# CI Course - EX11

## Theory Overview

### PSO + ANN:

Throughout the course, we have emphasized the significant impact of hyperparameters on model performance. Today, we will employ the Particle Swarm Optimization (PSO) algorithm to discover the optimal hyperparameters for a fully connected neural network.

### PSO reminder:

Particle Swarm Optimization (PSO) is an iterative optimization algorithm that updates a group of candidate solutions known as particles. Each particle possesses a position and velocity within the search space. During each iteration, particles are modified based on their inertia, personal best positions, and the global best position of the entire swarm.

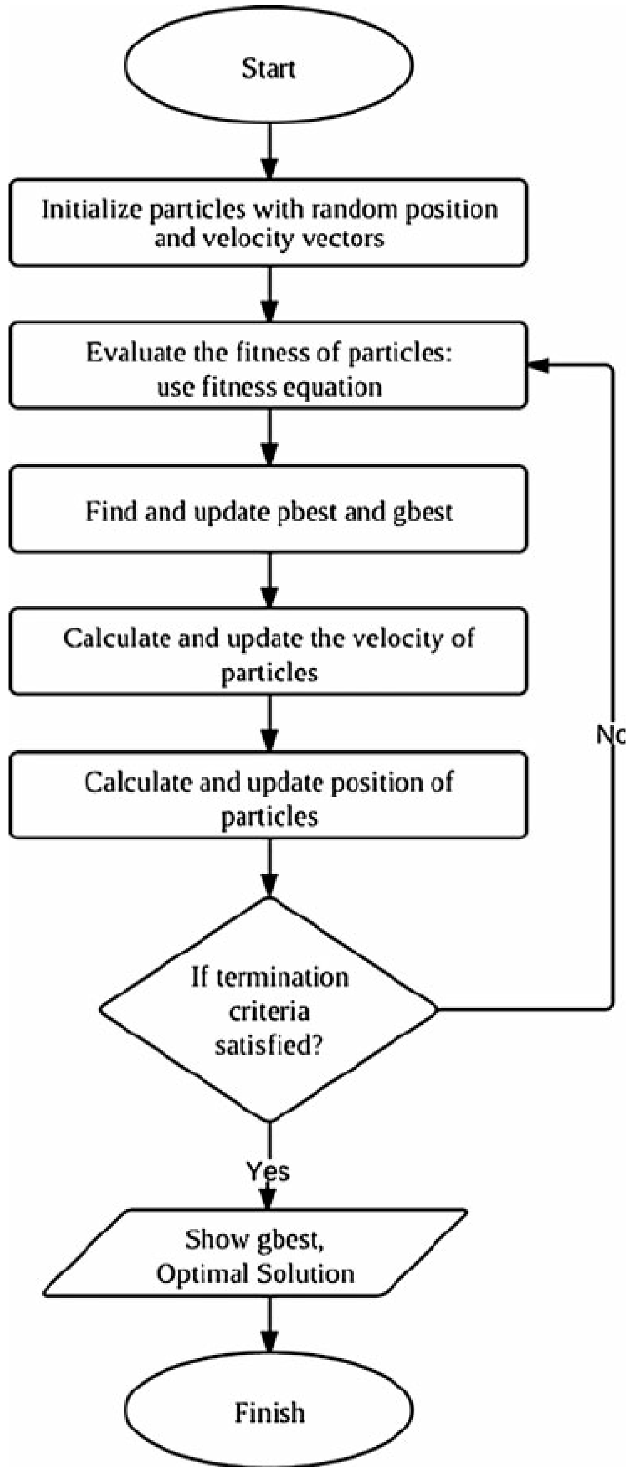
The update rule for each particle can be expressed as follows:

$$P_i^{t+1} = P_i^t + V_i^{t+1}$$

$$V_i^{t+1} = \underbrace{wV_i^t}_{\text{Inertia}} + \underbrace{c_1 r_1 (P_{best(i)}^t - P_i^t)}_{\text{Cognitive (Personal)}} + \underbrace{c_2 r_2 (P_{bestglobal}^t - P_i^t)}_{\text{Social (Global)}}$$

Particle update [Original Image]

Here, $V[i]$ and $P[i]$ represent the velocity and position of particle i, respectively. The inertia weight is denoted as $w$, while $c1$ and $c2$ are acceleration constants. $r1$ and $r2$ represents a random numbers between 0 and 1. Additionally, $Pbest[i]$ represents the personal best position of particle $i$, and $Pbest_{global}$ signifies the global best position of the entire swarm.


