```
IST1100266 Saúl Sandinha Gomes -
https://github.com/SaulSandinha/Assignment-2-sistemas-inteligentes
```

ANFIS - Regressão

```
import numpy as np
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import
mean_squared_error,accuracy_score,classification_report
import skfuzzy as fuzz
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import pandas
```

This section loads and prepares the dataset for the model:

Dataset choice: The code selects a regression dataset (diabetes from scikit-learn).

Feature extraction: X contains the input features as a NumPy array.

Target extraction: y contains the output values (continuous targets for regression).

```
# CHOOSE DATASET

# Regression dataset
data = datasets.load_diabetes(as_frame=True)

# Classification dataset
#data = datasets.fetch_openml("diabetes",version=1, as_frame=True)

X = data.data.values
y = data.target.values

# Converter labels em binário (0 = negativo, 1 = positivo) (só usado
em classification)
#y= np.array([1 if val == "tested_positive" else 0 for val in y])

# Converter para tensor PyTorch (coluna) ( só usado em classification)
#y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)


X.shape

(442, 10)
```

This section splits the dataset into training and testing subsets:

Purpose: Separating the dataset ensures that the model is evaluated on unseen data, providing a realistic estimate of generalization performance and preventing overfitting.

test_size = 0.2: Allocates 20% of the dataset to testing and 80% to training, a common split ratio that balances training data volume and evaluation reliability.

Random state: Setting random_state=42 ensures reproducibility of the split, so results remain consistent across runs.

```
#train test spliting
test_size=0.2
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=test_size,
random_state=42)
```

This section standardizes the input features to improve model performance:

Purpose: Standardization rescales features to have zero mean and unit variance, which helps improve convergence speed and stability in gradient-based learning algorithms.

fit_transform on training data: Learns the scaling parameters (mean and standard deviation) from the training set and applies the transformation.

transform on test data: Uses the same scaling parameters learned from the training set to ensure consistent feature representation and avoid data leakage.

Standardizing features is a fundamental preprocessing step, particularly important for models sensitive to feature scale.

```
# Standardize features
scaler=StandardScaler()
Xtr= scaler.fit_transform(Xtr)
Xte= scaler.transform(Xte)
```

This section applies Fuzzy C-Means (FCM) to the training data. Unlike k-means, FCM assigns each sample a degree of membership to each cluster, which is useful when classes overlap.

Parameters: n_clusters = 2 (binary setting), m = 2 (controls fuzziness; higher → softer memberships).

Preprocessing: Features Xtr are concatenated with labels ytr into Xexp, then transposed since skfuzzy expects input as features × samples.

Outputs: centers (cluster centroids), u (membership matrix), and fpc (partition coefficient, measures clustering quality).

In short, this step performs a soft clustering that models uncertainty by allowing points to partially belong to multiple clusters.

