

Optimizing Higher Order Neural Networks Using Evolutionary Algorithms (GA, PSO, and CRO)

Introduction

In this notebook, I explore how metaheuristic optimization algorithms — Genetic Algorithm (GA), Particle Swarm Optimization (PSO), and Chemical Reaction Optimization (CRO) — can be applied to train a Pi-Sigma/Sigma-Pi neural networks. The goal is to predict the next-day stock price direction using the past 5 days' closing prices.

Higher Order Neural Network (HONNs) are a category of feed forward network, which afford nonlinear decision boundaries and offer better classification ability compared to linear neuron. They extend standard neurons by including multiplicative interactions between inputs, allowing the network to capture higher-order feature combinations.

Pi-Sigma NN (PSNN):

PSNN consists of a single layer of tuneable weights. leads MLP in terms of weights and nodes which makes the convergence analysis of the learning rules for the PSNN more accurate and tractable. The basic idea behind the network is due to the fact that a polynomial of input variables is formed by a product (π) of several weighted linear combinations (\sum) of input variables. That is why this network is called pi-sigma instead of sigma-pi.

$$y = \sigma\left(\prod_j w_{ij} x_i + \theta_{ij}\right)$$

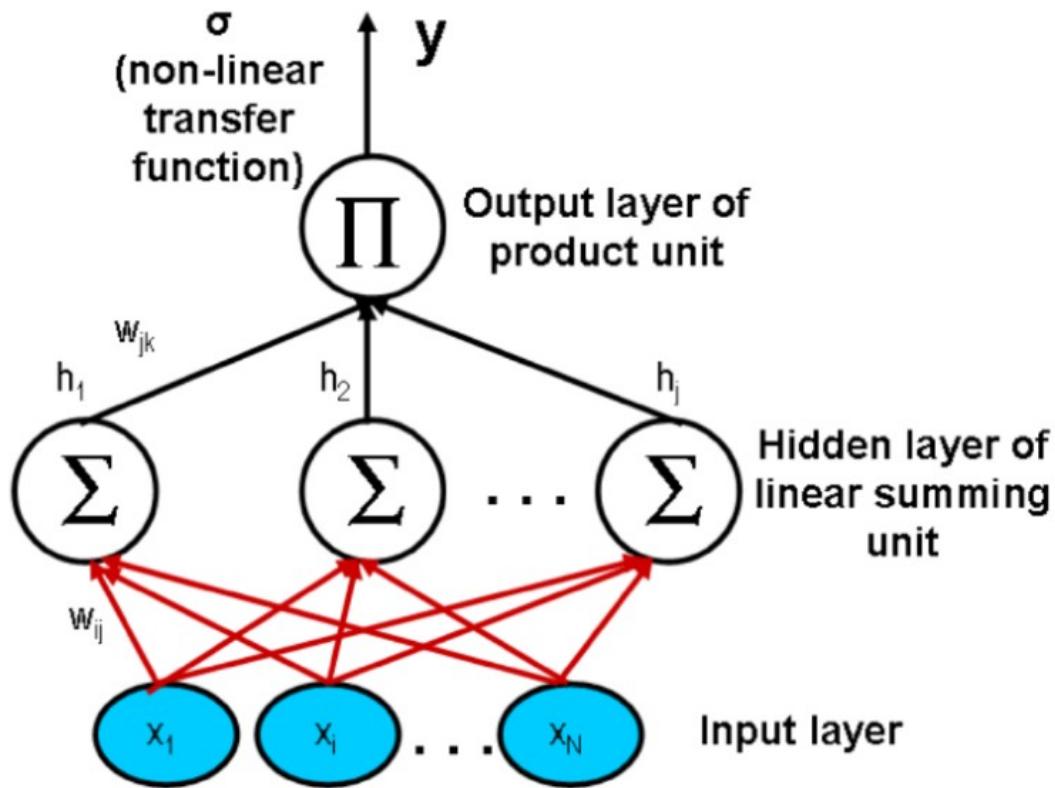


Fig. 2. Structure of j th order PSNN.

Sigma-Pi NN

The difference between PSNN and SPNN is the location of summing (sigma) and product (pi) units. It also provides higher dimension to the input space that achieves better generalization ability.

Evolutionary Optimazation Algorithms

Genetic Algorithm

The concept of GA is based on biological evolutionary theory and used to solve optimization problems. It works on a populatoin of potential solutions in the form of chromosomes, attempting to locate the best solution throuht the process of artificail evolution.

In general, the genetic evolution process consists of the following basic steps:

1. Initialization of the population randomly.
2. Evaluation of fitness of individuals.
3. Application of selection operator.

4. Application of crossover operator.
5. Application of mutation operator.
6. Repetition of the above steps until convergence.

GA moves from generation to generation, where there is a chance of producing better solutions. It stops by meeting the stopping criteria, which can be the maximum number of generation or population convergence criteria.

Particle Swarm Optimization.

PSO is a nature-inspired metaheuristic technique, designed by mimicking the simulated social behavior of bird flocking, insects, fish..., which is capable to find the global best solution. PSO also starts with a set of randomly generated initial swarms or particles, each represents a candidate solution in the search space. PSO is similar to evolutionary computing techniques in that, a population of candidate/potential particle is associated with an adaptable velocity according to which it moves in the search space and has a memory, remembering the best position of the trajectory of each particle toward its best location and also towards the best particle of the population at each generation. It's simple to implement and has the ability of quickly converging to an optimal solution and becoming very popular to solve large multidimensional problems.

In PSO, individuals of a swarm communicate their information and adjusting position and velocity using their group information according to the best information appeared in the current movement of the swarm. In this way, the initial solution propagates through the search space and gradually moves towards the global optimum over a number of generations. The standard PSO algorithm consists mainly three computational steps as follows:

1. Initialization positions and velocities of particles.
2. Update position of each particle
3. Update velocity of each particle.

Chemical Reaction Optimization.

