Topology Optimization

This is a manual for topology optimization by genetic algorithms and reinforcement learning software developed in the years of 2018 to 2023.

Table of contents:

1.1. Generative design of bionic partition by genetic algorithms and finite element analysis

1.1.1. Background and rationale

Modern research on aircraft design aim to reduce airplanes components weight, optimize aircraft performance and contribute to the challenge of reducing fuel consumption and operational costs. From this perspective novel materials and technologies are developed, but also advances in design methods and tools are put forward. Generative design is one of the approaches to automatically optimize component design. It uses evolutionary algorithms and topological optimization to generate novel, unconventional and complex structures like novel bionic partition for Airbus A 320 cabin interiors,[1], considered in this article.

1.1.2. Methodology

1.1.2.1. Finite element model

Finite element method **(**FEM**)** is a widely used method for numerically solving differential equations arising in engineering and mathematical modelling. The FEM is a general numerical method for solving partial differential equations in two or three space variables. To solve a problem, the FEM subdivides a large system into smaller, simpler parts that are called finite elements. This is achieved by a particular space discretization in the space dimensions, which is implemented by the construction of a mesh of the object: the numerical domain for the solution, which has a finite number of points. The finite element method formulation of a boundary value problem finally results in a system of algebraic equations. The method approximates the unknown function over the domain. The simple equations that model these finite elements are then assembled into a larger system of equations that models the entire problem. The FEM then approximates a solution by minimizing an associated error function via the calculus of variations (source Wikipedia).

For this model, space frame element was used,[2].

1.1.2.2. Genetic algorithms

According to Wikipedia, in computer science and operations research, a genetic algorithm (GA) is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on biologically inspired operators such as mutation, crossover, and selection. In a genetic algorithm, a population of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem is evolved toward better solutions. Each candidate solution has a set of properties (its chromosomes or genotype) which can be mutated and altered; traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible (with e.g., continuous variables). The evolution usually starts from a population of randomly generated individuals, and is an iterative process, with the population in each iteration called a generation. In each generation, the fitness of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved. The more fit individuals are stochastically selected from the current population, and everyone’s genome is modified (recombined and possibly randomly mutated) to form a new generation. The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

A good introductory book for GAs, extensively used in this article, is [3]. It uses Python as a primary programming language and DEAP framework for implementing genetic algorithms. DEAP is a novel evolutionary computation framework for rapid prototyping and testing of ideas. It seeks to make algorithms explicit and data structures transparent. It works in perfect harmony with parallelization mechanism such as multiprocessing and SCOOP, [4].

1.1.2.3. Combining FEM and GA

The main idea is to use genetic algorithms with fitness function that of finite element analysis.

1.1.3. Model and implementation

Bionic partition for Airbus A 320 cabin interiors is well known achievement of design by genetic algorithms, [5]. Here, I tried to recreate the results at 'schematic' scale.

The objective was to minimise the weight of the structure (total length of all elements) and maximise the strength (minimise maximum out of plane displacement) for space frame structure.

Some steps of the modelling are described below:

a) Define a single objective, minimising fitness strategy:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

b) Create the Individual class based on list:

```
creator.create("Individual", list, fitness=creator.FitnessMin)
```

c) Create an operator that randomly returns 0 or 1:

```
toolbox.register("zeroOrOne", random.randint, 0, 1)
```

d) Create the individual operator to fill up an Individual instance:

```
toolbox.register("individualCreator",
                 tools.initRepeat,
                 creator.Individual,
                 toolbox.zeroOrOne,
                 len(possible_lines))
```

e) Create the population operator to generate a list of individuals:

```
toolbox.register("populationCreator",
                 tools.initRepeat,
                 list,
                 toolbox.individualCreator)
```

f) Then, we instruct the genetic algorithm to use the FEM method instance for fitness evaluation:

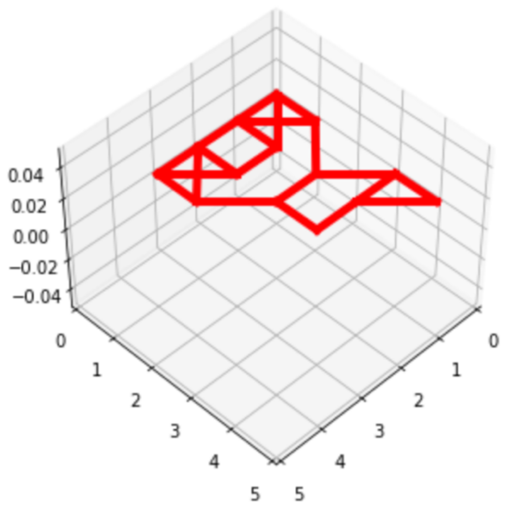
```
def staticFEM(individual):

    coord, elcon, bc_u_elim, f_after_u_elim = utils(individual)

    try:
        FEA_output_arr=FEA(coord, elcon, bc_u_elim, f_after_u_elim)
    except:
        return PENALTY_VALUE,

    strength=max_u(FEA_output_arr)

    weight=total_length(coord,elcon)

    return weight*abs(strength),  # return a tuple
```

g) Set genetic operators:

```
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit,
indpb=1.0/len(possible_lines))
```

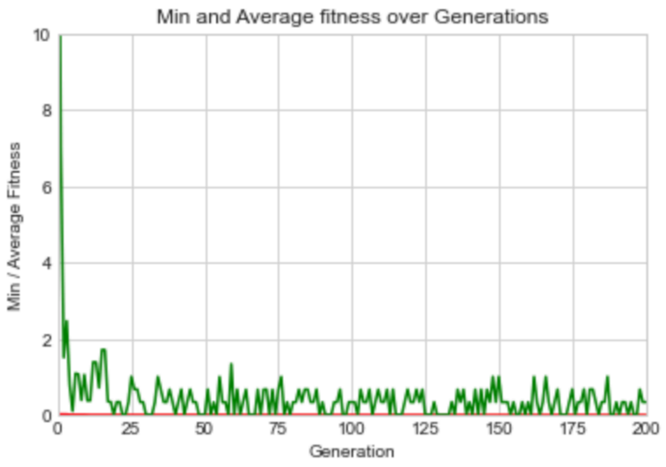
1.1.4. Final words

Results of the modelling is the produced design of 'Bionic partition'.



```
-- Displacement (Strength)--
-0.0005006090518706204

-- Length (Weight)--
24.313708498984763
```

The progress during optimization can be seen below.



1.1.5. Bibliography

[1] Generative Design: Advanced Design Optimization Processes for Aeronautical Application, S. Bagassi et al., ICAS 2016
[2] MATLAB Guide to Finite Elements. An Interactive Approach, Peter I. Kattan, 2nd edition
[3] Hands-On Genetic Algorithms with Python: Applying genetic algorithms to solve real-world deep learning and artificial intelligence problems. Wirsansky, Eyal.
[4] https://deap.readthedocs.io/en/master/
[5] https://www.youtube.com/watch?v=IxF1FItQV4Y

1.2. Generative design of a bridge like structure by genetic algorithms and finite element analysis

1.2.1 Background and rationale

Two-dimensional bridge-like plane truss fixed at its leftmost and rightmost bottom nodes was loaded in the middle node in downward vertical direction. Optimization objective was to minimize the middle node displacement.

1.2.2 Methodology

1.2.2.1 Finite element model

For this model, plane truss element was used,[1].

1.2.2.2 Combining FEM and GA

The main idea is to use genetic algorithms with fitness function that of finite element analysis. GA implementation was from [2].

1.2.3 Model and implementation

The following steps describe the main parts of GA program:

a) We start by setting the lower and upper boundary for each of the values representing some bridge intermediate nodes:

```
BOUNDS_LOW =  [0, 0, 2, 5, 5, 5]
BOUNDS_HIGH = [5, 5, 9, 12, 14, 14]
```

b) Since our goal is to minimize the middle node displacement, we define a single objective, minimization fitness strategy (because of negative displacement, it is maximization):

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

c) Now comes a particularly interesting part: since the solution is represented by a list of float values, each of a different range, we use the following loop to iterate over all pairs of lower-bound, upper-bound values. For each entry in data, we create a separate toolbox operator, which will be used to generate random float values in the appropriate range:

```
for i in range(NUM_OF_PARAMS):
    toolbox.register("hyperparameter_" + str(i),
                     random.uniform, BOUNDS_LOW[i], BOUNDS_HIGH[i])
```

d) Then, we create the parameter tuple, which contains the separate float number generators we just created for each parameter:

```
for i in range(NUM_OF_PARAMS):
    hyperparameters = hyperparameters +
(toolbox.__getattribute__("hyperparameter_" + str(i)),)
```

e) Now, we can use this parameter tuple, in conjunction with DEAP's built-in initCycle() operator, to create a new individualCreator operator that fills up an individual instance with a combination of randomly generated parameter values:

```
toolbox.register("individualCreator",
                 tools.initCycle,
                 creator.Individual,
                 hyperparameters,
                 n=1)
```

f) Then, we instruct the genetic algorithm to use the FEM method instance for fitness evaluation:

```
def femStatic(individual):
    x1=individual[0]
    x2=individual[1]
    x3=individual[2]
    x5=individual[3]
    x6=individual[4]
    x7=individual[5]

    coord=np.array([0.0,0.0,
                    x1,3.0,
                    x2,0.0,
                    x3,3.0,
                    7.0,0.0,
                    x5,3.0,
                    x6,3.0,
                    x7,0.0,
                    14.0,0.0]).reshape(9,2)

    displ = FEA_u(coord,
              elcon=np.array([[0, 1],[0, 2],[1, 2],[1, 3],[2, 3],
                              [2, 4],[3, 4],[3, 5],[4, 5],[4, 7],
                              [5, 7],[5, 6],[6, 7],[6, 8],[7, 8]]),
                              bc_u_elim=[0,1,16,17],

f_after_u_elim=np.array([0,0,0,0,0,0,0,-10,0,0,0,0,0,0]),
                              A=1, E=2e5)
    return displ[9],
```

g) Now, we need to define the genetic operators. While, for the selection operator, we use the usual tournament selection with a tournament size of 2, we choose crossover and mutation operators that are specialized for bounded float-list chromosomes and provide them with the boundaries we defined for each parameter:

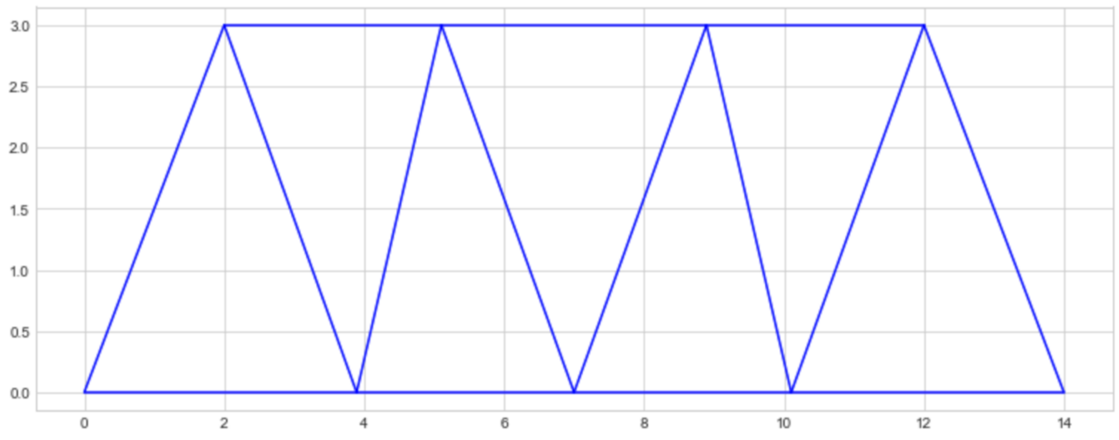
```
toolbox.register(“select”, tools.selTournament, tournsize=2)
 toolbox.register(“mate”,
 tools.cxSimulatedBinaryBounded,
 low=BOUNDS_LOW,
 up=BOUNDS_HIGH,
 eta=CROWDING_FACTOR)toolbox.register(“mutate”,

tools.mutPolynomialBounded,
 low=BOUNDS_LOW,
 up=BOUNDS_HIGH,
 eta=CROWDING_FACTOR,
 indpb=1.0 / NUM_OF_PARAMS)
```

h) In addition, we use the elitist approach, where the HOF (hall-of-fame) members — the current best individuals — are always passed untouched to the next generation:

```
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)population, logbook =
elitism.eaSimpleWithElitism(population,
 toolbox,
 cxpb=P_CROSSOVER,
 mutpb=P_MUTATION,
 ngen=MAX_GENERATIONS,
 stats=stats,
 halloffame=hof,
 verbose=True)
```