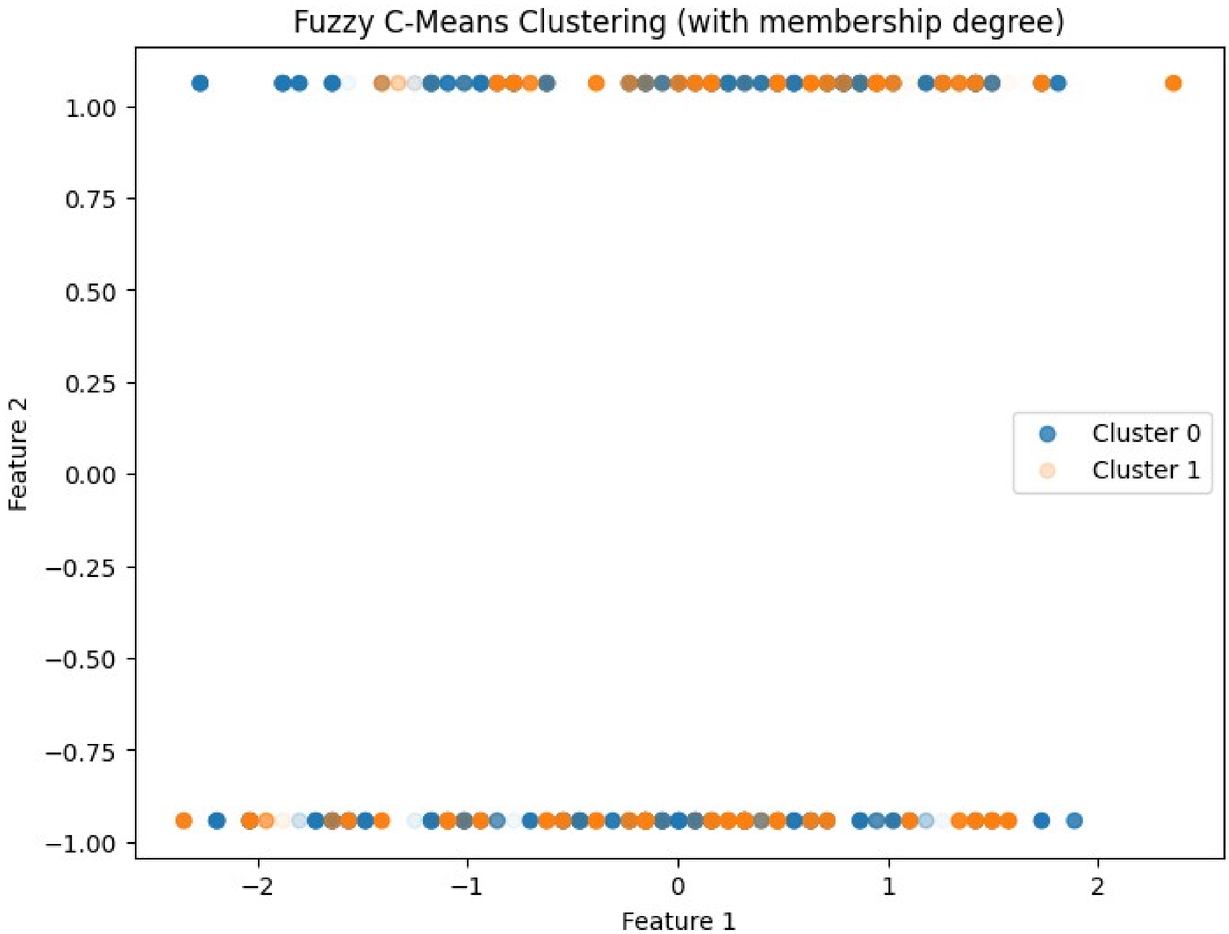
```
 # Number of clusters
n_clusters = 2
m=2
```

```python
# Concatenate target for clustering
Xexp=np.concatenate([Xtr, ytr.reshape(-1, 1)], axis=1)
#Xexp=Xtr

# Transpose data for skfuzzy (expects features x samples)
Xexp_T = Xexp.T

# Fuzzy C-means clustering
centers, u, u0, d, jm, p, fpc = fuzz.cluster.cmeans(
    Xexp_T, n_clusters, m=m, error=0.005, maxiter=1000, init=None,
)

centers.shape
```

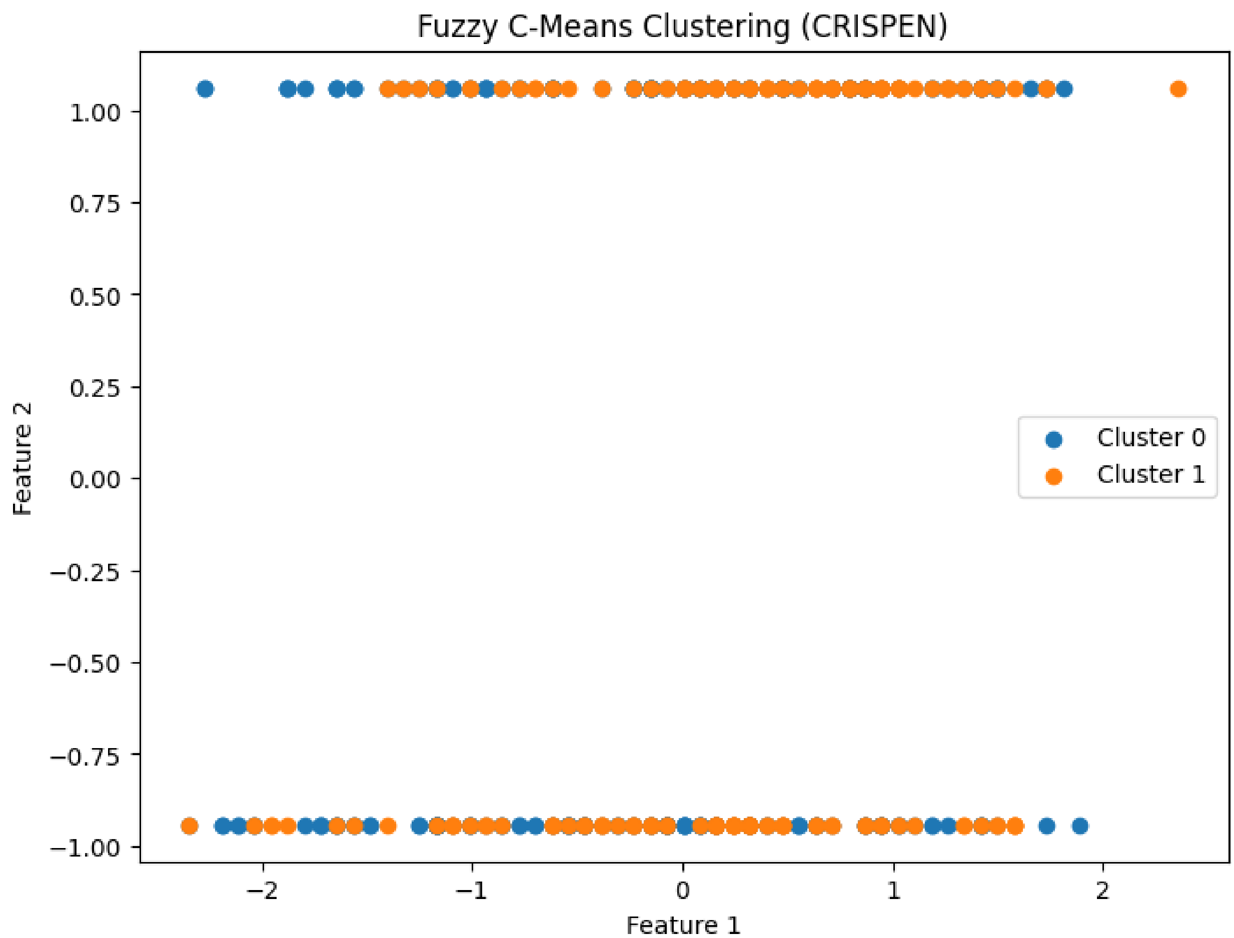
```
(2, 11)
```

```python
# Compute sigma (spread) for each cluster
sigmas = []
for j in range(n_clusters):
    # membership weights for cluster j, raised to m
    u_j = u[j, :] ** m
    # weighted variance for each feature
    var_j = np.average((Xexp - centers[j])**2, axis=0, weights=u_j)
    sigma_j = np.sqrt(var_j)
    sigmas.append(sigma_j)
sigmas=np.array(sigmas)

# Hard clustering from fuzzy membership
cluster_labels = np.argmax(u, axis=0)
print("Fuzzy partition coefficient (FPC):", fpc)

# Plot first two features with fuzzy membership
plt.figure(figsize=(8,6))
for j in range(n_clusters):
    plt.scatter(
        Xexp[cluster_labels == j, 0],             # Feature 1
        Xexp[cluster_labels == j, 1],             # Feature 2
        alpha=u[j, :],           # transparency ~ membership
        label=f'Cluster {j}'
    )

plt.title("Fuzzy C-Means Clustering (with membership degree)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
```

```
Fuzzy partition coefficient (FPC): 0.8556210024955945
```

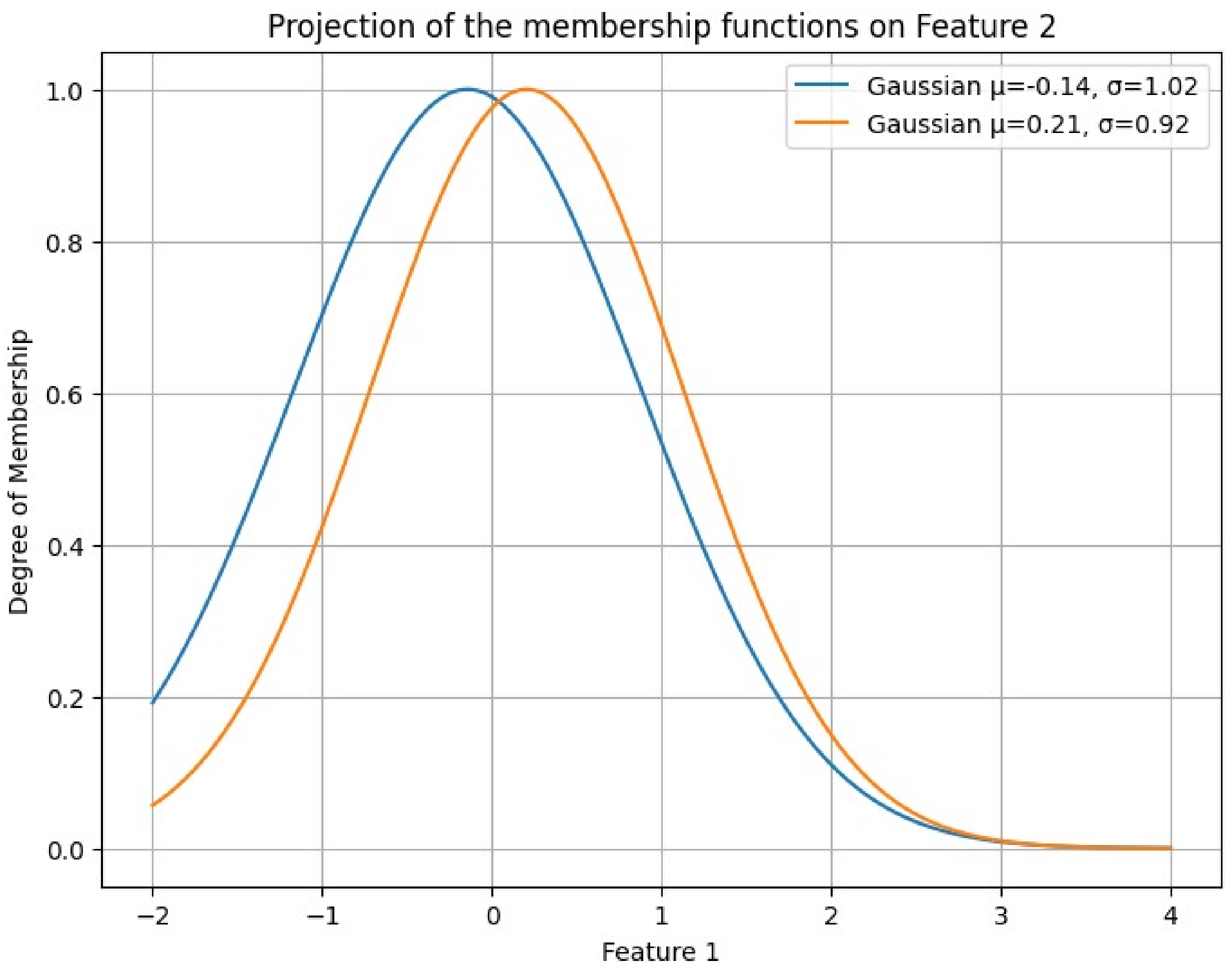



```python
# Plot first two features with cluster assignments
plt.figure(figsize=(8,6))
for j in range(n_clusters):
    plt.scatter(
        Xexp[cluster_labels == j, 0],
        Xexp[cluster_labels == j, 1],
        label=f'Cluster {j}'
    )

plt.title("Fuzzy C-Means Clustering (CRISPEN)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
```




```
# Gaussian formula
def gaussian(x, mu, sigma):
    return np.exp(-0.5 * ((x - mu)/sigma)**2)

lin=np.linspace(-2, 4, 500)
plt.figure(figsize=(8,6))

y_aux=[]
feature=0
for j in range(n_clusters):
# Compute curves
    y_aux.append(gaussian(lin, centers[j,feature], sigmas[j,feature]))

# Plot
    plt.plot(lin, y_aux[j], label=f"Gaussian
μ={np.round(centers[j,feature],2)},
σ={np.round(sigmas[j,feature],2)}")

plt.title("Projection of the membership functions on Feature 2")
plt.xlabel("Feature 1")
plt.ylabel("Degree of Membership")
```

```
plt.legend()
plt.grid(True)
plt.show()
```




```
# ---------------------------
# Gaussian Membership Function
# ---------------------------
class GaussianMF(nn.Module):
    def __init__(self, centers, sigmas, agg_prob):
        super().__init__()
        self.centers = nn.Parameter(torch.tensor(centers,
dtype=torch.float32))
        self.sigmas = nn.Parameter(torch.tensor(sigmas,
dtype=torch.float32))
        self.agg_prob=agg_prob

    def forward(self, x):
        # Expand for broadcasting
        # x: (batch, 1, n_dims), centers: (1, n_rules, n_dims),
```

```
sigmas: (1, n_rules, n_dims)
        diff = abs((x.unsqueeze(1) -
self.centers.unsqueeze(0))/self.sigmas.unsqueeze(0)) #(batch, n_rules,
n_dims)

        # Aggregation
        if self.agg_prob:
            dist = torch.norm(diff, dim=-1)  # (batch, n_rules) #
probablistic intersection
        else:
            dist = torch.max(diff, dim=-1).values  # (batch, n_rules)
# min intersection (min instersection of normal funtion is the same as
the max on dist)

        return torch.exp(-0.5 * dist ** 2)


# ---------------------------
# TSK Model
# ---------------------------
class TSK(nn.Module):
    def __init__(self, n_inputs, n_rules, centers,
sigmas,agg_prob=False):
        super().__init__()
        self.n_inputs = n_inputs
        self.n_rules = n_rules

        # Antecedents (Gaussian MFs)

        self.mfs=GaussianMF(centers, sigmas,agg_prob)

        # Consequents (linear functions of inputs)
        # Each rule has coeffs for each input + bias
        self.consequents = nn.Parameter(
            torch.randn(n_inputs + 1,n_rules)
        )

    def forward(self, x):
        # x: (batch, n_inputs)
        batch_size = x.shape[0]

        # Compute membership values for each input feature
        # firing_strengths: (batch, n_rules)
        firing_strengths = self.mfs(x)

        # Normalize memberships
        # norm_fs: (batch, n_rules)
        norm_fs = firing_strengths / (firing_strengths.sum(dim=1,
keepdim=True) + 1e-9)
```

```
        # Consequent output (linear model per rule)
        x_aug = torch.cat([x, torch.ones(batch_size, 1)], dim=1)  #
add bias

        rule_outputs = torch.einsum("br,rk->bk", x_aug,
self.consequents)  # (batch, rules)
        # Weighted sum
        output = torch.sum(norm_fs * rule_outputs, dim=1,
keepdim=True)

        return output, norm_fs, rule_outputs


# ---------------------------
# Least Squares Solver for Consequents (TSK)
# ---------------------------
def train_ls(model, X, y):
    with torch.no_grad():
        _, norm_fs, _ = model(X)

        # Design matrix for LS: combine normalized firing strengths
with input
        X_aug = torch.cat([X, torch.ones(X.shape[0], 1)], dim=1)

        Phi = torch.einsum("br,bi->bri", X_aug,
norm_fs).reshape(X.shape[0], -1)

        # Solve LS: consequents = (Phi^T Phi)^-1 Phi^T y

        theta= torch.linalg.lstsq(Phi, y).solution


        model.consequents.data =
theta.reshape(model.consequents.shape)



# ---------------------------
# Gradient Descent Training
# ---------------------------
def train_gd(model, X, y, epochs=100, lr=1e-3):
    optimizer = optim.Adam(model.parameters(), lr=lr)
    criterion = nn.MSELoss()
    for _ in range(epochs):
        optimizer.zero_grad()
        y_pred, _, _ = model(X)
        loss = criterion(y_pred, y)
        #print(loss)
        loss.backward()
        optimizer.step()
```

This function implements the hybrid learning algorithm for ANFIS, which alternates between gradient descent (GD) and least squares (LS):

Step A (GD): Update the antecedent parameters (membership functions) while freezing the consequents. This tunes the fuzzy sets to better represent the input space.

Step B (LS): Update the consequent parameters (linear coefficients) using least squares, while keeping the antecedents fixed. This ensures optimal rule outputs given the current fuzzy partitions.

By alternating these two steps for several iterations, the model combines the flexibility of GD with the efficiency of LS, achieving faster and more stable convergence than using GD alone.

```
# ---------------------------
# Hybrid Training (Classic ANFIS)
# ---------------------------
def train_hybrid_anfis(model, X, y, max_iters=10, gd_epochs=20, lr=1e-
3):
    train_ls(model, X, y)
    for _ in range(max_iters):
        # Step A: GD on antecedents (freeze consequents)
        model.consequents.requires_grad = False
        train_gd(model, X, y, epochs=gd_epochs, lr=lr)

        # Step B: LS on consequents (freeze antecedents)
        model.consequents.requires_grad = True
        model.mfs.requires_grad = False
        train_ls(model, X, y)

        # Re-enable antecedents
        model.mfs.requires_grad = True

# ---------------------------
# Alternative Hybrid Training (LS+ gradient descent on all)
# ---------------------------
def train_hybrid(model, X, y, epochs=100, lr=1e-4):
    # Step 1: LS for consequents
    train_ls(model, X, y)
    # Step 2: GD fine-tuning
    train_gd(model, X, y, epochs=epochs, lr=lr)

# Build model
model = TSK(n_inputs=Xtr.shape[1], n_rules=n_clusters,
centers=centers[:,:-1], sigmas=sigmas[:,:-1])

Xtr = torch.tensor(Xtr, dtype=torch.float32)
ytr = torch.tensor(ytr, dtype=torch.float32).reshape(-1,1)
Xte = torch.tensor(Xte, dtype=torch.float32)
yte = torch.tensor(yte, dtype=torch.float32).reshape(-1,1)
```

```
# Training with LS:
train_hybrid_anfis(model, Xtr, ytr, max_iters=10, gd_epochs=20, lr=1e-
3)

y_pred, _, _=model(Xte)
#performance metric for classification
#print(f'ACC:
{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.5)}')
#classification
#performance metric for regression
print(f'MSE:
{mean_squared_error(yte.detach().numpy(),y_pred.detach().numpy())}')
#regression

MSE:2655.36083984375
```

Redes Neuronais - Regressão

```
import numpy as np
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import
mean_squared_error,accuracy_score,classification_report
import matplotlib.pyplot as plt
import torch.nn.functional as F
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
import pandas
```

This section loads and prepares the dataset for the model:

Dataset choice: The active line loads a regression dataset (diabetes from scikit-learn). The commented section provides an alternative option for a classification task.

Feature extraction: X stores the input features as a NumPy array, representing the predictor variables.

Target extraction: y stores the target variable values, which are continuous in the regression case.

Shape inspection: X.shape verifies the dimensions of the feature matrix, ensuring correct dataset structure before proceeding.

```
# CHOOSE DATASET

# Regression dataset
data = datasets.load_diabetes(as_frame=True)
```

```
# Classification dataset
#data = datasets.fetch_openml("diabetes",version=1, as_frame=True)

X = data.data.values
y = data.target.values
X.shape
```

```
(442, 10)
```

```
#train test spliting
test_size=0.2
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=test_size,
random_state=42)

# Standardize features
scaler=StandardScaler()
Xtr= scaler.fit_transform(Xtr)
Xte= scaler.transform(Xte)
```

This class defines a fully connected neural network for regression/classification. The architecture consists of four hidden layers with 128 neurons each, followed by an output layer.

Hidden layers: Each linear layer is followed by a ReLU activation, introducing non-linearity.

Dropout: Applied after each hidden layer (p=0.5) to reduce overfitting by randomly deactivating neurons during training.

Output layer: Maps the final hidden representation to the desired output dimension (default = 1).

Overall, this deep MLP captures complex input–output relationships while dropout regularization helps improve generalization.

```
class MLP(nn.Module):
    def __init__(self, input_size, output_size=1, dropout_prob=0.5):
        super(MLP, self).__init__()

        self.fc1 = nn.Linear(input_size, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, 128)
        self.fc4 = nn.Linear(128, 128)
        self.out = nn.Linear(128, output_size)

        self.dropout = nn.Dropout(p=dropout_prob)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)

        x = F.relu(self.fc2(x))
        x = self.dropout(x)
```

```
        x = F.relu(self.fc3(x))
        x = self.dropout(x)

        x = F.relu(self.fc4(x))
        x = self.dropout(x)

        x = self.out(x)
        return x
```

The following hyperparameters control the training process:

num_epochs = 500: Number of complete passes through the training dataset, allowing the model sufficient iterations to converge.

lr = 0.00025: Learning rate for gradient-based optimization. A small value ensures stable updates and avoids overshooting minima.

dropout = 0.05: Low dropout probability, meaning most neurons are retained. This provides slight regularization without heavily reducing model capacity.

batch_size = 64: Number of samples per gradient update, balancing computational efficiency and gradient stability.

These values jointly define the trade-off between convergence speed, stability, and generalization performance.

```
num_epochs=500
lr=0.00025
dropout=0.05
batch_size=64

Xtr = torch.tensor(Xtr, dtype=torch.float32)
ytr = torch.tensor(ytr, dtype=torch.float32)
Xte = torch.tensor(Xte, dtype=torch.float32)
yte = torch.tensor(yte, dtype=torch.float32)

# Wrap Xtr and ytr into a dataset
train_dataset = TensorDataset(Xtr, ytr)

# Create DataLoader
train_dataloader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
```

This section prepares the model for training:

Model instantiation: Creates an MLP with input size matching the feature dimension of Xtr and the specified dropout rate. The model is moved to the selected device for computation.

Loss function: Uses Mean Squared Error (MSE), appropriate for regression tasks.

Optimizer: Adam optimizer is chosen for its adaptive learning rate and efficient convergence, with learning rate set by lr.

```
# Model, Loss, Optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = MLP(input_size=Xtr.shape[1], dropout_prob=dropout).to(device)
#criterion = nn.BCEWithLogitsLoss()  # for binary classification
criterion = nn.MSELoss() #for regression
optimizer = optim.Adam(model.parameters(), lr=lr)
```

This block implements the iterative training process for the MLP:

Epoch loop: Repeats the training process num_epochs times to allow the model to progressively learn patterns in the data.

Batch processing: Training data is processed in batches (batch_size = 64), improving computational efficiency and providing smoother gradient estimates.

Forward pass: The model computes predictions (logits) for the current batch.

Loss computation: Calculates the discrepancy between predictions and ground truth using the chosen loss function (criterion).

Backward pass: Gradients are computed via backpropagation, and the optimizer updates model parameters accordingly.

Loss tracking: The average loss per epoch is computed and printed, providing insight into training convergence.

This loop embodies the core supervised learning procedure, iteratively refining model parameters to minimize the loss function.

```
# Training loop
for epoch in range(num_epochs):
    model.train()
    epoch_loss = 0.0

    for batch_x, batch_y in train_dataloader:
        batch_x = batch_x.to(device)
        batch_y = batch_y.to(device)

        logits = model(batch_x)
        loss = criterion(logits, batch_y.view(-1, 1))

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
```

```python
    avg_loss = epoch_loss / len(train_dataloader)
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {avg_loss:.4f}")
```

```
Epoch [1/500], Loss: 29438.4925
Epoch [2/500], Loss: 29268.9616
Epoch [3/500], Loss: 29619.7917
Epoch [4/500], Loss: 29000.3421
Epoch [5/500], Loss: 29456.8353
Epoch [6/500], Loss: 29823.8522
Epoch [7/500], Loss: 29636.6051
Epoch [8/500], Loss: 29275.9593
Epoch [9/500], Loss: 29357.8249
Epoch [10/500], Loss: 28433.1009
Epoch [11/500], Loss: 26363.3385
Epoch [12/500], Loss: 26054.6383
Epoch [13/500], Loss: 24292.2386
Epoch [14/500], Loss: 21670.8688
Epoch [15/500], Loss: 18251.5378
Epoch [16/500], Loss: 14319.1774
Epoch [17/500], Loss: 11059.0828
Epoch [18/500], Loss: 8366.9289
Epoch [19/500], Loss: 6435.3175
Epoch [20/500], Loss: 5556.5446
Epoch [21/500], Loss: 5172.9698
Epoch [22/500], Loss: 4834.4103
Epoch [23/500], Loss: 4475.1899
Epoch [24/500], Loss: 4315.3626
Epoch [25/500], Loss: 4441.7754
Epoch [26/500], Loss: 4152.5026
Epoch [27/500], Loss: 3959.4728
Epoch [28/500], Loss: 3958.5800
Epoch [29/500], Loss: 3798.0827
Epoch [30/500], Loss: 3810.9907
Epoch [31/500], Loss: 4009.1216
Epoch [32/500], Loss: 3656.0756
Epoch [33/500], Loss: 3600.6818
Epoch [34/500], Loss: 3724.3566
Epoch [35/500], Loss: 3730.9461
Epoch [36/500], Loss: 3594.0675
Epoch [37/500], