply call the action_value_function() with the state s, the action a, the policy function, the rewards, and the discount factor gamma. The function returns the expected total reward an agent will receive starting from state s and taking action a and following the given policy.

Q-learning

A value-based reinforcement learning algorithm that learns the optimal action-value function by iteratively updating estimates based on experience. Q-learning is a popular and widely used algorithm in reinforcement learning that is used to learn an optimal policy for an agent in an environment. It is a model-free, value-based algorithm that learns a Q-value function, which estimates the expected reward of taking an action in a particular state and following the optimal policy thereafter.

The Q-value of a state-action pair (s, a) is denoted as Q(s, a) and represents the expected discounted reward that the agent will receive by taking action a in state s and following the optimal policy thereafter. The optimal policy is defined as the one that maximizes the expected cumulative reward over a sequence of actions.

The Q-learning algorithm updates the Q-values of state-action pairs iteratively, based on the reward obtained and the Q-values of the next state-action pairs. It uses the following update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

where r is the immediate reward obtained, α is the learning rate, γ is the discount factor, s' is the next state, and a' is the next action chosen based on the current Q-values. The algorithm iteratively updates the Q-values until the Q-values converge to their optimal values.

The Q-learning algorithm is known to converge to the optimal policy under certain conditions, and it has been successfully applied to a wide range of problems in reinforcement learning, such as game playing, robotic control, and autonomous driving, among others.

Here is an example of a basic Q-learning algorithm in Python:

```
import numpy as np

# Define the environment
n_states = 6
n_actions = 2
R = np.array([[0, 0, 0, 0, 1, 0],
              [0, 0, 0, 1, 0, 1]])
T = np.array([[[0.5, 0.5, 0, 0, 0, 0],
               [0.5, 0, 0.5, 0, 0, 0],
               [0, 0.5, 0, 0.5, 0, 0],
               [0, 0, 0.5, 0, 0.5, 0],
               [0, 0, 0, 0, 0, 1],
               [0, 0, 0, 0, 0, 1]],
              [[1, 0, 0, 0, 0, 0],
               [0.5, 0.5, 0, 0, 0, 0],
               [0, 0.5, 0, 0.5, 0, 0],
               [0, 0, 0.5, 0, 0.5, 0],
               [0, 0, 0, 0, 0, 1],
               [0, 0, 0, 0, 0, 1]]])

# Define the Q-learning parameters
alpha = 0.5    # learning rate
gamma = 0.9    # discount factor
epsilon = 0.1  # exploration rate
n_episodes = 1000

# Initialize the Q-value table
Q = np.zeros((n_states, n_actions))

# Q-learning algorithm
for episode in range(n_episodes):
    s = 0  # start from state 0
    done = False
    while not done:
        # Choose an action based on the epsilon-greedy policy
        if np.random.rand() < epsilon:
```

```
            a = np.random.randint(n_actions)
        else:
            a = np.argmax(Q[s, :])
        # Take the chosen action and observe the next state and reward
        s_next = np.random.choice(n_states, p=T[a, s, :])
        r = R[a, s_next]
        # Update the Q-value table
        Q[s, a] += alpha * (r + gamma * np.max(Q[s_next, :]) - Q[s, a])
        # Move to the next state
        s = s_next
        # Check if the goal state is reached
        if s == 4:
            done = True

# Print the learned Q-values
print(Q)
```

In this example, the Q-learning algorithm is applied to a simple grid-world environment with 6 states and 2 actions. The Q-value table is initialized with zeros and updated iteratively based on the observed rewards and the next Q-values. The algorithm uses an epsilon-greedy policy to balance exploration and exploitation during the learning process. After a fixed number of episodes, the learned Q-values are printed for each state-action pair.

Actor-Critic methods

In reinforcement learning, an actor and a critic are two key components of an actor-critic architecture that work together to learn and improve an agent's behavior. The actor is responsible for selecting actions based on the current state of the environment. It takes the current state as input and outputs an action that the agent should take. The actor's goal is to learn a policy that maximizes the expected reward over time. In other words, the actor is learning how to behave in the environment. The critic, on the other hand, evaluates the actions taken by the actor and provides feedback on how good or bad those actions were. It takes the current state and the action taken by the actor as input and outputs a value, which represents the expected reward that the agent can obtain from that state and action. The critic's goal is to learn the value function, which estimates the long-term expected reward for any state and action in the environment. In other words, the critic is learning how good or bad it is to be in a certain state and take a certain action.

The actor and critic work together to improve the agent's behavior. The actor uses the feedback from the critic to update its policy, and the critic uses the actions taken by the actor to update its value function. This process of updating the actor and critic continues over time, with the goal of improving the agent's behavior in the environment.

Deep Q-Network (DQN)

Deep Q-Network (DQN) is a deep reinforcement learning algorithm that uses a neural network to approximate the Q-function in Q-learning. It was introduced by Mnih et al. in 2013 and has since become a popular and effective approach to solving complex control tasks.

In traditional Q-learning, the Q-function is represented as a table that maps states and actions to their corresponding Q-values. However, in environments with high-dimensional state spaces, such as images, it becomes infeasible to maintain such a table. DQN uses a neural network to represent the Q-function instead.

The neural network takes the state as input and outputs the Q-value for each action. The network is trained using a variant of the Q-learning algorithm, where the target Q-value is computed using a Bellman equation update:

$$Q(s, a) = r + \gamma * \max Q(s', a')$$

where s is the current state, a is the action taken, r is the reward received, s' is the next state, γ is the discount factor, and max Q(s', a') is the maximum Q-value for all actions in the next state.

During training, the DQN agent interacts with the environment and stores its experiences in a replay buffer. The agent then samples batches of experiences from the buffer and uses them to update the parameters of the neural network. To improve stability during training, DQN uses a technique called target network, where a separate copy of the Q-network is used to compute the target Q-value, and its parameters are updated less frequently than the main network. DQN has been applied successfully to a variety of tasks, including playing Atari games, navigating in 3D environments, and controlling robotic systems.