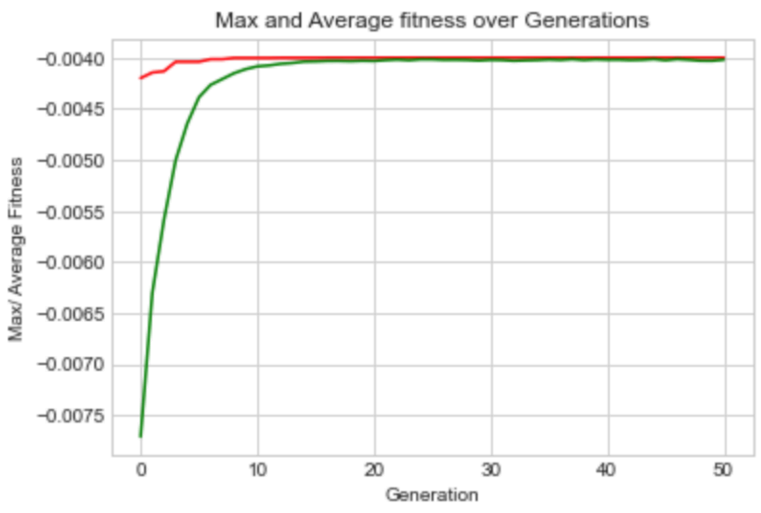
1.2.4 Final words

Results of the modelling is the optimized bridge design:

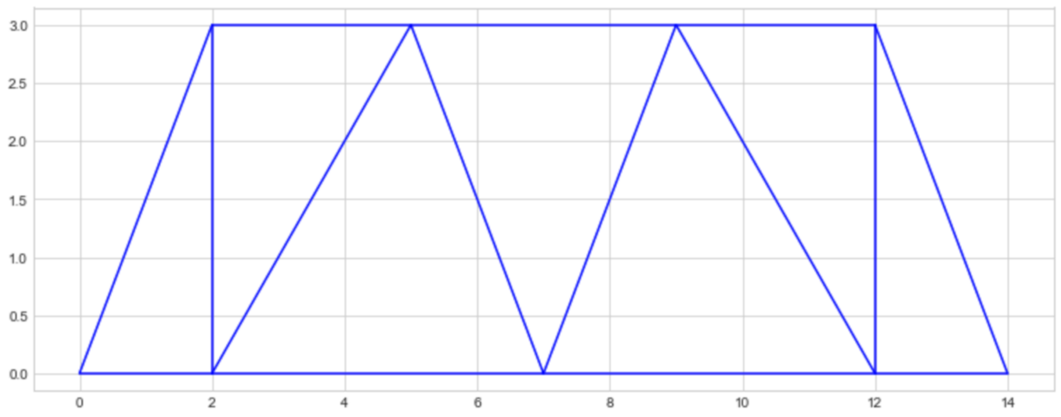


The progress during optimization can be seen below.

```
Double check:
displ -0.004000199879371854
```



Which one can compare with an initial design that has bigger middle node displacement of -0.005108382058368501.



1.2.5 Bibliography

[1] MATLAB Guide to Finite Elements. An Interactive Approach, Peter I. Kattan, 2nd edition

[2] Hands-On Genetic Algorithms with Python: Applying genetic algorithms to solve real-world deep learning and artificial intelligence problems. Wirsansky, Eyal.

2.1. Generative design of bionic partition by reinforcement learning and finite element analysis

2.1.1 Background and rationale

Deep reinforcement learning has had great success in artificial intelligence applications. Among them, beating the champion of the game of Go in 2016, mastering many Atari games and optimizing the work of data centers. In this article, I combine deep reinforcement learning and finite element analysis for the purpose of automating structural design of bionic partition for Airbus A 320 cabin interiors,[1].

AI in general and deep reinforcement learning in particular are powerful approaches in solving many nowadays problems in information technology, business, healthcare, and engineering. There is a myriad of applications for AI technologies that one can implement to make life easier. Structural engineering design is no exception. Designing a structure or a part of machinery is a very tiring process. One needs to make a lot of manual changes before resulting in the final design that satisfies structural loads. But this iterative process can be automated.

A typical approach to structural engineering design is finite element analysis. Several authors have tried to combine finite element analysis and machine learning [2-4]. For example, [3] have used deep-autoencoder to approximate the large deformations on a non-linear, muscle actuated beam. In [4], machine learning was used to predict the deformation of the breast tissues during the compression. However, little attention has been paid to using reinforcement leaning in assisting structural engineering design. Most resent attempt to apply reinforcement learning to topology optimization can be found in a work of Brown et al in [5].

2.1.2 Methodology

2.1.2.1 Finite element model

For this model, space frame element was used,[6].

2.1.2.2 Reinforcement learning

Reinforcement learning (RL) can be understood by using the concepts of agents, environments, states, observations, actions, and rewards. A reinforcement learning agent interacts with its environment in discrete time steps. At each time step, the agent receives an observation, which typically includes the reward. It then chooses an action from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state and the reward associated with transition is determined. The goal of a reinforcement learning agent is to collect as much reward as possible. The advantage of reinforcement learning is that one does not have to provide labeled data for training, and its generalizability. Reinforcement learning system learns by maximizing rewards with no supervision.