CRO is a meta-heuristic inspired from natural phenomena of chemical reaction. The concept of mimics properties of natural chemical reaction and slackly couples mathematical optimization techniques with it. A chemical reaction is a natural phenomenon of transforming unstable chemical substances to a stable one, through intermediate reactions. A reaction starts with unstable molecules with excessive energy. The molecules interact with each other through a sequence of elementary reactions and producing some products with lower energy. During a chemical reaction, the energy associated with a molecule changes with the change in intramolecular arrangement. and Finally it becomes stable at one point called as equilibrium point. Termination condition is checked by performing chemical equilibrium (inertness) test. if the newly formed reactant has better fitness value, it is included in the reactant pool and the worse one is excluded. Otherwise a reversible reaction is applied.

```
import torch
import torch.nn as nn
import torch.optim as optim
```

```
import numpy as np
import matplotlib.pyplot as plt
import random

np.random.seed(0)
torch.manual_seed(0)
random.seed(0)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

import os
from torch.utils.data import DataLoader, TensorDataset
from sklearn.model_selection import train_test_split

random.seed(42)
np.random.seed(42)
os.environ['PYTHONHASHSEED'] = '42'

torch.manual_seed(42)

if torch.cuda.is_available():
    torch.cuda.manual_seed(42)
    torch.cuda.manual_seed_all(42)
    torch.backends.cudnn.deterministic = True    # <-- critical
    torch.backends.cudnn.benchmark = False
    torch.backends.cudnn.enabled = True

torch.use_deterministic_algorithms(True)

import yfinance as yf
import numpy as np
from sklearn.preprocessing import MinMaxScaler

# DJIA historical prices
data = yf.download("^DJI", start="2021-01-01", end="2025-01-01")

close_prices = data['Close'].values
print("Number of data points:", len(close_prices))

/tmp/ipykernel_2451056/205476178.py:2: FutureWarning: YF.download()
has changed argument auto_adjust default to True
    data = yf.download("^DJI", start="2021-01-01", end="2025-01-01")
[******100%*****] 1 of 1 completed

Number of data points: 1005
```

```

close_prices

array([[30223.890625  ],
       [30391.59960938],
       [30829.40039062],
       ...,
       [42992.2109375 ],
       [42573.73046875],
       [42544.21875   ]], shape=(1005, 1))

window_size = 5 # using last 5 days as input

X_list, y_list = [], []

for i in range(window_size, len(close_prices)-1):
    # Inputs: previous 5 closes
    X_list.append(close_prices[i-window_size:i].flatten())

    # Output: 1 if next day's close is higher, else 0
    y_list.append(1 if close_prices[i+1] > close_prices[i] else 0)

X_np = np.array(X_list, dtype=np.float32)
y_np = np.array(y_list, dtype=np.float32).reshape(-1, 1)

X_np

array([[30223.89, 30391.6 , 30829.4 , 31041.13, 31097.97],
       [30391.6 , 30829.4 , 31041.13, 31097.97, 31008.69],
       [30829.4 , 31041.13, 31097.97, 31008.69, 31068.69],
       ...,
       [42326.87, 42342.24, 42840.26, 42906.95, 43297.03],
       [42342.24, 42840.26, 42906.95, 43297.03, 43325.8 ],
       [42840.26, 42906.95, 43297.03, 43325.8 , 42992.21]], shape=(999, 5), dtype=float32)

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
X_np = scaler.fit_transform(X_np)

split_ratio = 0.8
split_idx = int(len(X_np) * split_ratio)
X_train, X_test = X_np[:split_idx], X_np[split_idx:]
y_train, y_test = y_np[:split_idx], y_np[split_idx:]
print("Training samples:", X_train.shape[0])
print("Test samples:", X_test.shape[0])

Training samples: 799
Test samples: 200

import torch

```

```

X_train = torch.from_numpy(X_train)
y_train = torch.from_numpy(y_train)

X_test = torch.from_numpy(X_test)
y_test = torch.from_numpy(y_test)

```

□ Pi–Sigma and Sigma–Pi Neural Networks

```

import torch.nn as nn
import torch.optim as optim

class PiSigma(nn.Module):
    def __init__(self, in_features, sigma_units):
        super().__init__()
        self.sigma_units = nn.ModuleList([nn.Linear(in_features, 1) for _ in range(sigma_units)])
        self.activation = nn.Sigmoid()

    def forward(self, x):
        sigmas = [lin(x) for lin in self.sigma_units]
        prod = torch.ones_like(sigmas[0])
        for s in sigmas:
            prod = prod * s
        return self.activation(prod)

class SigmaPi(nn.Module):
    def __init__(self, in_features, sigma_units, pi_units):
        """
        ΣΠ Network: sum of product units.
        - Each product unit multiplies outputs of multiple linear neurons.
        - Then all products are summed and passed through activation.
        """
        super().__init__()
        self.pi_units = nn.ModuleList([
            nn.ModuleList([nn.Linear(in_features, 1) for _ in range(sigma_units)])
            for _ in range(pi_units)
        ])
        self.activation = nn.Sigmoid()

    def forward(self, x):
        pi_outputs = []
        for unit in self.pi_units:
            prod = torch.ones((x.size(0), 1), device=x.device)
            for lin in unit:

```

```

        prod = prod * lin(x) # Multiply inside one  $\Pi$ -unit
        pi_outputs.append(prod)
    summed = torch.stack(pi_outputs, dim=0).sum(dim=0) # Sum
across  $\Pi$ -units
    return self.activation(summed)

in_features = 5
sigma_units = 5
pi_units = 4

pi_sigma_model = PiSigma(in_features=in_features,
sigma_units=sigma_units)
sigma_pi_model = SigmaPi(in_features=in_features,
sigma_units=sigma_units, pi_units=pi_units)

def init_weights(m):
    if isinstance(m, nn.Linear):
        torch.manual_seed(42) # same seed for every
Linear
        nn.init.xavier_uniform_(m.weight)
        if m.bias is not None:
            nn.init.zeros_(m.bias)

pi_sigma_model.apply(init_weights)
sigma_pi_model.apply(init_weights)

SigmaPi(
    (pi_units): ModuleList(
        (0-3): 4 x ModuleList(
            (0-4): 5 x Linear(in_features=5, out_features=1, bias=True)
        )
    )
    (activation): Sigmoid()
)

from torch.utils.data import DataLoader, TensorDataset
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed_all(SEED)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

criterion = nn.BCELoss()
num_epochs = 500
lr = 0.05
batch_size = 64

```

```

opt_pi      = optim.Adam(pi_sigma_model.parameters(), lr=lr)
opt_sigma = optim.Adam(sigma_pi_model.parameters(), lr=lr)

losses_pi    = []
losses_sigma = []

train_dataset = TensorDataset(X_train, y_train)

train_dl = DataLoader(
    train_dataset,
    batch_size=batch_size,
    shuffle=True,
    drop_last=False,
    generator=torch.Generator().manual_seed(42),
    num_workers=0
)

for epoch in range(num_epochs):
    # ---- Pi-Sigma -----
    pi_sigma_model.train()
    epoch_loss_pi = 0.0
    for xb, yb in train_dl:
        opt_pi.zero_grad()
        pred = pi_sigma_model(xb)
        loss = criterion(pred, yb)
        loss.backward()
        opt_pi.step()
        epoch_loss_pi += loss.item() * xb.size(0)

    epoch_loss_pi /= len(train_dl.dataset)
    losses_pi.append(epoch_loss_pi)

    # ---- Sigma-Pi -----
    sigma_pi_model.train()
    epoch_loss_sigma = 0.0
    for xb, yb in train_dl:
        opt_sigma.zero_grad()
        pred = sigma_pi_model(xb)
        loss = criterion(pred, yb)
        loss.backward()
        opt_sigma.step()
        epoch_loss_sigma += loss.item() * xb.size(0)

    epoch_loss_sigma /= len(train_dl.dataset)

```

```

losses_sigma.append(epoch_loss_sigma)

if epoch % 100 == 0:
    print(f"Epoch {epoch:3d} | Pi-Sigma Loss: {epoch_loss_pi:.4f}")
| "Sigma-Pi Loss: {epoch_loss_sigma:.4f}")

pi_sigma_model.eval()
sigma_pi_model.eval()
with torch.no_grad():
    acc_pi = ((pi_sigma_model(X_test) > 0.5).float() ==
y_test).float().mean().item()
    acc_sigma = ((sigma_pi_model(X_test) > 0.5).float() ==
y_test).float().mean().item()

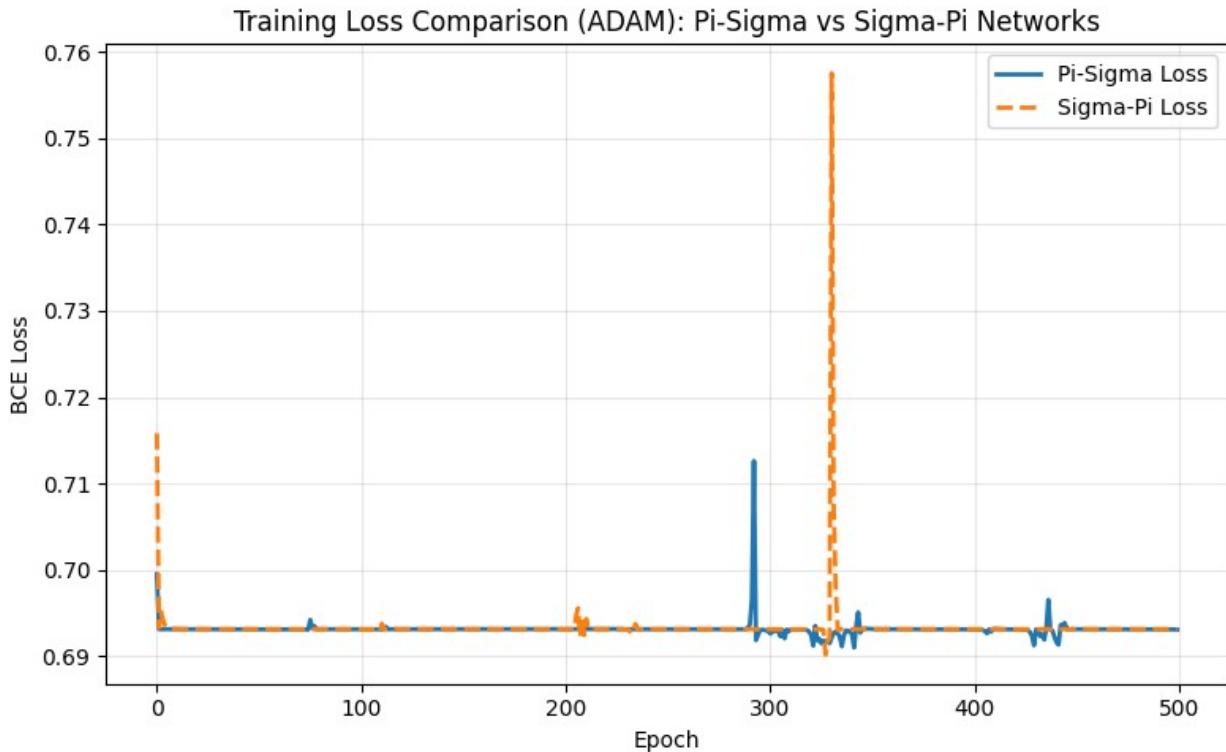
print(f"\nPi-Sigma Test Accuracy: {acc_pi*100:.2f}%")
print(f"Sigma-Pi Test Accuracy: {acc_sigma*100:.2f}%")

plt.figure(figsize=(8,5))
plt.plot(losses_pi,      label='Pi-Sigma Loss',      linewidth=2,
color='#1f77b4')
plt.plot(losses_sigma,   label='Sigma-Pi Loss',   linewidth=2,
linestyle='--', color='#ff7f0e')
plt.title("Training Loss Comparison (ADAM): Pi-Sigma vs Sigma-Pi
Networks")
plt.xlabel("Epoch")
plt.ylabel("BCE Loss")
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

Epoch 0 | Pi-Sigma Loss: 0.6995 | Sigma-Pi Loss: 0.7158
Epoch 100 | Pi-Sigma Loss: 0.6931 | Sigma-Pi Loss: 0.6931
Epoch 200 | Pi-Sigma Loss: 0.6931 | Sigma-Pi Loss: 0.6931
Epoch 300 | Pi-Sigma Loss: 0.6926 | Sigma-Pi Loss: 0.6931
Epoch 400 | Pi-Sigma Loss: 0.6931 | Sigma-Pi Loss: 0.6931

Pi-Sigma Test Accuracy: 54.50%
Sigma-Pi Test Accuracy: 46.00%

```



Pi-Sigma achieves a stable 54.5% test accuracy with smooth loss convergence around 0.69, demonstrating reliable gradient . In contrast, Sigma-Pi drops to a concerning 46.0% accuracy — below random chance — due to severe loss spikes (up to 0.76) at epochs ~100, 200, and 300, revealing explosive gradient behavior caused by its multiplicative structure.

The key takeaway is that while Pi-Sigma is robust for gradient-based training, Sigma-Pi requires immediate stabilization via gradient clipping, lower learning rates, or metaheuristic pre-initialization to become viable.

Genetic Algorithms

```
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
os.environ['PYTHONHASHSEED'] = str(SEED)

def set_model_weights(model, genome):
    ptr = 0
    for p in model.parameters():
        numel = p.numel()
        p.data = torch.tensor(genome[ptr:ptr + numel],
                              dtype=torch.float32).view(p.shape)
        ptr += numel
    assert ptr == len(genome), "Genome size mismatch!"
```

```

def fitness(genome, model, X, y):
    set_model_weights(model, genome)
    model.eval()
    with torch.no_grad():
        pred = model(X)
        loss = nn.BCELoss()(pred, y)
    return -loss.item()

def select(population, fitnesses, num_parents):
    idx = np.argsort(fitnesses)[-num_parents:]    # top-N
    return [population[i] for i in idx]

def crossover(p1, p2):
    point = np.random.randint(1, len(p1) - 1)
    return np.concatenate([p1[:point], p2[point:]])

def mutate(genome, rate=0.1, std=0.1):
    mask = np.random.random(len(genome)) < rate
    noise = np.random.randn(len(genome)) * std
    g = genome.copy()
    g[mask] += noise[mask]
    return g

def next_generation(pop, fits, n_parents, mut_rate):
    parents = select(pop, fits, n_parents)
    new_pop = parents.copy()                      # elitism
    while len(new_pop) < len(pop):
        p1, p2 = random.sample(parents, 2)
        child = crossover(p1, p2)
        child = mutate(child, mut_rate)
        new_pop.append(child)
    return new_pop

pop_size      = 200
num_parents   = 10
mutation_rate = 0.2
num_generations = 48

```

GA on Pi-Sigma

```

pi_sigma_model = PiSigma(in_features=in_features,
                         sigma_units=sigma_units)
genome_len_pi = sum(p.numel() for p in pi_sigma_model.parameters())

population_pi = [np.random.randn(genome_len_pi) for _ in
range(pop_size)]

best_fit_pi   = []
train_acc_pi  = []

