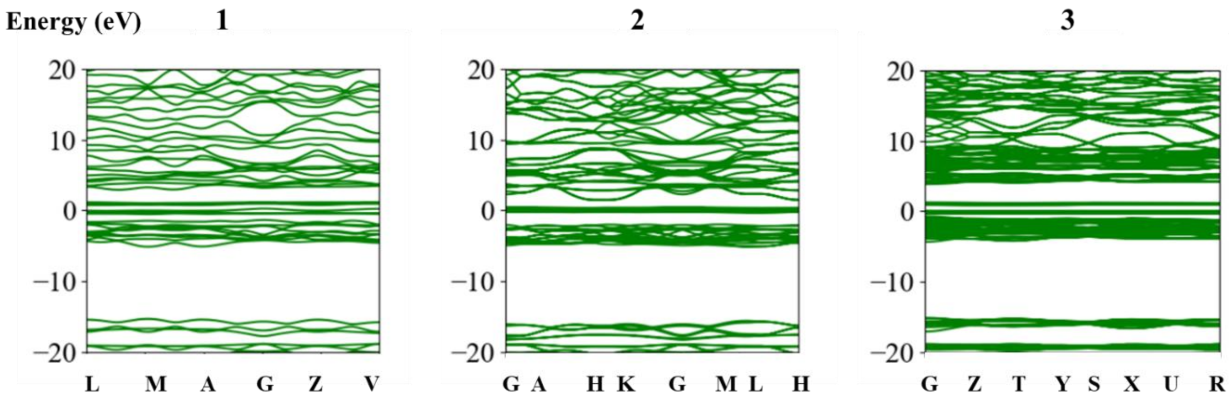
Supplementary Materials

Data-driven exploration and first-principles analysis of perovskite material

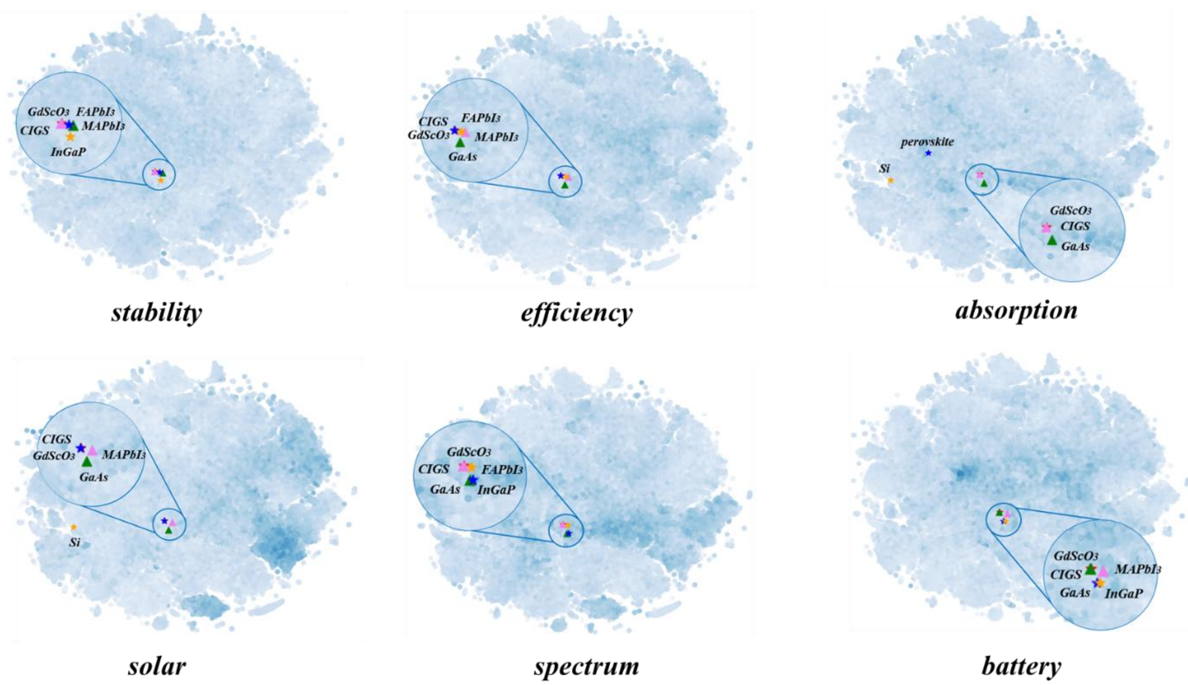
Lei Zhang[*]**, Jiacheng Zhou, Xuexiao Chen**

Department of Materials Physics, School of Chemistry and Materials Science, Nanjing University of Information Science & Technology, Nanjing 210044, Jiangsu, China.

[*]**Correspondence to:** Prof. Lei Zhang, Department of Materials Physics, School of Chemistry and Materials Science, Nanjing University of Information Science & Technology, 219 Ning Liu Road, Nanjing 210044, Jiangsu, China. E-mail: 002699@nuist.edu.cn



Supplementary Figure 1. Band structures of **1-3** with energy range from -20 eV to 20 eV of the predicted new structures **1-3** of $GdScO_3$.



Supplementary Figure 2. Visualization of language model for $GdScO_3$ applications (t-SNE from language model using 1.18 million scientific articles): stability, efficiency, absorption, solar, spectrum and battery.

Supplementary notes

Skip-gram method

The skip-gram method is a fundamental technique in the Word2Vec algorithm, used for learning word embeddings in natural language processing. This method is designed to predict the surrounding context words given a central target word within a sentence. In essence, the skip-gram model seeks to maximize the probability of the context words appearing around a specified target word, thereby capturing semantic relationships between words. The process begins with a large corpus of text, where the goal is to predict the context words surrounding each target word. For instance, in the sentence "The quick brown fox jumps over the lazy dog," if "fox" is the target word, the context words might include "quick," "brown," "jumps," and "over," depending on the context window size. The skip-gram model employs a shallow neural network with a single hidden layer. In this network, the input layer represents the target word, while the output layer provides a probability distribution over the vocabulary for the context words. During training, the model adjusts the weights to increase the likelihood of the actual context words given the target word. This adjustment is achieved through stochastic gradient descent and backpropagation.

In materials science, consider the sentence "The titanium dioxide exhibits high photocatalytic activity under UV light." If "titanium" is the target word, the skip-gram model predicts surrounding context words such as "dioxide," "exhibits," and "activity," depending on the context window size. For instance, with a window size of 2, the model would predict "dioxide" and "exhibits" based on the target word "titanium." By training on a large corpus, the model learns to represent words like "titanium" and "dioxide" in a vector space that reflects their semantic relationships.

Overall, the skip-gram method is highly effective in generating dense vector representations of words, where words with similar meanings are represented by vectors that are close to each other in the vector space. This capability makes it a valuable tool for various applications in natural language processing, including semantic analysis, machine translation, and information retrieval.

Paper collection

Criteria and process for selecting the 50,000 papers from SpringerLink: the papers are published between 2010 and 2020 to ensure that the data is relevant to current research trends. The search is specifically filtered to include papers within the domain of materials science, ensuring that the collected abstracts are pertinent to the objectives of this study. The keywords “material”, “chemical”, and “physics” are used to encompass a broad spectrum of topics within materials science, chemistry, and physics that are relevant to the research. More detailed description of the criteria for selecting the papers from SpringerLink is provided in the literature.[1–3] By applying these criteria, the selected papers are considered to be relevant to the research focus in materials science.

Background of genetic process

Symbolic algorithm is employed to simulate the process of evolution. The algorithm usually includes subtree mutation, point mutation and hoist mutation. Subtree mutation is a commonly used genetic algorithm operator that modifies a portion of the chromosome's subtree structure. By randomly selecting a subtree from the parent individual and replacing it with a randomly generated new subtree, subtree mutation introduces new combinations of genes and structures, increasing the algorithm's search space and promoting population diversity. Point mutation is another genetic algorithm operator that introduces new gene mutations by randomly changing the value of a gene position on the chromosome. Point mutation can introduce local changes in the chromosome, potentially allowing the algorithm to explore different solutions in the search space. Hoist mutation is a special kind of mutation operator used to modify individuals represented as mathematical expressions, which involves selecting a subtree and promoting it to a higher level, making the individual's structure more complex. This operation can increase the diversity of individuals during the genetic algorithm search process and may lead to the discovery of better solutions. By incorporating these genetic algorithm mutation operators, such as subtree mutation, point mutation, and hoist mutation, appropriate variations and diversity can be introduced, helping the optimization algorithm discover better solutions in the search space.

Input files (parameters for Genetic crystal structure prediction)

```
#GAsearch of fixed composition GdScO3
formulaType: fix
structureType: bulk
pressure: 0
initSize: 20             # number of structures of 1st generation
popSize: 20               # number of structures of per generation
numGen: 10                  # number of total generation
```

```
saveGood: 3                # number of good structures kept to the next generation
#structure parameters
symbols: ["Gd", "Sc", "O"]
formula: [1, 1, 3]
min_n_atoms: 5                 # minimum number of atoms per unit cell
max_n_atoms: 5                 # maximum number of atoms per unit cell
spacegroup: [2-230]
d_ratio: 0.6
volume_ratio: 3
#GA parameters
rand_ratio: 0.3                # fraction of random structures per generation (except 1st gen.)
add_sym: True                  # add symmetry to each structure during evolution
#main calculator settings
MainCalculator:
  calculator: 'vasp'
  jobPrefix: ['VASP1', 'VASP2', 'VASP3', 'VASP4']
  #vasp settings
  xc: PBE
  ppLabel: ["","_sv_GW",'_s']
  #parallel settings
  numParallel: 10              # number of parallel jobs
  numCore: 40                  # number of cores
  queueName: e52692v2ib!
  mode: serial
```

GA codes in MAGUS:

The GA algorithm code is provided in the GitHub website (MAGUS). The GA process is integrated in the MAGUS crystal structure prediction process. More codes are provided in https://github.com/Zhang-NJ-Lab/GdScO3_CSP/tree/main/generators. Exemplar codes (ga.py) describing the GA algorithm are:

TODO

```
# how to set k in edom
import logging
import numpy as np
from magus.utils import *
import prettytable as pt
from collections import defaultdict
import yaml
# from .reconstruct import reconstruct, cutcell, match_symmetry, resetLattice


log = logging.getLogger(__name__)


###################################
# How to select parents?
#
# How Evolutionary Crystal Structure Prediction Works—and Why.
#      Acc. Cheminp#      The Journal of Chemical Physics 141, 044711 (2014).
#
# For now, we use a scheme similar to oganov's, because it just use rank information and can be easily extend to multi-target search.
###################################

def f_prob(func_name = 'exp', k = 0.3):

    def exp(dom):
        return np.exp(-k * dom)
    def liner(dom):
        """
        [https://doi.org/10.1063/1.3097197]
        p[i] = p1 - (i - 1) p1 / c ; recommand value c: 2/3 population size
```

```
        """
        return 1 - (dom - 1) / (k * len(dom))

    if func_name == 'exp':
        return exp
    elif func_name == 'liner':
        return liner
    else:
        raise Exception("Unknown function name {}".format(func_name))



class GAGenerator:
    def __init__(self, op_list, op_prob, **parameters):
        Requirement = ['pop_size', 'n_cluster']
        Default={'rand_ratio': 0.3, 'add_sym': True, 'history_punish':1.0, 'k': 0.3, 'choice_func': 'exp'}
        check_parameters(self, parameters, Requirement, Default)

        assert len(op_list) == len(op_prob), "number of operations and probabilities not match"
        assert np.sum(op_prob) > 0 and np.all(op_prob >= 0), "unreasonable probability are given"
        self.op_list = op_list
        self.op_prob = op_prob / np.sum(op_prob)

        self.gen = 1

    def __repr__(self):
        ret = self.__class__.__name__
        ret += "\n-------------------"
        c, m = "\nCrossovers:", "\nMutations:"
```

```
        for op, prob in zip(self.op_list, self.op_prob):
            if op.n_input == 1:
                m += "\n {}: {:>5.2f}%".format(op.__class__.__name__.ljust(20, ' '), prob *
100)
            elif op.n_input == 2:
                c += "\n {}: {:>5.2f}%".format(op.__class__.__name__.ljust(20, ' '), prob *
100)
        ret += m + c
        ret += "\nRandom Ratio                  : {:.2%}".format(self.rand_ratio)
        ret += "\nNumber of cluster        : {}".format(self.n_cluster)
        ret += "\nAdd symmertry                : {}".format(self.add_sym)
        if self.history_punish != 1.0:
            ret += "\nHistory punishment      : {}".format(self.history_punish)
        ret += "\nSelection function       {}; k = {}".format(self.choice_func, self.k)
        ret += "\n-------------------\n"
        return ret

    @property
    def n_next(self):
        return int(self.pop_size * (1 - self.rand_ratio))

    def get_parents(self, pop, n_input):
        if n_input == 1:
            return self.get_ind(pop)
        elif n_input == 2:
            return self.get_pair(pop)

    def get_pair(self, pop, n_try=50):
        history_punish = self.history_punish
        assert 0 < history_punish <= 1, "history_punish should between 0 and 1"
```

```
        dom = np.array([ind.info['dominators'] for ind in pop])
        edom = (f_prob(k = self.k))(dom)
        used = np.array([ind.info['used'] for ind in pop])
        labels, _ = pop.clustering(self.n_cluster)
        fail = 0

        while fail < n_try:
            label = np.random.choice(np.unique(labels))
            indices = np.where(labels == label)[0]
            if len(indices) < 2:
                fail += 1
                continue
            prob = edom[indices] * history_punish ** used[indices]
            prob = prob / sum(prob)
            i, j = np.random.choice(indices, 2 , p=prob)
            pop[i].info['used'] += 1
            pop[j].info['used'] += 1
            return pop[i].copy(), pop[j].copy()

        indices = np.arange(len(pop))
        prob = edom[indices] * history_punish ** used[indices]
        prob = prob / sum(prob)
        i, j = np.random.choice(indices, 2 , p=prob)
        pop[i].info['used'] += 1
        pop[j].info['used'] += 1
        return pop[i].copy(), pop[j].copy()

    def get_ind(self, pop):
        history_punish = self.history_punish

        dom = np.array([ind.info['dominators'] for ind in pop])
```

```
        edom = (f_prob(k = self.k))(dom)
        used = np.array([ind.info['used'] for ind in pop])
        prob = edom * history_punish ** used
        prob = prob / sum(prob)
        choosed = []
        i = np.random.choice(len(pop), p=prob)
        pop[i].info['used'] += 1
        return pop[i].copy()

    def generate(self, pop, n):
        log.debug(self)
        # Add symmetry before crossover and mutation
        if self.add_sym:
            pop.add_symmetry()
        newpop = pop.__class__([], name='init', gen=self.gen)
        op_choosed_num = [0] * len(self.op_list)
        op_success_num = [0] * len(self.op_list)
        # Ensure that the operator is selected at least once
        # for i, op in enumerate(self.op_list):
        #     op_choosed_num[i] += 1
        #     cand = self.get_parents(pop, op.n_input)
        #     newind = op.get_new_individual(cand)
        #     if newind is not None:
        #         op_success_num[i] += 1
        #         newpop.append(newind)
        while len(newpop) < n:
            i = np.random.choice(len(self.op_list), p=self.op_prob)
            op_choosed_num[i] += 1
            op = self.op_list[i]
            cand = self.get_parents(pop, op.n_input)
            newind = op.get_new_individual(cand)
```

```
            if newind is not None:
                op_success_num[i] += 1
                newpop.append(newind)
        table = pt.PrettyTable()
        table.field_names = ['Operator', 'Probability ', 'SelectedTimes', 'SuccessNum']
        for i in range(len(self.op_list)):
            table.add_row([self.op_list[i].descriptor,
                           '{:.2%}'.format(self.op_prob[i]),
                           op_choosed_num[i],
                           op_success_num[i]])
        log.info("OP infomation: \n" + table.__str__())
        newpop.check()
        return newpop

    def select(self, pop, num):
        if num < len(pop):
            pop = pop[np.random.choice(len(pop), num, False)]
        return pop

    def get_next_pop(self, pop, n_next=None):
        # calculate dominators before choose structures
        pop.del_duplicate()
        pop.calc_dominators()
        n_next = n_next or self.n_next
        self.gen += 1
        newpop = self.generate(pop, n_next)
        return self.select(newpop, n_next)

    def save_all_parm_to_yaml(self):
        d = {}
        for op, prob in zip(self.op_list, self.op_prob):
```

```
            d[op.__class__.__name__] = {}
            d[op.__class__.__name__]['prob'] = float(prob)
            for k in op.Default.keys():
                d[op.__class__.__name__][k] = getattr(op, k)

        d['rand_ratio'] = self.rand_ratio
        d['n_cluster'] = self.n_cluster
        d['add_sym'] = self.add_sym
        d['history_punish'] = self.history_punish
        d['choice_func'] = self.choice_func
        d['k'] = self.k

        with open('gaparm.yaml', 'w') as f:
            f.write(yaml.dump(d))
        return


class AutoOPRatio(GAGenerator):
    def __init__(self, op_list, op_prob, **parameters):
        Default = {'good_ratio': 0.6, 'auto_random_ratio': True}
        check_parameters(self, parameters, [], Default)
        super().__init__(op_list, op_prob, **parameters)

    def change_op_ratio(self, pop):
        total_nums = defaultdict(int)
        good_nums = defaultdict(int)
        for ind in pop:
            origin = ind.info['origin']
            if origin == 'seed':
                continue
            if not self.auto_random_ratio and origin == 'random':
```

```
                    continue
                total_nums[origin] += 1
                if ind.info['dominators'] < len(pop) * self.good_ratio:
                    good_nums[origin] += 1
        op_grade = {op: good_nums[op] ** 2 / total_nums[op] for op in total_nums if
total_nums[op] > 0}
        table = pt.PrettyTable()
        table.field_names = ['Operator', 'Total ', 'Good', 'Grade']
        for op in self.op_list:
            grade = op_grade[op.descriptor] if op.descriptor in op_grade else 0
            table.add_row([op.descriptor,
                           total_nums[op.descriptor],
                           good_nums[op.descriptor],
                           np.round(grade, 3)])
        if self.auto_random_ratio and self.gen > 2:
            grade = op_grade['random'] if 'random' in op_grade else 0
            table.add_row(['random', total_nums['random'], good_nums['random'],
np.round(grade, 3)])
        log.debug("OP grade: \n" + table.__str__())
        if self.auto_random_ratio and self.gen > 2:
            if 'random' not in op_grade:
                op_grade['random'] = 0
            self.rand_ratio = 0.5 * (op_grade['random'] / sum(op_grade.values()) +
self.rand_ratio)
            del op_grade['random']
        for i, op in enumerate(self.op_list):
            if op.descriptor in op_grade:
                self.op_prob[i] = 0.5 * (op_grade[op.descriptor] / sum(op_grade.values()) +
self.op_prob[i])
            else:
                self.op_prob[i] = 0.5 * self.op_prob[i]
```

```
        self.op_prob /= np.sum(self.op_prob)


    def get_next_pop(self, pop, n_next=None):
        pop.calc_dominators()
        if self.gen > 1:
            self.change_op_ratio(pop)
            #self.save_all_parm_to_yaml()
        n_next = n_next or self.n_next
        newpop = self.generate(pop, n_next)
        self.gen += 1
        return self.select(newpop, n_next)
```