2.1.2.3 FEM to RL agent interaction

The finite element model represents an environment to which a RL agent applies actions and from which it gets observations and rewards. The agent uses neural network to decide on its actions. Actions change geometry of the structure or a component, and the resulting geometry is then subjected to FEA. Finite element analysis produces the state, which is then fed to neural network and the process repeats itself. The agent gets rewards if it meets the optimization objective of minimizing (weight) and/or maximizing (stiffness) target values. The outcome of the modeling is an optimized design of the component. The ‘inference stage’ is a usual a ‘predict’ function for a neural network where the RL agent makes greedy actions of altering the geometry based on observations only.

2.1.3 Model and implementation

The following steps describe the main parts of RL-FEA program:

a) Define FE model, action space, rewards, observations, and allowable moves of an agent.

b) Set up Gym (Python library) environment:

```python
class BionicEnv(gym.Env):

    metadata = {"render.modes": ["human"]}

    def __init__(self):
        super().__init__()
        self.action_space = spaces.Discrete(N_DISCRETE_ACTIONS)
        self.M=Model()
        self.M.reset(False,-500)
        self.x0=2
        self.y0=2
        self.obs=self.M.nn_input(self.x0,self.y0, 0)
        self.observation_space = spaces.Box(low=np.array([-1e10 for x in range(DIM)]),
                                            high=np.array([1e10 for y in range(DIM)]),
                                            shape=(DIM,),
                                           dtype=np.int64)
        self.step_=0
        self.needs_reset = True

    def step(self, action):

        x_new, y_new = self.x0, self.y0
        x_new, y_new = self.M.action_space(action, x_new, y_new)
        self.obs=self.M.nn_input(x_new,y_new,action)

        self.step_+=1
        reward=self.M.reward_(x_new,y_new,self.step_)
        self.x0,self.y0 = x_new,y_new

        done=False
        if self.M.break_flag:
            reward-=100
            done=True

        if self.needs_reset:
            raise RuntimeError("Tried to step environment that needs reset")

        if done:
            self.needs_reset = True

        return np.array(self.obs), reward, done, dict()

    def reset(self):
        self.M.reset(False,-500)

        if not self.M.flag_re:
            self.x0=random.choice([1,2,4])
            self.y0=random.choice([1,2,4])
        else:
            self.x0=3
            self.y0=3
        self.M.coord=[[self.x0,self.y0,0]]
        self.M.el_dic={(self.x0, self.y0):0}

        self.obs=self.M.nn_input(self.x0,self.y0,0)
        self.step_=0
        self.needs_reset = False
        return np.array(self.obs)

    def render(self, mode="human"):
        self.M.draw('blue')

    def close(self):
        pass
```

c) Set up Callback function to plot the training progress:

```python
class SaveOnBestTrainingRewardCallback(BaseCallback):
    """
    Callback for saving a model (the check is done every ``check_freq`` steps)
    based on the training reward (in practice, we recommend using ``EvalCallback``).

    :param check_freq: (int)
    :param log_dir: (str) Path to the folder where the model will be saved.
      It must contains the file created by the ``Monitor`` wrapper.
    :param verbose: (int)
    """

    def __init__(self, check_freq: int, log_dir: str, verbose=1):
        super().__init__(verbose)
        self.check_freq = check_freq
        self.log_dir = log_dir
        self.save_path = os.path.join(log_dir, "best_model")
        self.best_mean_reward = -np.inf

    def _init_callback(self) -> None:
        # Create folder if needed
        if self.save_path is not None:
            os.makedirs(self.save_path, exist_ok=True)

    def _on_step(self) -> bool:
        if self.n_calls % self.check_freq == 0:

            # Retrieve training reward
            x, y = ts2xy(load_results(self.log_dir), "timesteps")
            if len(x) > 0:
                # Mean training reward over the last 100 episodes
                mean_reward = np.mean(y[-100:])
                if self.verbose > 0:
                    print(f"Num timesteps: {self.num_timesteps}")
                    print(
                        f"Best mean reward: {self.best_mean_reward:.2f} - Last mean reward per episode: {mean_rew
                    )

                # New best model, you could save the agent here
                if mean_reward > self.best_mean_reward:
                    self.best_mean_reward = mean_reward
                    # Example for saving best model
                    if self.verbose > 0:
                        print(f"Saving new best model to {self.save_path}.zip")
                    self.model.save(self.save_path)

        return True
```

d) Train the model with PPO algorithm from Stable-Baselines3 (Python library)

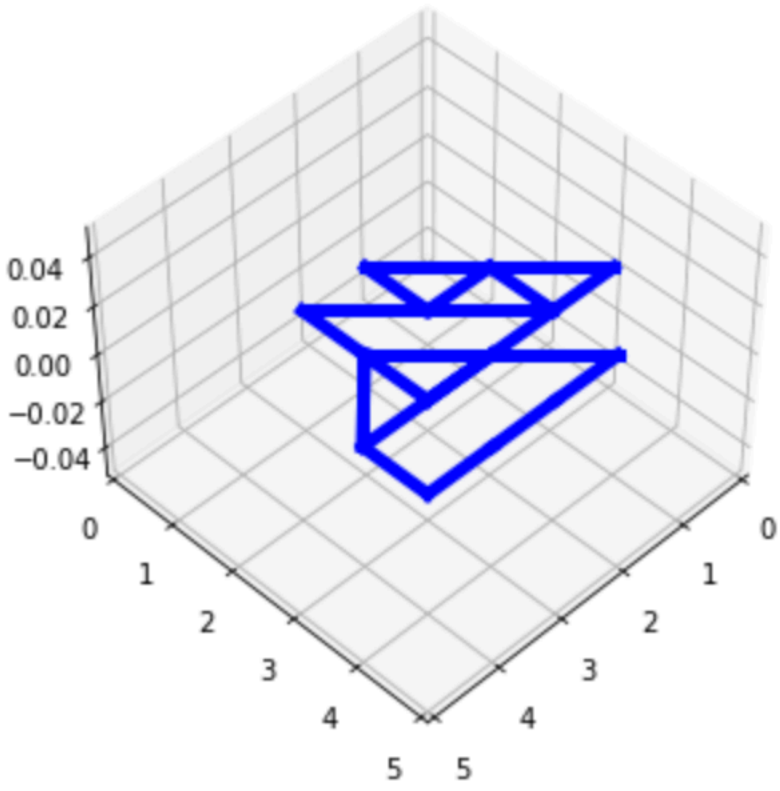
```python
start=time.time()
model = PPO("MlpPolicy", env).learn(total_timesteps=ts, callback=callback)
end=time.time()
```

2.1.4 Final words

One of the main advantages of using RL instead of conventional optimization algorithms is its ability to generalize. I have tested generalizability of the model by randomizing the agent's initial state (place in which it starts 'drawing' the geometry), and setting completely new initial states during an inference stage (the one that the agent has not seen during the training).

Also, I have included an action into state vector as doing so showed some advantages in terms of final design's characteristics.

Results of the modelling is the produced design of 'Bionic partition'.



```
env.M.length()
```

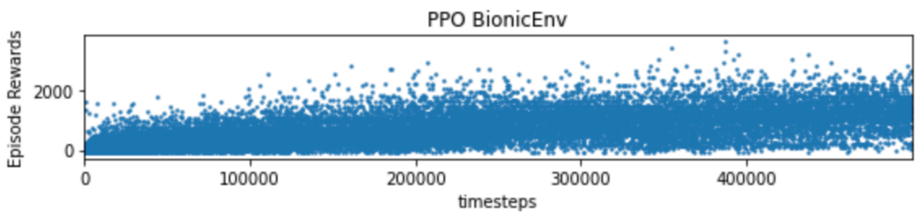
```
27.72792206135786
```

```
FEA_output_arr=env.M.FEA()
env.M.max_u(FEA_output_arr)
```

```
-0.009221648573500034
```

From the below graph it can be seen that the agent is progressing and learning. Apparently, increasing the number of learning steps should lead to better designs.

```
results_plotter.plot_results([log_dir], ts, results_plotter.X_TIMESTEPS, "PPO BionicEnv")
```



2.1.5 Bibliography

[1] https://www.youtube.com/watch?v=IxF1FItQV4Y

[2] Machine Learning and Finite Element Method for Physical Systems Modeling. O.Kononenko, I.Kononenko, arXiv.org

[3] Towards Finite-Element Simulation Using Deep Learning. Francois Roewer-Despres, Najeeb Khan, Ian Stavness, CMBBE 2018

[4] A finite element-based machine learning approach for modeling the mechanical behavior of the breast tissues under compression in real-time. Martinez-Martinez F, et al. ComputBiot

[5]https://www.sciencedirect.com/science/article/pii/S0264127522002933

[6] MATLAB Guide to Finite Elements. An Interactive Approach, Peter I. Kattan, 2nd edition

2.2. Generative design of bridge like structure by reinforcement learning and finite element analysis

2.2.1 Background and rationale

Two-dimensional bridge-like plane truss fixed at its leftmost and rightmost bottom nodes was loaded in the middle node in downward vertical direction. Optimization objective was to minimize the middle node displacement.

2.2.2 Methodology

2.2.2.1 Finite element model

For this model, plane truss element was used,[1].

2.2.2.2 FEM to RL agent interaction

An RL agent was set to receive observations and rewards from the environment (FEM).

2.2.3 Model and implementation

The following steps describe the main parts of RL-FEA program:

a) Define FE model, action space, rewards, observations, and allowable moves of an agent.

b) Set up Gym (Python library) environment.

c) Set up Callback function to plot the training progress.

d) Train the model with PPO algorithm from Stable-Baselines3 (Python library)

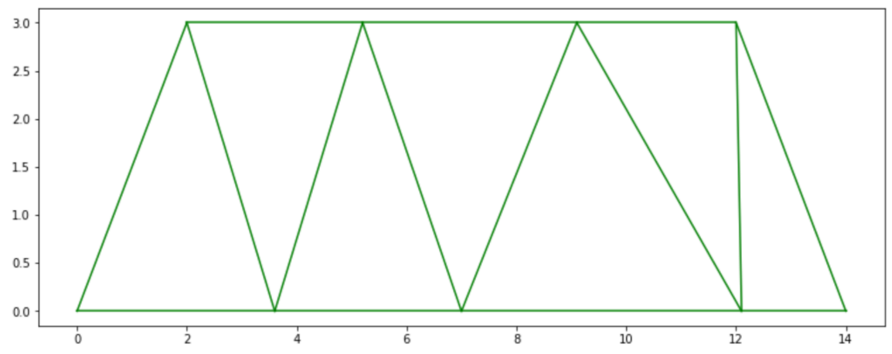
```
start=time.time()
model = PPO("MlpPolicy", env).learn(total_timesteps=ts, callback=callback)
end=time.time()
```

2.2.4 Final words

I have included an action into state vector as doing so showed some advantages in terms of the final design's characteristics.

Also, I was randomizing nodes where the agent was doing alternations during the training be better accommodate the agent's generalizability.

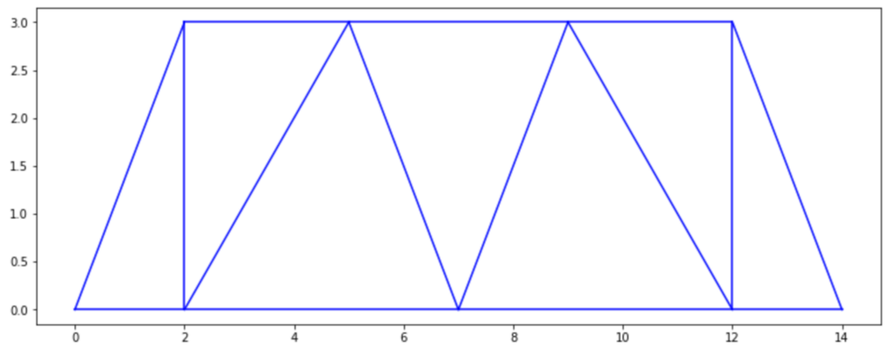
Results of the modelling is the produced design of bridge like structure:



```
obs[-2]
```

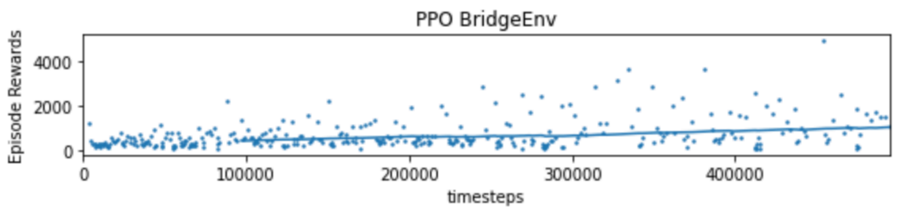
```
-0.004604604588490783
```

Which one can compare with an initial design that has bigger middle node displacement of -0.005108382058368501, hence worser stiffness.



From the below graph it can be seen the agent is progressing and learning. Apparently, increasing the number of learning steps should lead to better designs.

```
results_plotter.plot_results([log_dir], ts, results_plotter.X_TIMESTEPS, "PPO BridgeEnv")
```



2.2.5 Bibliography

[1] MATLAB Guide to Finite Elements. An Interactive Approach, Peter I. Kattan, 2nd edition

2.3. Material optimization by reinforcement learning and finite element analysis

2.3.1 Background and rationale

This model tries to optimize stress-strain state (reduce displacements to acceptable level) by combination of simple finite element analysis (axially loaded bar element) and reinforcement learning.

In essence, it is a simple form of material optimization where an area and Young's modulus of bar element are being altered during the training.

2.3.2 Methodology

2.3.2.1 Finite element model

For this model, an axially loaded bar element was used,[1].

2.3.2.2 FEM to RL agent interaction

An RL agent was set to receive observations and rewards from the environment (FEM).

2.3.3 Model and implementation

The following steps describe the main parts of RL-FEA program:

a) Define FE model, action space, rewards, observations, and allowable moves of an agent.

b) Set up Gym (Python library) environment.

```
class BarEnv(gym.Env):

    metadata = {"render.modes": ["human"]}

    def __init__(self):
        super().__init__()
        self.action_space = spaces.Discrete(N_DISCRETE_ACTIONS)

        self.A=3*random.random()
        self.E=2*random.random()
        self.obs=list(AxialBarFEM(self.A, self.E))+[0]
        self.observation_space = spaces.Box(low=np.array([-np.inf for x in range(DIM)]),
                                            high=np.array([np.inf for y in range(DIM)]),
                                            shape=(DIM,),
                                           dtype=np.float64)
        self.needs_reset = True

    def step(self, action):

        obs_=self.obs
        self.A,self.E=prestep(action,self.A,self.E)
        self.obs=list(AxialBarFEM(self.A,self.E))+[action]
        reward=reward_(obs_,self.obs)

        done=False
        if self.obs[0]<0.1:
            done = True

        if self.needs_reset:
            raise RuntimeError("Tried to step environment that needs reset")

        if done:
            self.needs_reset = True

        return np.array(self.obs), reward, done, dict()

    def reset(self):
        self.A=3*random.random()
        self.E=2*random.random()
        self.obs=list(AxialBarFEM(self.A, self.E))+[0]
        self.needs_reset = False
        return np.array(self.obs)

    def render(self, mode="human"):
        pass

    def close(self):
        pass
```

c) Set up Callback function to plot the training progress.

d) Train the model with PPO algorithm from Stable-Baselines3 (Python library)

```
start=time.time()
model = PPO("MlpPolicy", env).learn(total_timesteps=ts, callback=callback)
end=time.time()
```

2.3.4 Final words

I have included an action into the state vector as doing so showed some advantages in terms of the final design’s characteristics.

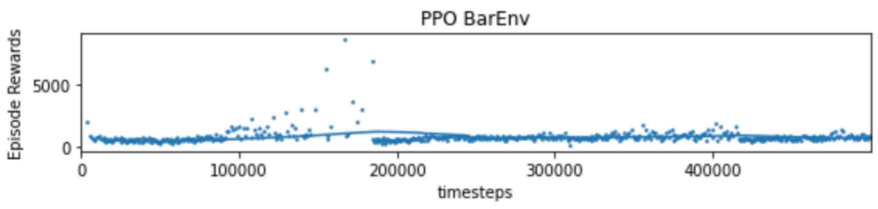
Results of the modelling can be seen below:

```
if obs[0]>THR:
    print("Bad initial parameters! Try increasing initial cross-sectional area A, Young's modulus E and/or number
elif obs[0]<0.8*THR:
    print("You can get better parameters. Try decreasing initial area A and/or Young's modulus E")
else:
    print("Solution converged! MaxDispl={}, A={},E={}".format(obs[0],obs[1],obs[2]))
```

```
Solution converged! MaxDispl=1.9062606657432373, A=2.021107671573183,E=0.34607242762415025
```

From the graph it can be seen the agent is gradually progressing and learning.

```
results_plotter.plot_results([log_dir], ts, results_plotter.X_TIMESTEPS, "PPO BarEnv")
```



This model can be seen as simple form of material optimization with RL.

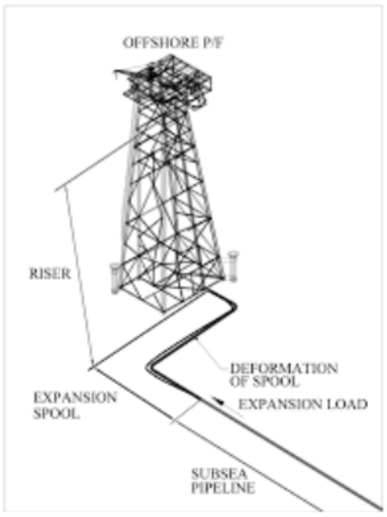
2.3.5 Bibliography

[1] MATLAB Guide to Finite Elements. An Interactive Approach, Peter I. Kattan, 2nd edition

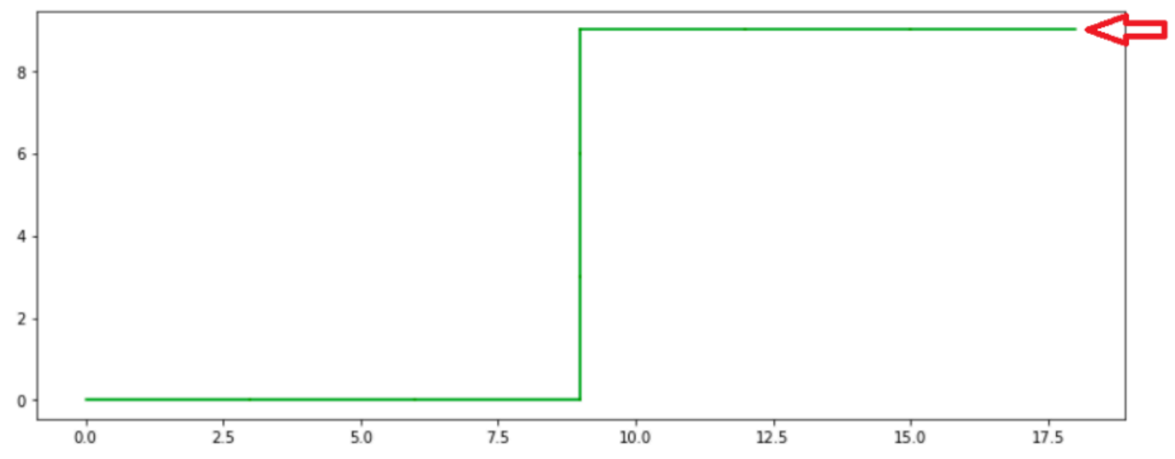
2.4. Generative design of subsea spools by reinforcement learning and finite element analysis

2.4.1 Background and rationale

Spools are used to connect the subsea pipeline with a fixed riser nearby the offshore platform.



In this model, two-dimensional spool-like plane frame was loaded at its rightmost node in compressing horizontal direction, leftmost node was completely fixed (see below). The environment state (geometry and maximum nodal displacement) was fed to an agent (neural network) which produced actions of altering geometry of a subsea spool in a certain way. The optimization objective was to minimize the maximum nodal displacement. After training, inference was used to obtain optimal geometry of a spool. The modeling has shown that the agent learned how to meet the objective.



2.4.2 Methodology

2.4.2.1 Finite element model

For this model, plane frame element was used,[1].

2.4.2.2 FEM to RL agent interaction

An RL agent was set to receive observations and rewards from the environment (FEM).

2.4.3 Model and implementation

The following steps describe the main parts of RL-FEA program:

a) Define FE model, action space, rewards, observations, and allowable moves of an agent.

b) Set up Gym (Python library) environment.

c) Set up Callback function to plot the training progress.

d) Train the model with PPO algorithm from Stable-Baselines3 (Python library)

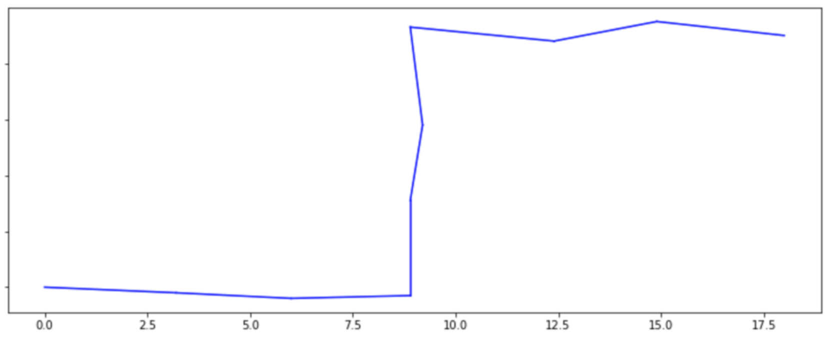
```
start=time.time()
model = PPO("MlpPolicy", env).learn(total_timesteps=ts, callback=callback)
end=time.time()
```

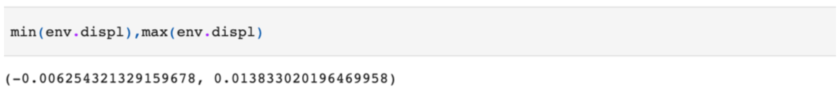
2.4.4 Final words

I have included an action into the state vector as doing so showed some advantages in terms of final design's characteristics.

Also, I was randomizing nodes where the agent was doing alternations during the training be better accommodate the agent's generalizability.

Results of the modelling is the produced design of spool like structure:

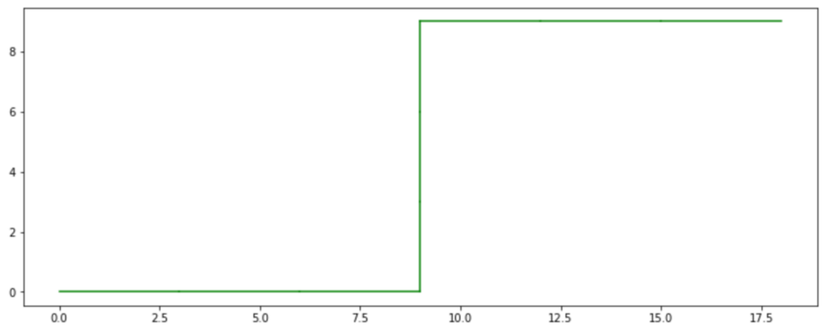




```
min(env.displ),max(env.displ)
```

```
(-0.006254321329159678, 0.013833020196469958)
```

Which one can compare with an initial design that has bigger middle node (min, max) displacement of (-0.9257571525764527, 1.3885714431377043).



2.4.5 Bibliography

[1] MATLAB Guide to Finite Elements. An Interactive Approach, Peter I. Kattan, 2nd edition

2.5. Generative design by AlphaZero/MCTS and finite element analysis

Disclaimer: These models are now available only in older commits of my Gigala GitHub repository (e.g., in 623d866f83943dc8193db15ac04afadba3b50f86)

2.5.1 Background and rationale

AlphaZero is a computer program developed by artificial intelligence research company Google DeepMind to master the games of chess, shogi and go. It uses RL, Monte Carlo tree search (MCTS), and self-play to improve its policy, hence was a good fit to test the task of generative design.

2.5.2 Methodology

2.5.2.1 Finite element model

For this model, space frame element was used,[1].

2.5.2.2 AlphaZero/MCTS algorithms

AlphaZero is a computer program developed by artificial intelligence research company Google DeepMind to master the games of chess, shogi and go. The algorithm uses an approach like AlphaGo Zero. On December 5, 2017, the DeepMind team released a preprint introducing AlphaZero, which within 24 hours achieved a superhuman level of play in these three games by defeating world-champion programs, Stockfish, elmo, and the 3-day version of AlphaGo Zero, [2].

In computer science, Monte Carlo tree search (MCTS) is a heuristic search algorithm for some kinds of decision processes, most notably those employed in game play. MCTS has been used for decades in computer Go programs. It has been used in other board games like chess and shogi games with incomplete information such as bridge and poker, as well as in real-time video games (such as Total War: Rome II's implementation in the high-level campaign AI), [3].

2.5.2.3 Finite Element to AlphaZero/MCTS

The finite element model represents an environment to which an agent applies actions and from which it gets rewards. The agent uses AlphaZero/MCTS algorithm to decide on its actions. Actions change geometry of the structure of a bionic partition; the resulting geometry is then subjected to FEA. The agent gets rewards if it meets the optimization objective of minimizing (weight) and maximizing (stiffness) target values. The outcome of the modeling (produced ‘online’ during the ‘game’ after the training of AlphaZero algorithm) is an optimized design of the component.

2.5.3 Model and implementation

In this work, I made an agent do actions of drawing elements between grid nodes. For 5x5 grid, there were 72 possible elements (actions). I applied AlphaZero algorithm in a game between two players. Rules of the game were as follows: whoever drew a structure that passed through certain nodes (checkpoints), produced connected structure, had at least two neighbors for each node and an improved strength and weight compared to the previous move, won. I decided to apply the algorithm in a form of a game between two players because of the ability to replace one player with a human engineer to assist an RL agent in engineering design, and because I was interested in how the agents would behave (when creating a structure) in competitive game from the standpoint of the game theory. The overall objective of the agents was to minimize the structure’s weight while maximizing its strength. It took a while to train AlphaZero algorithm for about 250 iterations.

I also tried to play a ‘two-boards’ game where instead of building one structure on a single board by two RL agents, as above, I made two separate RL agents compete in building two separate structures on two separate boards. Hence, RL agents’ action did not block each other’s. The overall objective of the agents was to minimize the structures’ weights while maximizing their strengths. Whoever did it better - won. It took a while to train AlphaZero algorithm for 150 iterations.

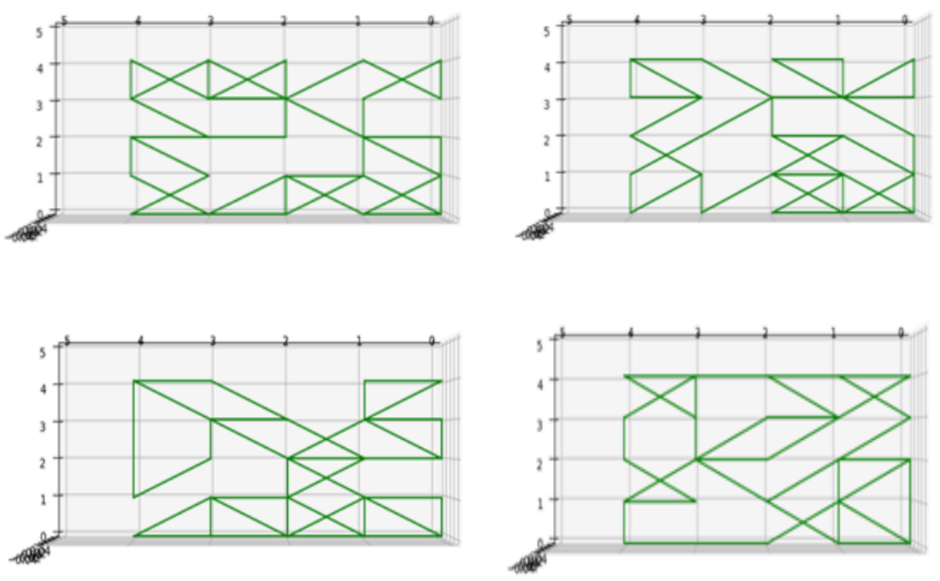
Finally, I applied MCTS in a game between two players with the same rules. I decided to apply algorithm in a form of a game between two players because of the ability to replace one player with a human engineer to assist an agent in engineering design. The overall objective of the agent was to minimize the structure’s weight while maximizing its strength. It took a while to complete the game if the number of playouts was big. The bigger the number of playouts were, the more intelligent the agent were.

2.5.4 Final words

AlphaZero player after appropriate training is much smarter than pure MCTS player (it has much more wins than pure MCTS). Typical game between AlphaZero and MCTS players takes on average more steps than a random play. Results of the modeling (the game between AlphaZero and MCTS players) show that agents are usually capable of producing valid outcomes within 45 game steps (win limit). Codebase for the model can be found at my Gigala GitHub page (in old commits). I borrowed the code for the AlphaZero algorithm from [4].

For ‘two-boards’ game, results of the modeling have shown that RL agents, in the current formulation of the problem, are more inclined to optimize for strength. However, the model might produce different results with more training and hyperparameters tuning and/or can be easily adjusted to optimize for weight.

For MCTS, results of the modeling show that an agent is usually capable of producing valid outcomes within 40 game steps (I incentivized the agent to do so by adjusting the parameter of win limit) with 500 playouts. Sample results for the MCTS algorithm can be seen below:



2.5.5 Bibliography

[1] MATLAB Guide to Finite Elements. An Interactive Approach, Peter I. Kattan, 2nd edition

[2] https://en.wikipedia.org/wiki/AlphaZero

[3] https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

[4] https://github.com/junxiaosong/AlphaZero_Gomoku