Policy Gradients

In reinforcement learning, a policy gradient is a family of algorithms that optimize the parameters of a policy by directly maximizing the expected reward, without explicitly estimating the value function. In other words, the objective of policy gradient methods is to learn a policy that can directly map a state to an action, rather than estimating the value of each state-action pair. The policy is usually parameterized as a neural network, where the input is the current state, and the output is a probability distribution over possible actions. The goal is to learn the optimal policy by iteratively updating the network weights based on the gradient of the expected return with respect to the policy parameters. The policy gradient is computed using the gradient ascent method, which iteratively updates the policy parameters in the direction of the gradient of the expected reward. The gradient is computed using the score function gradient estimator, which takes the form:

$$\nabla\theta\, J(\theta) = E[\nabla\theta \log \pi(a|s) * Q\pi(s,a)]$$

where θ is the parameter vector of the policy network, $J(\theta)$ is the expected reward (also known as the performance objective), $\pi(a|s)$ is the probability of taking action a in state s according to the policy, and $Q\pi(s,a)$ is the expected discounted reward of taking action a in state s, following the policy π.

To estimate the expected return, policy gradient methods use Monte Carlo methods, where a set of trajectories is sampled from the environment using the current policy. These trajectories are then used to compute the gradient of the expected reward with respect to the policy parameters, which is used to update the policy parameters. Policy gradient methods have been shown to be effective for solving complex, high-dimensional control tasks, such as playing video games, controlling robotic systems, and natural language processing. Some examples of policy gradient algorithms include REINFORCE, Actor-Critic, and Proximal Policy Optimization (PPO).

Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) is a model-free, off-policy, actor-critic algorithm that combines the ideas of DQN and policy gradient methods to learn a deterministic policy for continuous action spaces. It was introduced by Lillicrap et al. in 2016 and has since become a popular and effective approach to solving continuous control tasks. DDPG is an actor-critic method, which means that it uses two neural networks, one for the actor and one for the critic. The actor network takes the current state as input and outputs a deterministic action, while the critic network takes the current state and action as input and outputs an estimate of the Q-value for that state-action pair. The critic network is used to update the actor network by providing a signal of the quality of the action taken.

The actor network is trained using policy gradients, which involves computing the gradient of the expected reward with respect to the policy parameters, and updating the actor network in the direction of the gradient using the gradient ascent method. The critic network is trained using the temporal difference (TD) learning algorithm, where the target Q-value is computed using the Bellman equation, and the loss function is the mean squared error between the predicted Q-value and the target Q-value. To improve stability during training, DDPG uses several techniques, including experience replay and target networks. Experience replay is used to store the experiences of the agent in a replay buffer and to sample batches of experiences from the buffer to update the networks. Target networks are used to reduce the correlation between the target and predicted Q-values by slowly updating the target network parameters using a soft update rule. DDPG has been applied successfully to a variety of continuous control tasks, such as robotic manipulation, locomotion, and navigation. It has also been extended to handle multi-agent environments in the form of MADDPG (Multi-Agent Deep Deterministic Policy Gradient).

OpenAI Gym and DeepMindAI

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It provides a collection of standardized environments, or "tasks," that researchers and developers can use to evaluate and test their reinforcement learning algorithms. These environments include classic control problems, Atari games, board games, robotics simulations, and more. OpenAI Gym provides a simple and unified API for interacting with these environments, which makes it easy to train and evaluate reinforcement learning algorithms across a range of different domains. The API includes methods for getting the current state of the environment, taking an action, getting the reward, and checking whether the episode has ended.

In addition to the environments, OpenAI Gym also provides tools for visualizing the performance of reinforcement learning algorithms, such as graphs of the reward over time and videos of the agent's behavior in the environment. It also supports distributed training across multiple machines using the Ray distributed computing system. OpenAI Gym is an open-source project and is widely used in the reinforcement learning research community. It has been used to benchmark and compare a variety of reinforcement learning algorithms, including Deep Q-Networks, Policy Gradient methods, Actor-Critic methods, and more. As of late 2022 OpenAI has announced it will no longer support or develop Gym futher. This project has been taken over by the Farama Foundation (https://github.com/Farama-Foundation/Gymnasium). See their documentation on how to keep your Gym code up-to-date.

DeepMind, a research lab founded in 2010 headquartered in London, has made significant contributions to the field of reinforcement learning. Here are some of their key contributions:

Deep Q-Networks (DQN): In 2013, DeepMind introduced the DQN algorithm, which was the first to demonstrate human-level performance on a suite of Atari games using only raw pixels as input. DQN used a deep neural network to approximate the Q-function, and used experience replay and a target network to improve stability and reduce correlation in the training process.

AlphaGo: In 2016, DeepMind's AlphaGo defeated the world champion of the board game Go, marking a significant milestone in the development of artificial intelligence. AlphaGo used a combination of Monte Carlo tree search and deep neural networks to evaluate the quality of moves and select the next move to play.

AlphaZero: In 2017, DeepMind introduced AlphaZero, which was able to achieve superhuman performance on the games of chess, shogi, and Go, using a single algorithm and a single set of hyperparameters. AlphaZero combined Monte Carlo tree search with a deep neural network to learn to play the games from scratch, without any human knowledge of the games.

MuZero: In 2019, DeepMind introduced MuZero, which is a general-purpose algorithm that can learn to play any game without any prior knowledge of the rules. MuZero uses a combination of Monte Carlo tree search and a learned model of the environment to simulate future states and rewards, and learns to predict the value of each state and the policy for selecting actions.

OpenAI Gym: In 2016, OpenAI, a research lab co-founded by Elon Musk and others, partnered with DeepMind to release OpenAI Gym, a toolkit for developing and comparing reinforcement learning algorithms. OpenAI Gym provides a collection of standardized environments, or "tasks," that researchers and developers can use to evaluate and test their reinforcement learning algorithms.

Explainability and interpretability of deep reinforcement learning models

Explainability and interpretability are important properties of deep reinforcement learning models because they enable us to understand how the model makes decisions and how it might be improved. Explainability refers to the ability of a model to provide a clear and concise explanation of how it arrived at a particular decision or prediction. In the context of reinforcement learning, explainability might involve understanding which features of the environment the model is paying attention to, or how the model is choosing actions based on those features.

Interpretability, on the other hand, refers to the ability to understand the internal workings of the model, such as the structure of the neural network and the values of its parameters. Interpretability is important because it enables researchers and practitioners to diagnose problems with the model and to identify areas where it might be improved.

There are several techniques that can be used to improve the explainability and interpretability of deep reinforcement learning models, including:

Visualizing model activations: This involves plotting the output of individual neurons or layers of the model to better understand which features of the input are being used to make decisions.

Attention mechanisms: Attention mechanisms allow the model to focus on specific regions of the input, which can help improve explainability by highlighting the most important features of the environment.

Model compression: Simplifying the model structure can help improve interpretability by making it easier to understand how the model is making decisions.

Input perturbations: Changing the input in specific ways can help reveal which features the model is relying on to make decisions.

Counterfactual reasoning: This involves generating alternative scenarios that could have occurred, and using those scenarios to better understand why the model made a particular decision.

Sample Efficiency

Sample efficiency in reinforcement learning refers to how many interactions with the environment, or "samples," are required for an agent to learn a good policy. In other words, it is a measure of how quickly the agent can learn from experience. Sample efficiency is an important consideration in reinforcement learning because interacting with the environment can be expensive or time-consuming in many real-world scenarios. For example, here are several ways to measure sample efficiency in RL, including:

Total number of interactions: This measures the total number of interactions an agent has with its environment during training. A more sample-efficient algorithm would require fewer interactions to achieve the same level of performance.

Time to convergence: This measures the time it takes for an algorithm to converge to an optimal policy. A more sample-efficient algorithm would converge faster, requiring less time to achieve the same level of performance.

Data efficiency: This measures how much data an algorithm needs to achieve a certain level of performance. A more sample-efficient algorithm would need less data to achieve the same level of performance.

Sample complexity: This measures the number of samples required to learn an effective policy. A more sample-efficient algorithm would have a lower sample complexity, requiring fewer samples to achieve the same level of performance.

It's important to note that the best way to measure sample efficiency can depend on the specific problem being tackled, and that different metrics may be more or less appropriate in different contexts. Additionally, there may be trade-offs between sample efficiency and other factors, such as computational efficiency or generalization ability

Training an AI agent to play a video game such as Pac-Man using reinforcement learning involves applying these concepts in a specific way. Here is an overview of how these concepts are used in training an AI agent to play Pac-Man using reinforcement learning:

Markov Decision Process (MDP): The Pac-Man game environment can be modeled as an MDP by defining the state space, action space, and reward function. The state space includes the positions of all the characters on the board, as well as the locations of the dots and power pellets. The action space includes the possible movements of Pac-Man, such as up, down, left, and right. The reward function assigns a reward to the agent for each action it takes, such as eating a dot or power pellet or getting caught by a ghost.

Agent and Environment: The AI agent interacts with the Pac-Man game environment by taking actions and receiving observations and rewards. The agent observes the current state of the game environment and selects an action based on its current policy.

Observation/State Space: The set of possible observations or states that the agent can perceive includes the current position of Pac-Man and the locations of the dots and power pellets, as well as the positions of the ghosts.

Action Space: The set of possible actions that the agent can take includes moving Pac-Man up, down, left, or right.

Rewards and Discounting: The agent receives rewards for each action it takes, such as eating a dot or power pellet or getting caught by a ghost. The rewards are often discounted to give more weight to immediate rewards than to future rewards.

Policy (π): The agent's policy is its strategy for selecting actions based on the current state of the game environment. In reinforcement learning, the agent's policy is updated over time as it learns from its experience.

Q-Learning: Q-learning is a popular value-based reinforcement learning algorithm that can be used to train an AI agent to play Pac-Man. The algorithm learns the optimal action-value function by iteratively updating estimates based on experience.

Exploration/Exploitation Tradeoff: To learn an optimal policy, the agent must balance exploration (trying new actions) and exploitation (using known good actions) to maximize the expected reward.

By applying these concepts, an AI agent can learn to play Pac-Man through trial and error, gradually improving its policy to achieve higher scores and better performance. With enough training, the AI agent can even surpass human-level performance and achieve superhuman play. Implementing an AI agent to play Pac-Man using reinforcement learning in Python involves applying the concepts described earlier using appropriate libraries and tools. Here is a high-level overview of how this can be done:

Markov Decision Process (MDP): The Pac-Man game environment can be modeled as an MDP using a library like OpenAI Gym or PyGame.

Agent and Environment: The AI agent can be implemented using a deep reinforcement learning library like TensorFlow, Keras, or PyTorch. The agent interacts with the Pac-Man game environment by taking actions and receiving observations and rewards.

Observation/State Space: The set of possible observations or states that the agent can perceive can be represented using a NumPy array or a PyTorch tensor.

Action Space: The set of possible actions that the agent can take can be represented using a NumPy array or a PyTorch tensor.

Rewards and Discounting: The agent receives rewards for each action it takes, and the rewards can be discounted using a discount factor like 0.99.

Policy (π): The agent's policy can be implemented using a deep neural network that takes the current state of the game environment as input and outputs a probability distribution over the possible actions.

Q-Learning: Q-learning can be implemented using a deep Q-network (DQN) that learns the optimal action-value function by iteratively updating estimates based on experience. This can be implemented using a deep reinforcement learning library like TensorFlow, Keras, or PyTorch.

Exploration/Exploitation Tradeoff: To balance exploration and exploitation, an epsilon-greedy policy can be used, where the agent selects the optimal action with probability 1-epsilon and a random action with probability epsilon. The idea is to choose the best action with probability 1-epsilon, and a random action with probability epsilon.

Here's how it works:

At each time step, the agent selects an action based on the current state.

With probability 1-epsilon, the agent selects the action with the highest Q-value (exploitation).

With probability epsilon, the agent selects a random action (exploration).

Over time, as the agent collects more experience, epsilon is typically decreased, so that the agent relies more on exploitation and less on exploration.

Here's an example of how to implement an epsilon-greedy policy in Python:

```
import numpy as np

def epsilon_greedy_policy(Q, state, epsilon):
    # Choose the action with the highest Q-value with probability 1-epsilon
    if np.random.uniform(0, 1) > epsilon:
        action = np.argmax(Q[state, :])
    # Choose a random action with probability epsilon
    else:
        action = np.random.choice(np.arange(Q.shape[1]))
    return action
```

In this example, Q is the Q-table that contains the estimated Q-values for each state-action pair, state is the current state, and epsilon is the exploration rate. The function returns the action selected by the epsilon-greedy policy.

Model or Model-Free Reinforcement Learning:

Model-based and model-free learning are two approaches for learning optimal policies from interactions with an environment. Model-based learning involves building a model of the environment that captures the dynamics of the state transitions and rewards. The agent uses this model to simulate future trajectories and evaluate potential actions before taking them. In other words, the agent learns the optimal policy by planning ahead using its model of the environment. This approach can be more sample-efficient than model-free learning because the agent can use its model to learn from simulated experiences before interacting with the environment. On the other hand, model-free learning does not involve building an explicit model of the environment. Instead, the agent learns the optimal policy by directly estimating the value of each state or state-action pair through trial-and-error interactions with the environment. This approach involves updating the agent's value estimates based on the observed rewards and next states without using a model to simulate future trajectories. Model-free learning is generally simpler and more scalable than model-based learning, but may require more interactions with the environment to learn an optimal policy. Overall, the choice between model-based and model-free learning depends on the specifics of the problem at hand, including the size of the state and action spaces, the complexity of the dynamics and rewards, and the available computational resources.

A simple model-based reinforcement learning algorithm using PyTorch.

```
import torch
import torch.nn as nn
import torch.optim as optim
import random
```

```
# Define the environment
n_states = 5
n_actions = 2
transition_prob = torch.tensor([
    [0.7, 0.3],
    [0.4, 0.6],
    [0.2, 0.8],
    [0.1, 0.9],
    [0.5, 0.5],
])
rewards = torch.tensor([-0.1, -0.2, -0.3, -0.4, 1.0])

# Define the model
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.fc1 = nn.Linear(n_states, 10)
        self.fc2 = nn.Linear(10, n_states * n_actions)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x.view(-1, n_states, n_actions)

model = Model()
optimizer = optim.Adam(model.parameters())

# Train the model
n_episodes = 1000
for i in range(n_episodes):
    state = random.randint(0, n_states - 1)
    history = []
    done = False

    while not done:
        # Generate an action from the model's belief of the environment
        action_probs = model(torch.eye(n_states)[state])
        action = torch.multinomial(action_probs, 1).item()

        # Take the action and observe the next state and reward
        next_state = torch.multinomial(transition_prob[state], 1).item()
        reward = rewards[next_state]

        # Update the model's belief of the environment based on the observed transition
        target = reward + 0.9 * torch.max(model(torch.eye(n_states)[next_state]))
        loss = nn.MSELoss()(model(torch.eye(n_states)[state])[0][action], target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Update the current state
        state = next_state

        # Check if the episode is done
        if reward > 0:
            done = True
            print("Episode {} completed in {} steps".format(i, len(history)))
```

In this example, we define a simple environment with 5 states, 2 actions, and transition probabilities and rewards represented as tensors. We then define a neural network model that takes a one-hot encoded state vector as input and outputs a tensor of action probabilities for each state. We use the model to generate actions and update its belief of the environment based on observed transitions. The model is trained using a mean squared error loss and an Adam optimizer. Finally, we run multiple episodes and print the number of steps taken to reach a positive reward.

Simple model-free reinforcement learning algorithm using PyTorch and the Q-learning algorithm.

```
import torch
import random

# Define the environment
n_states = 5
n_actions = 2
transition_prob = torch.tensor([
    [0.7, 0.3],
    [0.4, 0.6],
    [0.2, 0.8],
    [0.1, 0.9],
    [0.5, 0.5],
])
rewards = torch.tensor([-0.1, -0.2, -0.3, -0.4, 1.0])

# Define the Q-function
Q = torch.zeros(n_states, n_actions)

# Set the learning rate and discount factor
lr = 0.1
gamma = 0.9

# Train the Q-function
n_episodes = 1000
for i in range(n_episodes):
    state = random.randint(0, n_states - 1)
    done = False

    while not done:
        # Choose an action using an epsilon-greedy policy
        if random.random() < 0.1:
            action = random.randint(0, n_actions - 1)
        else:
            action = torch.argmax(Q[state]).item()

        # Take the action and observe the next state and reward
        next_state = torch.multinomial(transition_prob[state], 1).item()
        reward = rewards[next_state]

        # Update the Q-function using the Q-learning algorithm
        td_error = reward + gamma * torch.max(Q[next_state]) - Q[state][action]
        Q[state][action] += lr * td_error

        # Update the current state
        state = next_state
```

```
        # Check if the episode is done
        if reward > 0:
            done = True
            print("Episode {} completed in {} steps".format(i, len(history)))
```

In this example, we define a simple environment with 5 states, 2 actions, and transition probabilities and rewards represented as tensors. We then define a Q-function as a tensor of state-action values and use the Q-learning algorithm to update the Q-function based on observed transitions. The Q-function is updated using a learning rate and discount factor, and actions are chosen using an epsilon-greedy policy. Finally, we run multiple episodes and print the number of steps taken to reach a positive reward.

Solving CartPole using a policy gradient:

```
import gym
import numpy as np
import tensorflow as tf

# Define the policy network
inputs = tf.keras.layers.Input(shape=(4,))
dense = tf.keras.layers.Dense(16, activation='relu')(inputs)
outputs = tf.keras.layers.Dense(2, activation='softmax')(dense)
model = tf.keras.models.Model(inputs=inputs, outputs=outputs)

# Define the optimizer and loss function
optimizer = tf.keras.optimizers.Adam(lr=0.001)
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()

# Define the environment
env = gym.make('CartPole-v1')

# Define the training loop
num_episodes = 1000
discount_factor = 0.99
for i in range(num_episodes):
    # Reset the environment for each episode
    state = env.reset()
    states, actions, rewards = [], [], []
    done = False

    # Run the episode until termination
    while not done:
        # Get the action probabilities from the policy network
        logits = model(np.array([state]))
        action_probs = tf.nn.softmax(logits).numpy()[0]

        # Sample an action from the action probabilities
        action = np.random.choice(env.action_space.n, p=action_probs)

        # Take the chosen action and observe the reward and next state
        next_state, reward, done, _ = env.step(action)
```

```
            # Record the state, action, and reward
            states.append(state)
            actions.append(action)
            rewards.append(reward)

            # Update the state for the next iteration
            state = next_state

        # Compute the discounted rewards
        discounted_rewards = []
        running_sum = 0
        for r in reversed(rewards):
            running_sum = r + discount_factor * running_sum
            discounted_rewards.append(running_sum)
        discounted_rewards.reverse()
        discounted_rewards = np.array(discounted_rewards)
        discounted_rewards = (discounted_rewards - np.mean(discounted_rewards)) /
(np.std(discounted_rewards) + 1e-10)

        # Compute the loss and update the policy network
        with tf.GradientTape() as tape:
            logits = model(np.array(states))
            loss = -tf.reduce_mean(tf.math.log(tf.reduce_sum(logits * tf.one_hot(actions,
depth=2), axis=1)) * discounted_rewards)
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))

        # Print the episode score
        score = sum(rewards)
        print(f"Episode {i+1}: Score = {score}")
```

Previously we covered training an RL agent using a policy gradient. Another way to train an RL agent is using DQN.

dqn algorithm in pytorch that can serve as a starting point:

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import gym

# define the q-network
class qnetwork(nn.module):
    def __init__(self, state_dim, action_dim, hidden_dim):
        super(qnetwork, self).__init__()
        self.linear1 = nn.linear(state_dim, hidden_dim)
        self.linear2 = nn.linear(hidden_dim, hidden_dim)
        self.linear3 = nn.linear(hidden_dim, action_dim)

    def forward(self, state):
        x = torch.relu(self.linear1(state))
        x = torch.relu(self.linear2(x))
        x = self.linear3(x)
        return x
```

```
# define the dqn agent
class dqnagent:
    def __init__(self, env, hidden_dim, lr, gamma, epsilon):
        self.env = env
        self.q_net = qnetwork(env.observation_space.shape[0], env.action_space.n,
hidden_dim)
        self.target_q_net = qnetwork(env.observation_space.shape[0], env.action_space.n,
hidden_dim)
        self.target_q_net.load_state_dict(self.q_net.state_dict())
        self.optimizer = optim.adam(self.q_net.parameters(), lr=lr)
        self.gamma = gamma
        self.epsilon = epsilon

    def act(self, state):
        if np.random.uniform() < self.epsilon:
            return self.env.action_space.sample()
        else:
            with torch.no_grad():
                q_values = self.q_net(torch.floattensor(state))
                return q_values.argmax().item()

    def update(self, batch_size):
        states, actions, rewards, next_states, dones =
self.replay_buffer.sample(batch_size)

        with torch.no_grad():
            target_q_values = self.target_q_net(torch.floattensor(next_states)).max(dim=1,
keepdim=true)[0]
            target_q_values = rewards + self.gamma * target_q_values * (1 - dones)

        q_values = self.q_net(torch.floattensor(states)).gather(1,
torch.longtensor(actions))

        loss = nn.functional.mse_loss(q_values, target_q_values)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    def train(self, num_episodes, batch_size):
        for episode in range(num_episodes):
            state = self.env.reset()
            done = false
            while not done:
                action = self.act(state)
                next_state, reward, done, info = self.env.step(action)
                self.replay_buffer.add(state, action, reward, next_state, done)
                state = next_state
                if len(self.replay_buffer) >= batch_size:
                    self.update(batch_size)
            if episode % 10 == 0:
                self.target_q_net.load_state_dict(self.q_net.state_dict())
```

this code defines a Q-network and a DQN agent, and includes the main training loop for the agent. it assumes that the OpenAI gym environment is already defined and initialized. note that this code assumes the existence of a replay buffer class that contains the replay buffer data and sampling methods. the implementation of the replay buffer is omitted for brevity. Also, note that this is a relatively basic implementation of a DQN algorithm and may not be optimal for all problems.

Implementation of a replay buffer using a deque as a circular buffer with a reference to the replay memory, which can be used in a DQN algorithm:

```
from collections import deque
```

```
import random

class replaybuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = deque(maxlen=capacity)
        self.memory = np.zeros((capacity, state_dim + action_dim + 1 + state_dim),
dtype=np.float32)
        self.position = 0

    def add(self, state, action, reward, next_state, done):
        transition = (state, action, reward, next_state, done)
        self.buffer.append(transition)
        self.memory[self.position] = np.concatenate((state, [action, reward], next_state))
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        indices = random.sample(range(len(self.buffer)), batch_size)
        states, actions, rewards, next_states, dones = [], [], [], [], []
        for index in indices:
            state, action, reward, next_state, done = self.buffer[index]
            states.append(state)
            actions.append(action)
            rewards.append(reward)
            next_states.append(next_state)
            dones.append(done)
        return states, actions, rewards, next_states, dones

    def __len__(self):
        return len(self.buffer)
```

in this implementation, the replay buffer is initialized with a capacity, and the memory is allocated with zeros to store the transitions in the form of (state, action, reward, next_state, done). whenever a new transition is added to the replay buffer, it is appended to the buffer and the corresponding entry in the memory is updated. when the replay buffer is full, the new entries overwrite the oldest ones, creating a circular buffer.

the sample method is used to retrieve a batch of transitions from the replay buffer. it randomly selects batch_size transitions and returns the states, actions, rewards, next_states, and dones in separate lists. note that in this implementation, the state_dim and action_dim variables are assumed to be defined elsewhere. also, the np module from numpy is assumed to be imported.

State Space and Action Space:

state_dim and action_dim are variables that represent the dimensionality of the state space and action space, respectively, in a reinforcement learning problem. the state space is the set of all possible states that an agent can be in, while the action space is the set of all possible actions that an agent can take. In the implementation I provided earlier, i assumed that state_dim and action_dim were defined elsewhere, so you would need to define them yourself based on the specifics of your problem. for example, if you were working on a problem where the state was represented as a vector of length 4 and the action was represented as a scalar value, you would define state_dim as 4 and action_dim as 1.

here's an example of how you could define state_dim and action_dim in a simple environment where the state is represented as a vector of length 2 and the action is a scalar value:

```
state_dim = 2
action_dim = 1
```

you would define these variables based on the characteristics of your environment and the way you choose to represent the state and action spaces in your implementation.

Reinforcement Learning with Unity Game Dev Engine

Unity is a powerful and popular game development engine that has been used to create some of the most successful and popular games across a wide range of genres. It is a versatile platform that allows developers to create games for a variety of platforms including PC, consoles, mobile devices, and virtual reality. Unity offers a comprehensive set of tools, a robust asset store, and an active community of developers that make it an ideal choice for both beginners and experienced game developers.

One of the strengths of Unity is its ease of use and flexibility. Developers can use Unity's visual editor to create 2D and 3D games without needing to know how to code, but for more advanced functionality, developers can use C# or other programming languages to create custom scripts. Unity also offers a range of features including physics, lighting, audio, and animation tools that allow developers to create games with impressive graphics and immersive gameplay.

Another advantage of Unity is its asset store, which offers a wide range of pre-made assets such as models, textures, and sound effects that developers can use to speed up their game development process. Additionally, the store includes plugins that add additional functionality to the engine, such as support for specific platforms or integration with third-party services.

Overall, Unity is a powerful game development engine that is accessible to developers of all skill levels. Its versatility, ease of use, and robust community make it a popular choice for creating games across a variety of platforms and genres.

To install Unity game engine on your computer, follow these steps:

Go to the Unity website at https://unity.com and click on the "Get started" button.

If you already have a Unity account, log in. If you don't have an account, create one by clicking the "Create account" button and following the instructions.

Once you are logged in, click on the "Download Unity" button.

Choose the version of Unity that you want to install. You can choose either the latest version or an older version if you need to be compatible with a specific project.

Choose the operating system you are using. Unity supports Windows, Mac, and Linux.

Select the additional components you want to install. You can choose to install components like Visual Studio, which is a powerful code editor that integrates with Unity.

Click the "Download Installer" button to download the Unity installer.

Run the installer and follow the instructions to complete the installation process.

Once the installation is complete, you can open Unity and start creating games. If you encounter any issues during the installation process, consult the Unity documentation or seek help from the Unity community.

step-by-step tutorial on how to create a "Hello World" program in Unity using C#:

Open Unity and create a new project. You can name the project anything you like.

In the Unity editor, click on the "Create" button and select "C# Script". Name the script "HelloWorld".

Double-click the "HelloWorld" script to open it in your preferred code editor.

In the script, type the following code:

```
using UnityEngine;
using System.Collections;

public class HelloWorld : MonoBehaviour
{
    void Start()
    {
        Debug.Log("Hello World!");
    }
}
```

Save the script and go back to the Unity editor.

Drag the "HelloWorld" script onto the "Main Camera" object in the Hierarchy window.

Press the "Play" button to run the game.

You should see "Hello World!" printed in the console window at the bottom of the Unity editor.

Congratulations, you have created a "Hello World" program in Unity using C#! This basic program demonstrates how to use the Start() method to run code when the game starts and how to use the Debug.Log() method to print messages to the console. From here, you can start to experiment with more advanced features of Unity and C# to create your own games and interactive applications.

Unity RL Kit:

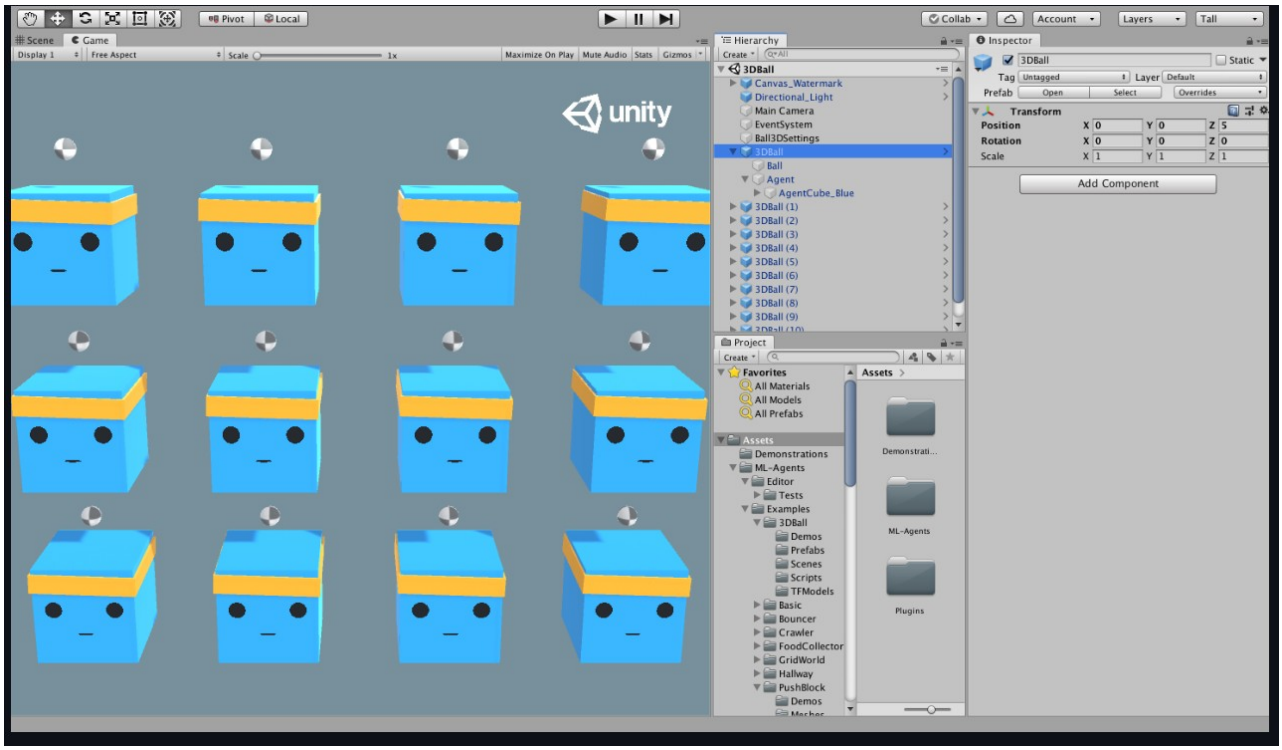
The Unity Machine Learning Agents (ML-Agents) toolkit is an open-source framework that enables Unity developers to integrate artificial intelligence (AI) and machine learning (ML) technologies into their games and simulations. The toolkit provides a range of features, tools, and resources that make it easier for developers to train agents to learn behaviors in virtual environments.

The Unity ML-Agents toolkit (https://github.com/Unity-Technologies/ml-agents/tree/release-0.15.1) includes a range of algorithms, such as deep reinforcement learning, that can be used to train agents to perform tasks in complex environments. Developers can use the toolkit to train agents to learn skills such as navigation, object manipulation, and decision making. The toolkit is built on top of the Unity game engine and provides an interface for developers to easily create and control agents, set up environments, and run simulations. It also includes features such as data collection, visualization, and analysis, which help developers monitor and optimize the performance of their trained agents.

The ML-Agents toolkit is designed to be accessible to developers of all skill levels, and it includes a range of tutorials, documentation, and example projects to help developers get started with using AI and ML in their Unity projects.

Balance Ball with RL

The following example is based on https://github.com/Unity-Technologies/ml-agents/blob/release-0.15.1/docs/Getting-Started-with-Balance-Ball.md Like in the OpenAI Gym example of CartPole, the objective of this game is to balance a ball instead of a pole. It uses Python libraries to train the Agent and integrates this into a Unity Game.



Screenshot from the Unity Game Engine Interface for Balance Ball

The Agent is the actor that observes and takes actions in the environment. In the 3D Balance Ball environment, the Agent components are placed on the twelve "Agent" GameObjects. The base Agent object has a few properties that affect its behavior:

Behavior Parameters — Every Agent must have a Behavior. The Behavior determines how an Agent makes decisions. More on Behavior Parameters in the next section.
Max Step — Defines how many simulation steps can occur before the Agent's episode ends. In 3D Balance Ball, an Agent restarts after 5000 steps.

Before making a decision, an agent collects its observation about its state in the world. The vector observation is a vector of floating point numbers which contain relevant information for the agent to make decisions.
The Behavior Parameters of the 3D Balance Ball example uses a Space Size of 8. This means that the feature vector containing the Agent's observations contains eight elements: the x and z components of the agent cube's rotation and the x, y, and z components of the ball's relative position and velocity.

An Agent is given instructions in the form of a float array of actions. ML-Agents toolkit classifies actions into two types: the Continuous vector action space is a vector of numbers that can vary continuously. What each element of the vector means is defined by the Agent logic (the training process just learns what values are better given particular state observations based on the rewards received when it tries different values). For example, an element might represent a force or torque applied to a Rigidbody in the Agent. The Discrete action vector space defines its actions as tables. An action given to the Agent is an array of indices into tables.

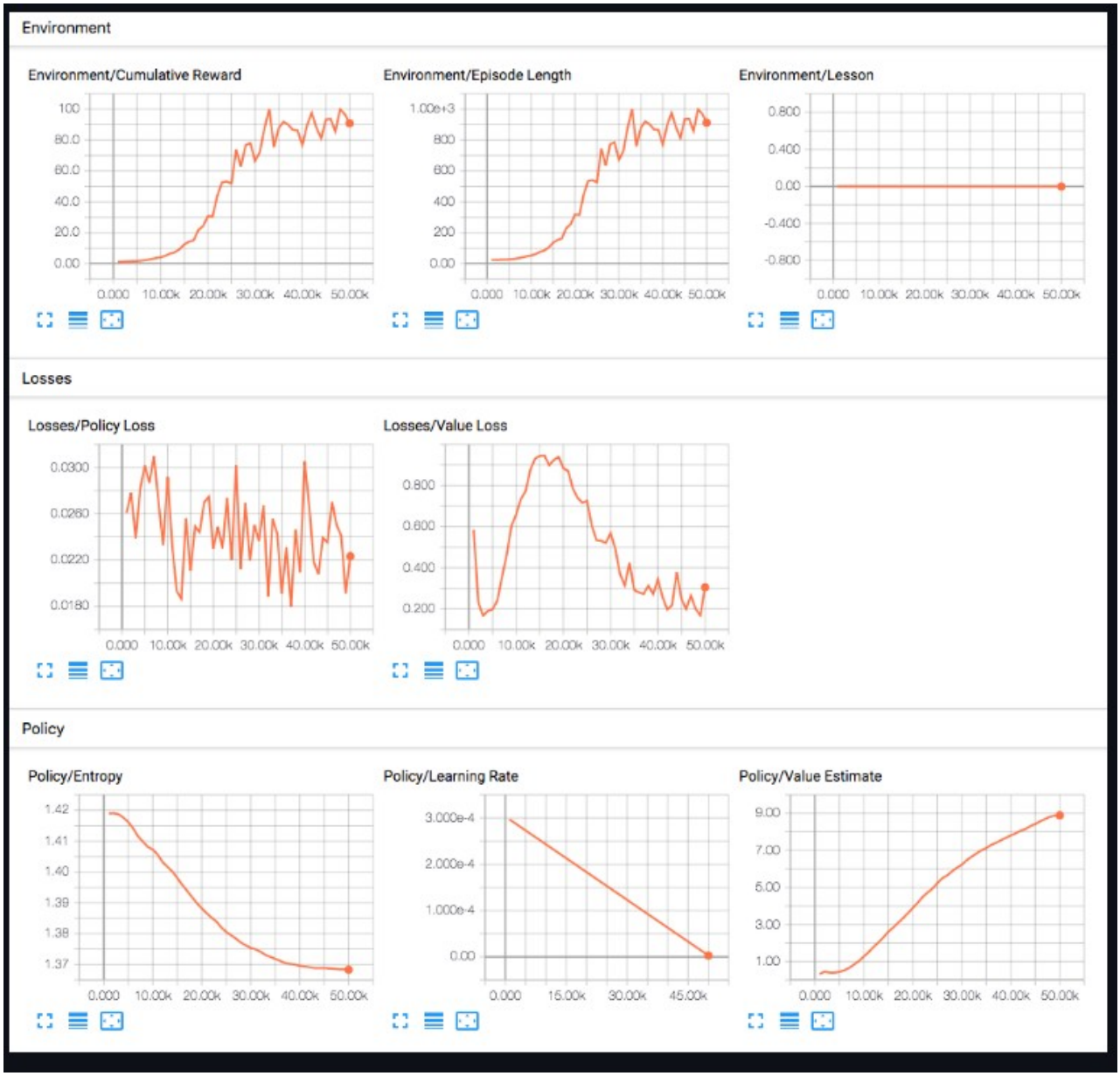
The 3D Balance Ball example is programmed to use continuous action space with Space Size of 2.

In order to train an agent to correctly balance the ball, we provide two deep reinforcement learning algorithms.

The default algorithm is Proximal Policy Optimization (PPO). This is a method that has been shown to be more general purpose and stable than many other RL algorithms. For more information on PPO, OpenAI has a blog post explaining it, and our page for how to use it in training.

The framework provides a Soft-Actor Critic, an off-policy algorithm that has been shown to be both stable and sample-efficient. For more information on SAC.

You initialize the training cycle by issuing a Python command, then navigate to your Unity interface and press the play button which starts the training cycles. Eventually, using Tensorboard you can see how the training went as it outputs graphs that show the results:



Lesson - only interesting when performing curriculum training. This is not used in the 3D Balance Ball environment.
Cumulative Reward - The mean cumulative episode reward over all agents. Should increase during a successful training session.
Entropy - How random the decisions of the model are. Should slowly decrease during a successful training process. If it decreases too quickly, the beta hyperparameter should be increased.
Episode Length - The mean length of each episode in the environment for all agents.
Learning Rate - How large a step the training algorithm takes as it searches for the optimal policy. Should decrease over time.

Policy Loss - The mean loss of the policy function update. Correlates to how much the policy (process for deciding actions) is changing. The magnitude of this should decrease during a successful training session.
Value Estimate - The mean value estimate for all states visited by the agent. Should increase during a successful training session.
Value Loss - The mean loss of the value function update. Correlates to how well the model is able to predict the value of each state. This should decrease during a successful training session.

This brief overview introduces you to how RL works in a video game and how to develop a game using RL. As this technology progresses we will see increasing numbers of video games incorporating RL into their mechanics, some for NPC development, some for game play elements.

BIBLIOGRAPHY

Baker, B. (2019) '*Slitherine's Command: the wargame transforming operational simulation'* in Analysis https://www.army-technology.com/features/military-simulation-game/

Calvano, Emilio and Calzolari, Giacomo and Denicolo, Vincenzo and Pastorello, Sergio, (2018) *'Artificial Intelligence, Algorithmic Pricing and Collusion*' (December 20, 2018). Available at SSRN: https://ssrn.com/abstract=3304991or http://dx.doi.org/10.2139/ssrn.3304991 https://www.technologyreview.com/the-download/612947/pricing-algorithms-can-learn-to-collude-with-each-other-to-raise-prices/ (accessed 2/15/2019)

De Byl, Penny (2019) *The Beginner's Guide to Artificial Intelligence in Unity*.
A practical guide to programming non-player characters for games.
Udemy Course: https://www.udemy.com/artificial-intelligence-in-unity/

De Jonge, Dave (2010) *Optimizing a Diplomacy Bot Using Genetic Algorithms*, Masters Thesis
Http://www.ellought.demon.co.uk/dipai

Fallon, John (2013) *Believable Behaviour of Background Characters in Open World Games*
https://www.scss.tcd.ie/publications/theses/diss/2013/TCD-SCSS-DISSERTATION-2013-024.pdf

Goodfellow, Ian. Bengio, Yoshua. Courville, Aaron (2016*) 'Deep Learning'* MIT Press online:
http://www.deeplearningbook.org

Green (2019) '*What Can Psychopaths Teach us About AI*' online: https://thenextweb.com/artificial-intelligence/2019/03/15/what-can-psychopaths-teach-us-about-ai

Gruber, Capt. Donald. (2015) '*Tactical AI in Real Time Strategy Games'* Air Force Institute of Technology

Gunn, E.A.A. Craenen, B.G.W., Hart, E. (2009) '*A Taxonomy of Video Games and AI'*

Hudlicka, Eva (2008) *What Are We Modeling When We Model Emotion*?
Psychometrix Associates, Inc. https://works.bepress.com/eva_hudlicka/8/

Jagielski, Matthew. (2018) *'Manipulating Machine Learning: Poisoning Attacks and Countermeasures for Regression Learning*' IEEE Security and Privacy Symposium
Http://github.com/jagielski/manip-ml
presentation video: https://youtu.be/ahC4KPd9lSY (accessed 3/6/19)

Larsen, John Alec. (2016) *A Chatbot Service for use in Video Game Development*,
https://core.ac.uk/reader/396761

McCarron, Michael J. (2023) *Battlespace of Mind: AI and Cybernetics in Information Warfare*, Trine Day Publishing, ISBN: 9781634244244

Millington, Ian. Funge, John (2016) *Artificial intelligence for games* – 2nd ed. p. cm. Includes index. ISBN 978-0-12-374731-0

Osaba, Osonde and Davis, Paul (2017) '*An Artificial Intelligence/Machine Learning Perspective on Social Simulation: New Data and New Challenges*' Rand Corp.

Robertson, Glen. Watson, Ian. (2014) *A Review of Real-Time Strategy Game AI* University of Auckland

Sanatan, Marcus. (2019) '*Theory of Computation: Finite State Machines*', online: https://stackabuse.com/theory-of-computation-finite-state-machines/ (accessed 6/15/19)

Steinhardt, Jacob. Koh, Pang Wei, Liang, Percy. (2018) '*Certified Defenses for Data Poisoning Attacks*' Neuro Information Processing Systems Conference online: https://papers.nips.cc/paper/6943-certified-defenses-for-data-poisoning-attacks.pdf

Surber, Regina (2018) '*Artificial Intelligence: Autonomous Technology (AT), Lethal Autonomous Weapons Systems (LAWS) and Peace Time Threats*' ICT4Peace Foundation and the Zurich Hub for Ethics and Technology (ZHET)

Sutton, R., Barto, A (1998) *Reinforcement Learning An Introduction* ISBN: 9780262193986 Publisher: The MIT Pres, http://incompleteideas.net/book/ebook/the-book.html

Tarasenko, Sergey. (2016). *Emotionally Colorful Reflexive Games*. https://arxiv.org/abs/1101.0820

Thompson, Tommy (2014) In the Directors Chair lecture 4, https://medium.com/@t2thompson/in-the-directors-chair-the-ai-of-left-4-dead-78f0d4fbf86a

Thompson, Tommy (2018) '*Facing Your Fear*' online: https://aiandgames.com/facing-your-fear/
-- *The Road to War The AI of Total War* (2018b)
Online:
https://www.gamasutra.com/blogs/TommyThompson/20180131/313865/The_Road_To_War__The_AI_of_Total_War_Part_1.php

Rose, Caroline (2015) *Realistic Dialogue Engine for Video Games* The University of Western Ontario

Webb, Amy (2019) '*The Big Nine: How the Tech Titans and Their Thinking Machines Could Warp Humanity*' Business Insider Online: https://www.businessinsider.com/amy-webb-big-nine-artificial-intelligence-2019-2/?r=US&IR=T (accessed 2/25/19)

Weber et al, (2011) *Building Human-Level AI for Real Time Strategy Games*, 2011 AAAI Fall Symposium

ABOUT THE AUTHOR

Michael Joseph McCarron is a Software Engineer and Data Scientist. He lives in Galway, Ireland.