Start
Initialize particles with random position and velocity vectors
Evaluate the fitness of particles: use fitness equation
Find and update pbest and gbest
Calculate and update the velocity of particles
Calculate and update position of particles
If termination criteria satisfied?
No
Yes
Show gbest, Optimal Solution
Finish

After updating all particles, their positions are evaluated using an objective function to determine their fitness. Subsequently, the personal best positions and global best position are updated based on these fitness values. The algorithm continues iterating until a stopping criterion is met.

---

**Exersice - PSO optimization of digits classification NN**

In this exercise we will utilize a custom PSO class, as shown below:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
# from PSO import PSO
import torch.nn.functional as F

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

%matplotlib qt

class PSO():
    def __init__(self, live_plot = False):
        # Hyperparameters of the algorithm
        self.c1 = 0.1 # cognitive acceleration coefficient
        self.c2 = 0.1 # social acceleration coefficient
        self.w = 0.8 # inertia weight

        self.live_plot = live_plot # whether to plot the results in real-time
        np.random.seed(100) # set random seed for reproducibility

    def define_problem(self, func = None, n_particles = 20, bounds = []):
        # Define the optimization problem
        self.n_particles = n_particles # number of particles in the swarm
        self.ndim = bounds.shape[1] # dimensionality of the problem

        self.func = func # objective function to minimize

        # Initialize particles randomly within the bounds
        self.X = np.random.rand(self.n_particles, self.ndim) * (bounds[1,:]-bounds[0,:]) + b

        # Initialize particle velocities randomly
        self.V = np.random.randn(self.n_particles, self.ndim) * 0.1
        self.bounds = bounds

        # Initialize personal best positions as first generation
        self.pbest = self.X.copy()

        # Evaluate objective function for personal best positions
        self.pbest_obj = self.func(self.X)

        # Initialize global best position as the best personal best position
        self.gbest = self.pbest[self.pbest_obj.argmin(), :].copy()

        # Evaluate objective function for global best position
        self.gbest_obj = self.pbest_obj.min()

    def update(self):
        # Update particle positions and velocities
        r1, r2 = np.random.rand(2) # generate random numbers for stochastic update
        V = self.w * self.V + self.c1*r1*(self.pbest - self.X) + self.c2*r2*(self.gbest.resh

        X_temp = self.X + self.V

        for i in range(self.n_particles): # Check for constraints
            if np.all(X_temp[i,:] > self.bounds[0,:]) and np.all(X_temp[i,:] < self.bounds[
                # Update particle position if it is within the bounds
                self.X[i,:] = X_temp[i,:].copy()

        obj = self.func(self.X)

        # Update personal best positions and objective values
        self.pbest[(self.pbest_obj >= obj), :] = self.X[(self.pbest_obj >= obj), :]
        self.pbest_obj = np.array([self.pbest_obj, obj]).min(axis=0)

        # Update global best position and objective value
        self.gbest = self.pbest[self.pbest_obj.argmin(), :]
        self.gbest_obj = self.pbest_obj.min()

    def run(self, max_iter = 1000):
        # Run the optimization algorithm for a specified number of iterations
        self.G = [] # list to store the global best objective value at each iteration

        for i in range(max_iter):
            # Update particle positions and velocities
            self.update()

            print(f'Iteration {i}, best fit so far: {np.round(self.gbest_obj, 5)}')
            # Store global best objective value at this iteration
            self.G.append(self.gbest_obj.copy())

            if self.live_plot:
                # Plot results in real-time if live_plot is True
                self.plot(iter = i)

        return self.gbest, self.gbest_obj

    def plot(self, iter = None):
        plt.figure(0)

        if self.live_plot:
            plt.clf()

        plt.plot(np.arange(len(self.G)), self.G)

        plt.xlabel('Iterations')
        plt.ylabel('Fit')

        if self.live_plot:
            plt.title(f'Best global fit: {np.round(self.gbest_obj, 5)}')

            if iter > 1:
                plt.xlim([0, iter])

            plt.pause(1e-2)

        else:
            plt.show()
```

In previous exercises, we worked on a familiar task of classifying handwritten digits from images (for further details, refer to EX7).

***However, our focus today is different:***

We aim to utilize the PSO class to discover the optimal parameters for a neural network that minimize the test loss.

The core function of the PSO class is `define_problem`, which requires the objective function and the bounds array as inputs.

The objective function - in this case, refers to the function we want to minimize, which will be the training function of the neural network script.

The `bounds` array - the limits or boundaries of the search space for the hyperparameters of the neural network.

---

**Solution**

Solution steps:

1. Load data and pre-process
2. Define train function and model based on hyperparameters
3. Define the PSO problem
4. Run PSO optimization and print results

---

**Load data and pre-process**

```
data = load_digits() # load the digits dataset
X, y = [], np.array(data.target) # extract the target values

# Reshape the images into 1D arrays
for x in data.images:
    x = x.reshape((-1,))
    X.append(x)
X = np.array(X)

# Scale the data to have zero mean and unit variance
scaler = StandardScaler().fit(X)
X = scaler.transform(X)

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, random_state=42)

# Convert the data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train).to(torch.int64)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test).to(torch.int64)

num_classes = len(np.unique(y_train)) # number of classes in the target variable
num_features = X_train.shape[1] # number of features in the input data

# set dictionary of activation functions
A = {0: nn.ReLU(), 1: nn.Tanh(), 2: nn.Sigmoid(), 3: nn.LeakyReLU()}
```

---

**Define train function and model based on hyperparameters** The NN hyperparameters will be specified by `Z` array.

```
def train_function(Z):

    L_test = [] # list to store the test loss for each set of hyperparameters

    for z in Z:
        learning_rate = z[0] # learning rate for the optimizer
        batch_size = int(z[1]) # batch size for training
        activation_function = A[int(z[2])] # activation function for the hidden layers
        h = int(z[3]) # number of hidden layers
        m = int(z[4]) # number of neurons in each hidden layer
        p_do = z[5] # dropout rate for the hidden layers
        reg = z[6] # regularization parameter for weight decay

        # Define the architecture of the neural network using a list of layers
        Layers = [nn.Linear(num_features, m), activation_function]

        for i in range(h):
            Layers.append(nn.Linear(m, m))
            Layers.append(nn.Dropout(p = p_do))
            Layers.append(activation_function)

        Layers.append(nn.Linear(m, num_classes))
        Layers.append(nn.Softmax(dim=1))

        model = nn.Sequential(*Layers) # create a PyTorch model from the list of layers

        loss_fn = nn.CrossEntropyLoss() # define the loss function as cross-entropy loss
        optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay = reg) # 

        epochs = 6 # number of epochs to train for

        for epoch in range(epochs):
            for i in range(0, len(X_train), batch_size):
                Xbatch = X_train[i:i+batch_size,:] # extract a batch of training data

                y_pred = model(Xbatch) # compute predictions using a forward pass through th

                ybatch = y_train[i:i+batch_size] # extract corresponding target values

                loss = loss_fn(y_pred, ybatch) # compute the loss

                optimizer.zero_grad() # zero the gradients before running the backward pass

                loss.backward() # compute gradients using backpropagation

                optimizer.step() # update model parameters using gradient descent
        L_test.append(float(loss_fn(model(X_test), y_test).detach().numpy())) # compute tes

    return np.array(L_test)
```

---

### Define the PSO problem

```
bounds = np.array([[1e-6, 2, 0, 0, 5, 0, 0], [1e-1, 300, 4, 10, 70, 0.8, 1e-3]]) # correspo
n_particles = 10
max_iter =10
P = PSO(live_plot = True) # create an instance of the PSO class
P.define_problem(func = train_function, n_particles = n_particles, bounds = bounds) # defin
```

---

### Run PSO optimization and print results

```
z, f = P.run(max_iter = max_iter) # run the optimization algorithm

learning_rate = z[0]
batch_size = int(z[1])
activation_function = A[int(z[2])]
h = int(z[3]) # Number of hidden layers
m = int(z[4]) # Number of neurons in each hidden layer
p_do = z[5] # Dropout value
reg = z[6] # Regularization

print(f'Solution: {z}')
print(f'--- Best fitness {f} with:')
print(f'Learning rate: {learning_rate}')
print(f'Batch size: {batch_size}')
print(f'Activation function: {str(activation_function)}')
print(f'{h} hidden layers with {m} neurons each')
print(f'Dropout: {p_do}, regularization: {reg}')
```

```
Iteration 0, best fit so far: 1.51679
Iteration 1, best fit so far: 1.51679
Iteration 2, best fit so far: 1.51679
Iteration 3, best fit so far: 1.51268
Iteration 4, best fit so far: 1.51036
Iteration 5, best fit so far: 1.51036
Iteration 6, best fit so far: 1.51036
Iteration 7, best fit so far: 1.51036
Iteration 8, best fit so far: 1.51036
Iteration 9, best fit so far: 1.51036
Solution: [9.40030419e-02 2.45659515e+02 1.34444780e+00 1.75410454e+00
```

```
 2.92340830e+01 4.55080588e-03 2.52426353e-04]
--- Best fitness 1.510364055633545 with:
Learning rate: 0.09400304193241785
Batch size: 245
Activation function: Tanh()
1 hidden layers with 29 neurons each
Dropout: 0.004550805882058739, regularization: 0.00025242635344484044
```

## Summary

In this class we cover:

1. Short reminder on PSO
2. Use PSO to optimize NN for digits classification.

Good luck on the exams!