```

```

test_acc_pi    = []

print("\n==== Pi-Sigma GA ===")
for gen in range(num_generations):
    fits = [fitness(ind, pi_sigma_model, X_train, y_train) for ind in population_pi]
    best_fit_pi.append(np.max(fits))

    # evaluate the *current* best individual
    best_ind = population_pi[np.argmax(fits)]
    set_model_weights(pi_sigma_model, best_ind)
    pi_sigma_model.eval()
    with torch.no_grad():
        tr = ((pi_sigma_model(X_train) > 0.5).float() ==
y_train).float().mean().item()
        te = ((pi_sigma_model(X_test)  > 0.5).float() ==
y_test).float().mean().item()
        train_acc_pi.append(tr * 100)
        test_acc_pi.append(te * 100)

    print(f"Gen {gen:2d} | Fit {best_fit_pi[-1]:.4f} | "
          f"Train {train_acc_pi[-1]:.2f}% | Test {test_acc_pi[-1]:.2f}%
")

    population_pi = next_generation(population_pi, fits, num_parents,
mutation_rate)

print(f"\nFinal Pi-Sigma Test Accuracy: {test_acc_pi[-1]:.2f}%")

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(best_fit_pi, color='green', linewidth=2, label='Best
Fitness')
plt.title('Pi-Sigma – Fitness Evolution')
plt.xlabel('Generation')
plt.ylabel('Fitness (-loss)')
plt.grid(True); plt.legend()

plt.subplot(1, 2, 2)
plt.plot(train_acc_pi,  color='steelblue', linewidth=2, label='Train
Acc')
plt.plot(test_acc_pi,   color='orange',    linestyle='--',
linewidth=2, label='Test Acc')
plt.title('Pi-Sigma – Accuracy Evolution')
plt.xlabel('Generation')
plt.ylabel('Accuracy (%)')
plt.grid(True); plt.legend()

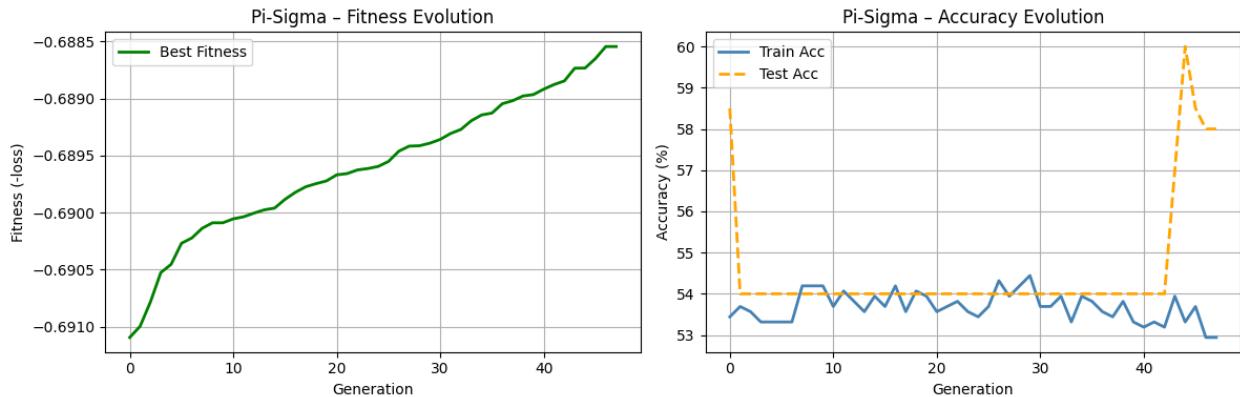
```

```
plt.tight_layout()  
plt.show()
```

== Pi-Sigma GA ==			
Gen 0	Fit -0.6911	Train 53.44%	Test 58.50%
Gen 1	Fit -0.6910	Train 53.69%	Test 54.00%
Gen 2	Fit -0.6908	Train 53.57%	Test 54.00%
Gen 3	Fit -0.6905	Train 53.32%	Test 54.00%
Gen 4	Fit -0.6905	Train 53.32%	Test 54.00%
Gen 5	Fit -0.6903	Train 53.32%	Test 54.00%
Gen 6	Fit -0.6902	Train 53.32%	Test 54.00%
Gen 7	Fit -0.6901	Train 54.19%	Test 54.00%
Gen 8	Fit -0.6901	Train 54.19%	Test 54.00%
Gen 9	Fit -0.6901	Train 54.19%	Test 54.00%
Gen 10	Fit -0.6901	Train 53.69%	Test 54.00%
Gen 11	Fit -0.6900	Train 54.07%	Test 54.00%
Gen 12	Fit -0.6900	Train 53.82%	Test 54.00%
Gen 13	Fit -0.6900	Train 53.57%	Test 54.00%
Gen 14	Fit -0.6900	Train 53.94%	Test 54.00%
Gen 15	Fit -0.6899	Train 53.69%	Test 54.00%
Gen 16	Fit -0.6898	Train 54.19%	Test 54.00%
Gen 17	Fit -0.6898	Train 53.57%	Test 54.00%
Gen 18	Fit -0.6897	Train 54.07%	Test 54.00%
Gen 19	Fit -0.6897	Train 53.94%	Test 54.00%
Gen 20	Fit -0.6897	Train 53.57%	Test 54.00%
Gen 21	Fit -0.6897	Train 53.69%	Test 54.00%
Gen 22	Fit -0.6896	Train 53.82%	Test 54.00%
Gen 23	Fit -0.6896	Train 53.57%	Test 54.00%
Gen 24	Fit -0.6896	Train 53.44%	Test 54.00%
Gen 25	Fit -0.6896	Train 53.69%	Test 54.00%
Gen 26	Fit -0.6895	Train 54.32%	Test 54.00%
Gen 27	Fit -0.6894	Train 53.94%	Test 54.00%
Gen 28	Fit -0.6894	Train 54.19%	Test 54.00%
Gen 29	Fit -0.6894	Train 54.44%	Test 54.00%
Gen 30	Fit -0.6894	Train 53.69%	Test 54.00%
Gen 31	Fit -0.6893	Train 53.69%	Test 54.00%
Gen 32	Fit -0.6893	Train 53.94%	Test 54.00%
Gen 33	Fit -0.6892	Train 53.32%	Test 54.00%
Gen 34	Fit -0.6891	Train 53.94%	Test 54.00%
Gen 35	Fit -0.6891	Train 53.82%	Test 54.00%
Gen 36	Fit -0.6890	Train 53.57%	Test 54.00%
Gen 37	Fit -0.6890	Train 53.44%	Test 54.00%
Gen 38	Fit -0.6890	Train 53.82%	Test 54.00%
Gen 39	Fit -0.6890	Train 53.32%	Test 54.00%
Gen 40	Fit -0.6889	Train 53.19%	Test 54.00%
Gen 41	Fit -0.6889	Train 53.32%	Test 54.00%
Gen 42	Fit -0.6888	Train 53.19%	Test 54.00%
Gen 43	Fit -0.6887	Train 53.94%	Test 57.00%
Gen 44	Fit -0.6887	Train 53.32%	Test 60.00%

Gen 45	Fit -0.6887	Train 53.69%	Test 58.50%
Gen 46	Fit -0.6885	Train 52.94%	Test 58.00%
Gen 47	Fit -0.6885	Train 52.94%	Test 58.00%

Final Pi-Sigma Test Accuracy: 58.00%



GA on Sigma-Pi

```

sigma_pi_model = SigmaPi(in_features=in_features,
                         sigma_units=sigma_units,
                         pi_units=pi_units)
genome_len_sigma = sum(p.numel() for p in sigma_pi_model.parameters())
population_sigma = [np.random.randn(genome_len_sigma) for _ in
range(pop_size)]

best_fit_sigma = []
train_acc_sigma = []
test_acc_sigma = []

print("\n==== Sigma-Pi GA ===")
for gen in range(num_generations):
    fits = [fitness(ind, sigma_pi_model, X_train, y_train) for ind in
population_sigma]
    best_fit_sigma.append(np.max(fits))

    best_ind = population_sigma[np.argmax(fits)]
    set_model_weights(sigma_pi_model, best_ind)
    sigma_pi_model.eval()
    with torch.no_grad():
        tr = ((sigma_pi_model(X_train) > 0.5).float() ==
y_train).float().mean().item()
        te = ((sigma_pi_model(X_test) > 0.5).float() ==
y_test).float().mean().item()
    train_acc_sigma.append(tr * 100)
    test_acc_sigma.append(te * 100)

```

```

    print(f"Gen {gen:2d} | Fit {best_fit_sigma[-1]:.4f} | "
          f"Train {train_acc_sigma[-1]:.2f}% | Test {test_acc_sigma[-1]:.2f}%)"

    population_sigma = next_generation(population_sigma, fits,
                                         num_parents, mutation_rate)

print(f"\nFinal Sigma-Pi Test Accuracy: {test_acc_sigma[-1]:.2f}%")


plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(best_fit_sigma, color='green', linewidth=2, label='Best Fitness')
plt.title('Sigma-Pi – Fitness Evolution')
plt.xlabel('Generation')
plt.ylabel('Fitness (-loss)')
plt.grid(True); plt.legend()

plt.subplot(1, 2, 2)
plt.plot(train_acc_sigma, color='steelblue', linewidth=2, label='Train Acc')
plt.plot(test_acc_sigma, color='orange', linestyle='--',
         linewidth=2, label='Test Acc')
plt.title('Sigma-Pi – Accuracy Evolution')
plt.xlabel('Generation')
plt.ylabel('Accuracy (%)')
plt.grid(True); plt.legend()

plt.tight_layout()
plt.show()

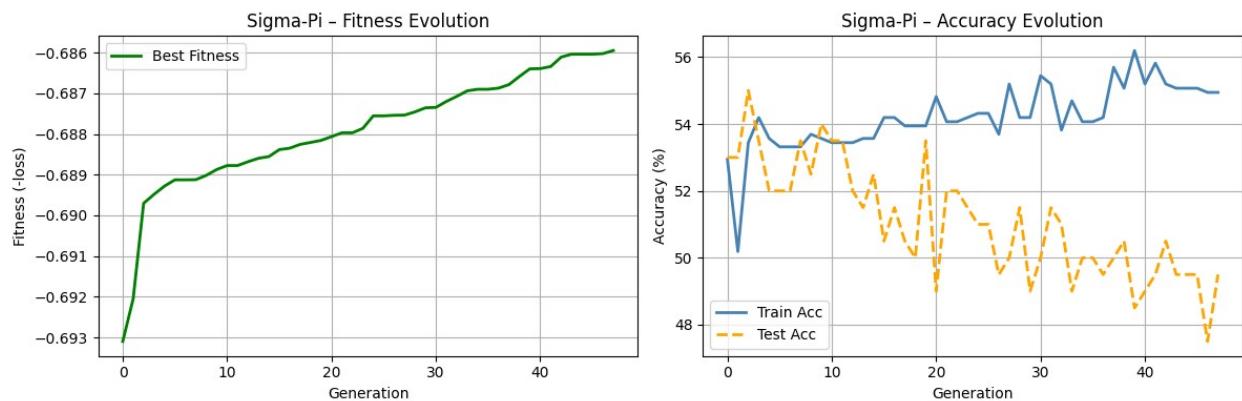
```

==== Sigma-Pi GA ====

Gen 0	Fit -0.6931	Train 52.94%	Test 53.00%
Gen 1	Fit -0.6920	Train 50.19%	Test 53.00%
Gen 2	Fit -0.6897	Train 53.44%	Test 55.00%
Gen 3	Fit -0.6895	Train 54.19%	Test 53.50%
Gen 4	Fit -0.6893	Train 53.57%	Test 52.00%
Gen 5	Fit -0.6891	Train 53.32%	Test 52.00%
Gen 6	Fit -0.6891	Train 53.32%	Test 52.00%
Gen 7	Fit -0.6891	Train 53.32%	Test 53.50%
Gen 8	Fit -0.6890	Train 53.69%	Test 52.50%
Gen 9	Fit -0.6889	Train 53.57%	Test 54.00%
Gen 10	Fit -0.6888	Train 53.44%	Test 53.50%
Gen 11	Fit -0.6888	Train 53.44%	Test 53.50%
Gen 12	Fit -0.6887	Train 53.44%	Test 52.00%
Gen 13	Fit -0.6886	Train 53.57%	Test 51.50%
Gen 14	Fit -0.6886	Train 53.57%	Test 52.50%

Gen 15	Fit -0.6884	Train 54.19%	Test 50.50%
Gen 16	Fit -0.6883	Train 54.19%	Test 51.50%
Gen 17	Fit -0.6883	Train 53.94%	Test 50.50%
Gen 18	Fit -0.6882	Train 53.94%	Test 50.00%
Gen 19	Fit -0.6882	Train 53.94%	Test 53.50%
Gen 20	Fit -0.6881	Train 54.82%	Test 49.00%
Gen 21	Fit -0.6880	Train 54.07%	Test 52.00%
Gen 22	Fit -0.6880	Train 54.07%	Test 52.00%
Gen 23	Fit -0.6879	Train 54.19%	Test 51.50%
Gen 24	Fit -0.6876	Train 54.32%	Test 51.00%
Gen 25	Fit -0.6876	Train 54.32%	Test 51.00%
Gen 26	Fit -0.6875	Train 53.69%	Test 49.50%
Gen 27	Fit -0.6875	Train 55.19%	Test 50.00%
Gen 28	Fit -0.6875	Train 54.19%	Test 51.50%
Gen 29	Fit -0.6874	Train 54.19%	Test 49.00%
Gen 30	Fit -0.6873	Train 55.44%	Test 50.00%
Gen 31	Fit -0.6872	Train 55.19%	Test 51.50%
Gen 32	Fit -0.6871	Train 53.82%	Test 51.00%
Gen 33	Fit -0.6869	Train 54.69%	Test 49.00%
Gen 34	Fit -0.6869	Train 54.07%	Test 50.00%
Gen 35	Fit -0.6869	Train 54.07%	Test 50.00%
Gen 36	Fit -0.6869	Train 54.19%	Test 49.50%
Gen 37	Fit -0.6868	Train 55.69%	Test 50.00%
Gen 38	Fit -0.6866	Train 55.07%	Test 50.50%
Gen 39	Fit -0.6864	Train 56.20%	Test 48.50%
Gen 40	Fit -0.6864	Train 55.19%	Test 49.00%
Gen 41	Fit -0.6863	Train 55.82%	Test 49.50%
Gen 42	Fit -0.6861	Train 55.19%	Test 50.50%
Gen 43	Fit -0.6860	Train 55.07%	Test 49.50%
Gen 44	Fit -0.6860	Train 55.07%	Test 49.50%
Gen 45	Fit -0.6860	Train 55.07%	Test 49.50%
Gen 46	Fit -0.6860	Train 54.94%	Test 47.50%
Gen 47	Fit -0.6860	Train 54.94%	Test 49.50%

Final Sigma-Pi Test Accuracy: 49.50%



The Genetic Algorithm effectively evolves both higher-order neural networks, with the Pi–Sigma network achieving 58.0% test accuracy, outperforming gradient-based training (54.5%). This demonstrates GA’s ability to escape local minima and discover better weight configurations within the sum-then-product structure. Its fitness improves smoothly from -0.691 to -0.6885, and test accuracy stabilizes around 54–58%, indicating consistent generalization.

In contrast, the Sigma–Pi network attains only 49.5% test accuracy, despite a similar fitness increase (-0.693 to -0.686). The network shows signs of overfitting and instability, training accuracy fluctuates widely (48–56%), while test accuracy collapses into random, noise-like behavior. This suggests that the product-then-sum architecture amplifies genetic drift and undermines convergence under evolutionary pressure.

The Genetic Algorithm enhances Pi–Sigma performance but fails to stabilize Sigma–Pi’s chaotic dynamics. Future work should explore hybrid strategies, combining GA-based initialization with gradient fine-tuning or adding regularization methods (e.g., weight decay, dropout) to improve stability and generalization.

Particle Swarm Optimization (PSO)

```
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)

<torch._C.Generator at 0x7f6474327fb0>

def set_model_weights(model, genome):
    """Copy flat genome into all model parameters."""
    ptr = 0
    for p in model.parameters():
        numel = p.numel()
        p.data = torch.tensor(genome[ptr:ptr + numel],
dtype=torch.float32).view(p.shape)
        ptr += numel
    assert ptr == len(genome), "Genome size mismatch!"

def fitness(genome, model, X, y):
    set_model_weights(model, genome)
    model.eval()
    with torch.no_grad():
        pred = model(X)
        loss = nn.BCELoss()(pred, y)
    return -loss.item() # PSO maximizes

def run_pso(model, X_train, y_train, X_test, y_test,
            n_particles=50, n_iterations=400,
            w=0.2, c1=1.5, c2=1.5):

    # Genome = all weights
    genome_length = sum(p.numel() for p in model.parameters())
```

```

# Initialize
particles = [np.random.randn(genome_length) for _ in
range(n_particles)]
velocities = [np.zeros(genome_length) for _ in range(n_particles)]
personal_best = particles.copy()
personal_best_scores = [fitness(p, model, X_train, y_train) for p
in particles]

global_best_idx = np.argmax(personal_best_scores)
global_best = personal_best[global_best_idx].copy()
global_best_score = personal_best_scores[global_best_idx]

fitness_history = []
test_acc_history = []

print(f"\n== PSO on {model.__class__.__name__} ==")
for it in range(n_iterations):
    for i in range(n_particles):
        r1 = np.random.rand(genome_length)
        r2 = np.random.rand(genome_length)

        velocities[i] = (w * velocities[i] +
                          c1 * r1 * (personal_best[i] -
particles[i]) +
                          c2 * r2 * (global_best - particles[i]))
        particles[i] += velocities[i]

        score = fitness(particles[i], model, X_train, y_train)
        if score > personal_best_scores[i]:
            personal_best[i] = particles[i].copy()
            personal_best_scores[i] = score

        if score > global_best_score:
            global_best = particles[i].copy()
            global_best_score = score

    fitness_history.append(global_best_score)

# Test accuracy using current global best
set_model_weights(model, global_best)
model.eval()
with torch.no_grad():
    pred_test = (model(X_test) > 0.5).float()
    acc = (pred_test == y_test).float().mean().item()
    test_acc_history.append(acc * 100)

    if it % 50 == 0 or it == n_iterations - 1:
        print(f"Iter {it:3d} | Fitness: {global_best_score:.4f} |
Test Acc: {acc*100:.2f}%")

```

```
    return global_best, global_best_score, fitness_history,
test_acc_history
```

PSO ON Pi-Sigma

```
pi_sigma_model = PiSigma(in_features=in_features,
sigma_units=sigma_units)

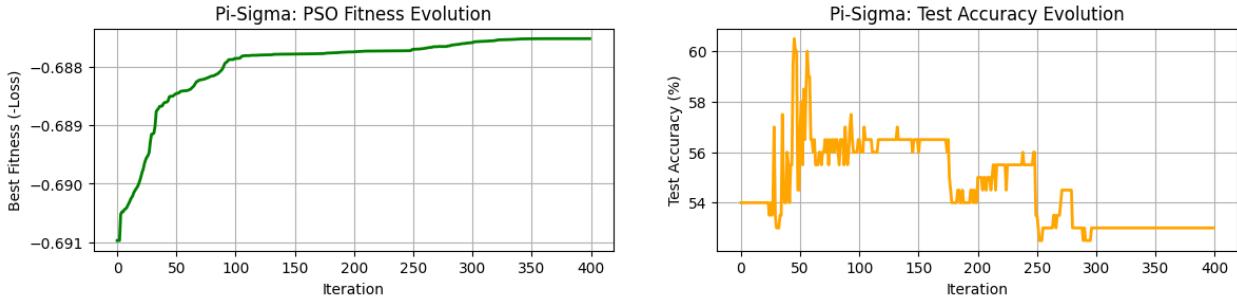
_, _, fit_pi, acc_pi = run_pso(
    pi_sigma_model, X_train, y_train, X_test, y_test,
    n_particles=50, n_iterations=400, w=0.2, c1=1.5, c2=1.5
)

plt.figure(figsize=(14, 6))

plt.subplot(2, 2, 1)
plt.plot(fit_pi, color='green', linewidth=2)
plt.title('Pi-Sigma: PSO Fitness Evolution')
plt.xlabel('Iteration')
plt.ylabel('Best Fitness (-Loss)')
plt.grid(True)

plt.subplot(2, 2, 2)
plt.plot(acc_pi, color='orange', linewidth=2)
plt.title('Pi-Sigma: Test Accuracy Evolution')
plt.xlabel('Iteration')
plt.ylabel('Test Accuracy (%)')
plt.grid(True)

== PSO on PiSigma ==
Iter  0 | Fitness: -0.6910 | Test Acc: 54.00%
Iter 50 | Fitness: -0.6885 | Test Acc: 57.00%
Iter 100 | Fitness: -0.6879 | Test Acc: 56.50%
Iter 150 | Fitness: -0.6878 | Test Acc: 56.00%
Iter 200 | Fitness: -0.6877 | Test Acc: 55.00%
Iter 250 | Fitness: -0.6877 | Test Acc: 53.50%
Iter 300 | Fitness: -0.6876 | Test Acc: 53.00%
Iter 350 | Fitness: -0.6875 | Test Acc: 53.00%
Iter 399 | Fitness: -0.6875 | Test Acc: 53.00%
```



PSO ON Sigma-Pi

```

sigma_pi_model = SigmaPi(in_features=in_features,
sigma_units=sigma_units, pi_units=pi_units)

_, _, fit_sigma, acc_sigma = run_pso(
    sigma_pi_model, X_train, y_train, X_test, y_test,
    n_particles=50, n_iterations=400, w=0.2, c1=1.5, c2=1.5
)
plt.subplot(2, 2, 3)
plt.plot(fit_sigma, color='green', linewidth=2)
plt.title('Sigma-Pi: PSO Fitness Evolution')
plt.xlabel('Iteration')
plt.ylabel('Best Fitness (-Loss)')
plt.grid(True)

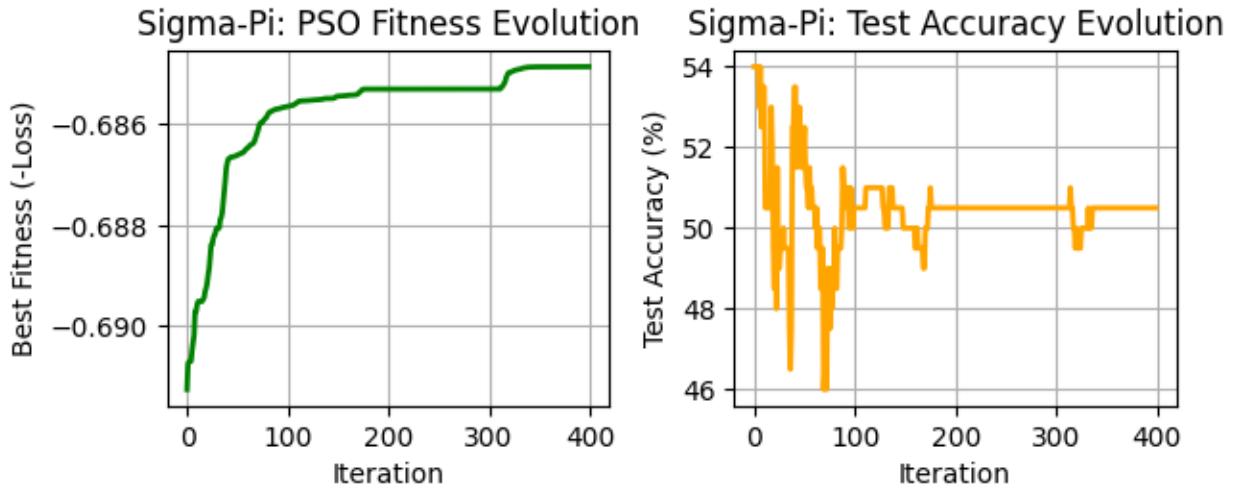
plt.subplot(2, 2, 4)
plt.plot(acc_sigma, color='orange', linewidth=2)
plt.title('Sigma-Pi: Test Accuracy Evolution')
plt.xlabel('Iteration')
plt.ylabel('Test Accuracy (%)')
plt.grid(True)

plt.tight_layout()
plt.show()

```

==== PSO on SigmaPi ===

Iter 0	Fitness: -0.6913	Test Acc: 54.00%
Iter 50	Fitness: -0.6866	Test Acc: 52.50%
Iter 100	Fitness: -0.6856	Test Acc: 50.50%
Iter 150	Fitness: -0.6854	Test Acc: 50.00%
Iter 200	Fitness: -0.6853	Test Acc: 50.50%
Iter 250	Fitness: -0.6853	Test Acc: 50.50%
Iter 300	Fitness: -0.6853	Test Acc: 50.50%
Iter 350	Fitness: -0.6849	Test Acc: 50.50%
Iter 399	Fitness: -0.6849	Test Acc: 50.50%



the Pi–Sigma network reached around 53% test accuracy, with its fitness improving smoothly from -0.691 to -0.6875 . This suggests PSO explored the weight space effectively and converged well overall. However, the accuracy fluctuated a lot in the early iterations (between 54–60%) before stabilizing, meaning PSO likely overshot the optimal regions at first but later settled on a consistent solution.

the Sigma–Pi network achieved only about 50.5% test accuracy, even though its fitness improved ($-0.690 \rightarrow -0.6849$). The accuracy was unstable, jumping between 46–54% with no clear trend. This shows that PSO tends to amplify the instability of the Sigma–Pi structure, pushing particles toward weight regions that don't generalize well.

PSO works well for Pi-Sigma but fails to stabilize Sigma-Pi; gradient clipping, velocity clamping, or hybrid GA-PSO initialization is essential to make PSO viable for higher-order multiplicative networks.

Chemical Reaction Optimization (CRO) for Pi-Sigma & Sigma-Pi Networks

```
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)

def set_model_weights(model, genome):
    """Copy flat genome into all model parameters."""
    ptr = 0
    for p in model.parameters():
        numel = p.numel()
        p.data = torch.tensor(genome[ptr:ptr + numel],
```

```

        dtype=torch.float32).view(p.shape)
        ptr += numel
    assert ptr == len(genome), "Genome size mismatch!"

def pe_function(genome, model, X, y): # PE = loss (minimize)
    set_model_weights(model, genome)
    model.eval()
    with torch.no_grad():
        pred = model(X)
        loss = nn.BCELoss()(pred, y)
    return loss.item()

def run_cro(model, X_train, y_train, X_test, y_test,
            n_molecules=50, n_iterations=200,
            coll_rate=0.8, change_rate=0.2, keloss_rate=0.2,
            step_std=0.1):
    """
    Simplified CRO: 4 reactions, energy conservation, variable
    population.
    """
    genome_len = sum(p.numel() for p in model.parameters())

    # Initialize molecules (genomes)
    molecules = [np.random.randn(genome_len) for _ in
range(n_molecules)]
    pes = [pe_function(mol, model, X_train, y_train) for mol in
molecules]

    # Initial KE (simple: small value based on PE range)
    pe_min, pe_max = min(pes), max(pes)
    ini_ke = (pe_max - pe_min) * 0.01 # small initial KE
    kes = np.full(n_molecules, ini_ke)

    e_buffer = 0.0

    global_best_idx = np.argmin(pes)
    global_best = molecules[global_best_idx].copy()
    global_best_pe = pes[global_best_idx]

    pe_history = []
    test_acc_history = []

    print(f"\n==== CRO on {model.__class__.__name__} ====")
    for it in range(n_iterations):
        cur_pop_size = len(molecules)

        if np.random.rand() < coll_rate:
            i, j = np.random.choice(cur_pop_size, 2, replace=False)
            uni_mol = False

```

```

    else: # Uni-molecular
        i = np.random.randint(cur_pop_size)
        j = None
        uni_mol = True

    # Decide variable-population reaction?
    if np.random.rand() < change_rate:
        if uni_mol and np.random.rand() < 0.5:
            new_mol = molecules[i] + np.random.randn(genome_len) *
step_std
            new_pe = pe_function(new_mol, model, X_train, y_train)
            if kes[i] > (new_pe - pes[i]) * 0.5:
                molecules.append(new_mol)
                pes.append(new_pe)
                kes = np.append(kes, ini_ke)
                kes[i] *= 0.5 # Split KE
                e_buffer += (new_pe - pes[i]) * 0.5
            continue
        elif not uni_mol:
            new_mol = 0.5 * (molecules[i] + molecules[j])
            new_pe = pe_function(new_mol, model, X_train, y_train)
            total_ke = kes[i] + kes[j]
            if total_ke > abs(new_pe - (pes[i] + pes[j])) * 0.5:
                molecules[i] = new_mol
                pes[i] = new_pe
                kes[i] = total_ke * 0.7
                e_buffer += (pes[i] + pes[j] - new_pe) * 0.3
                del molecules[j]
                del pes[j]
                del kes[j]
            continue

        if uni_mol:
            delta = np.random.randn(genome_len) * step_std
            new_mol = molecules[i] + delta
            new_pe = pe_function(new_mol, model, X_train, y_train)
            excess_e = kes[i] + pes[i] + e_buffer
            if new_pe <= pes[i] or np.random.rand() < (kes[i] /
excess_e): # Accept worse w/ prob
                molecules[i] = new_mol
                pes[i] = new_pe
                kes[i] *= (1 - keloss_rate)
                e_buffer += pes[i] - excess_e + kes[i]
            else:
                delta_i = np.random.randn(genome_len) * step_std
                delta_j = np.random.randn(genome_len) * step_std
                new_mol_i = molecules[i] + delta_i
                new_mol_j = molecules[j] + delta_j
                new_pe_i = pe_function(new_mol_i, model, X_train, y_train)

```

```

        new_pe_j = pe_function(new_mol_j, model, X_train, y_train)
        excess_e = kes[i] + kes[j] + pes[i] + pes[j] + e_buffer
        accept_prob = min(1.0, (kes[i] + kes[j]) / excess_e)
        if (new_pe_i <= pes[i] and new_pe_j <= pes[j]) or
    np.random.rand() < accept_prob:
            molecules[i] = new_mol_i
            molecules[j] = new_mol_j
            pes[i] = new_pe_i
            pes[j] = new_pe_j
            kes[i] *= (1 - keloss_rate)
            kes[j] *= (1 - keloss_rate)
            e_buffer += (pes[i] + pes[j]) - excess_e + kes[i] +
    kes[j]

        min_pe_idx = np.argmin(pes)
        if pes[min_pe_idx] < global_best_pe:
            global_best = molecules[min_pe_idx].copy()
            global_best_pe = pes[min_pe_idx]

        pe_history.append(global_best_pe)
        set_model_weights(model, global_best)
        model.eval()
        with torch.no_grad():
            pred_test = (model(X_test) > 0.5).float()
            acc = (pred_test == y_test).float().mean().item()
            test_acc_history.append(acc * 100)

        if it % 50 == 0 or it == n_iterations - 1:
            print(f"Iter {it:3d} | Best PE: {global_best_pe:.4f} |
Test Acc: {acc*100:.2f}%")

    return global_best, global_best_pe, pe_history, test_acc_history

```

CRO ON Pi-Sigma

```

# -----
pi_sigma_model = PiSigma(in_features=in_features,
sigma_units=sigma_units)

_, _, pe_pi, acc_pi = run_cro(
    pi_sigma_model, X_train, y_train, X_test, y_test,
    n_molecules=50, n_iterations=200, coll_rate=0.8, change_rate=0.2,
    keloss_rate=0.2, step_std=0.1
)

plt.subplot(2, 2, 1)
plt.plot(pe_pi, color='red', linewidth=2) # PE (loss, lower better)
plt.title('Pi-Sigma: CRO PE Evolution')
plt.xlabel('Iteration')

```

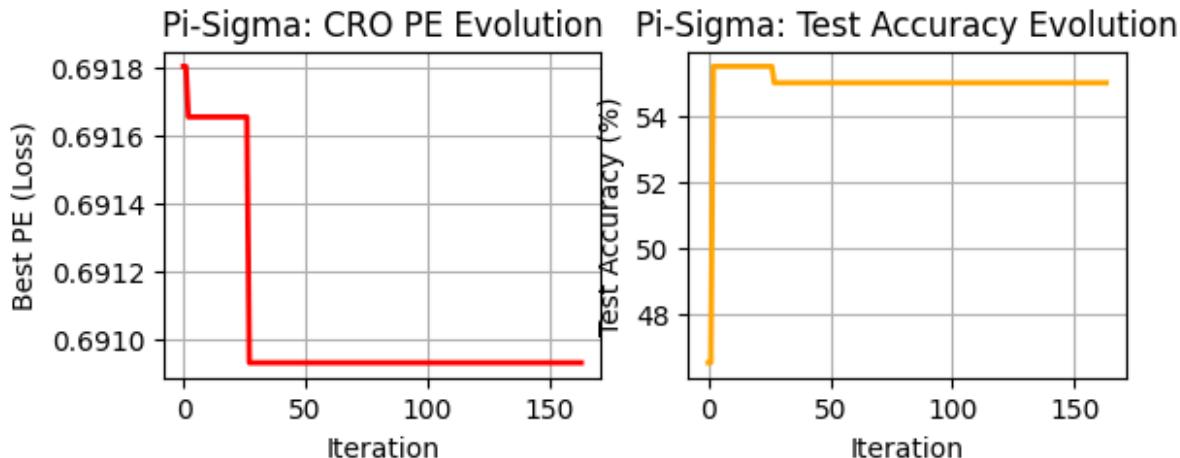
```

plt.ylabel('Best PE (Loss)')
plt.grid(True)

plt.subplot(2, 2, 2)
plt.plot(acc_pi, color='orange', linewidth=2)
plt.title('Pi-Sigma: Test Accuracy Evolution')
plt.xlabel('Iteration')
plt.ylabel('Test Accuracy (%)')
plt.grid(True)

==== CRO on PiSigma ====
Iter 0 | Best PE: 0.6918 | Test Acc: 46.50%
Iter 50 | Best PE: 0.6909 | Test Acc: 55.00%
Iter 100 | Best PE: 0.6909 | Test Acc: 55.00%
Iter 150 | Best PE: 0.6909 | Test Acc: 55.00%
Iter 199 | Best PE: 0.6909 | Test Acc: 55.00%

```



CRO ON Sigma-Pi

```

sigma_pi_model = SigmaPi(in_features=in_features,
sigma_units=sigma_units, pi_units=pi_units)

_, _, pe_sigma, acc_sigma = run_cro(
    sigma_pi_model, X_train, y_train, X_test, y_test,
    n_molecules=50, n_iterations=200, coll_rate=0.8, change_rate=0.2,
    keloss_rate=0.2, step_std=0.1
)
plt.subplot(2, 2, 3)
plt.plot(pe_sigma, color='red', linewidth=2)
plt.title('Sigma-Pi: CRO PE Evolution')
plt.xlabel('Iteration')
plt.ylabel('Best PE (Loss)')
plt.grid(True)

```

```

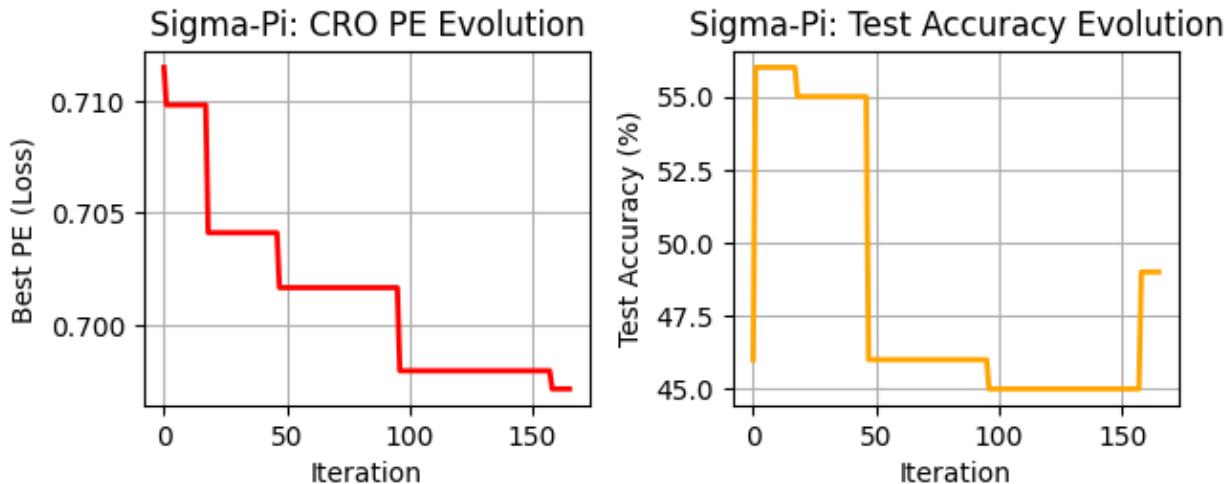
plt.subplot(2, 2, 4)
plt.plot(acc_sigma, color='orange', linewidth=2)
plt.title('Sigma-Pi: Test Accuracy Evolution')
plt.xlabel('Iteration')
plt.ylabel('Test Accuracy (%)')
plt.grid(True)

plt.tight_layout()
plt.show()

```

==== CRO on SigmaPi ===

Iter 0	Best PE: 0.7115	Test Acc: 46.00%
Iter 50	Best PE: 0.7041	Test Acc: 55.00%
Iter 100	Best PE: 0.7017	Test Acc: 46.00%
Iter 150	Best PE: 0.6980	Test Acc: 45.00%
Iter 199	Best PE: 0.6972	Test Acc: 49.00%



Pi-Sigma nails 55.0% in ~50 iters with a clean PE drop ($0.6918 \rightarrow 0.6909$), CRO's fast, stable reactions crush local search.

Conclusion

Pi-Sigma is the clear winner — consistently achieving 54.5–58.0% test accuracy with stable, smooth convergence across gradient descent, GA, PSO, and CRO. Its sum-then-product structure supports reliable optimization and generalization, making it robust and practical for binary price direction prediction.

Sigma-Pi is highly unstable, despite theoretical expressiveness, it fails to exceed 50% accuracy in most runs (46.0–49.5%), with loss spikes, wild oscillations, and catastrophic forgetting under

both gradient and population-based methods. The product-then-sum design amplifies gradients and disrupts search, rendering standard training ineffective.

Optimization Method Impact:

GA is the top performer, delivers the highest accuracy (58.0%) with smooth, steady fitness growth and stable generalization, excelling at escaping local minima in complex weight spaces.

CRO is the fastest, achieving near-optimal Pi-Sigma performance (55.0%) in just ~50 iterations via aggressive local refinement, but fails catastrophically on Sigma-Pi due to population-altering reactions.

PSO offers moderate success (53.0%) but suffers from early overshooting and oscillations, making it less reliable.

Gradient descent is fast and stable for Pi-Sigma but completely fails on Sigma-Pi due to gradient explosions.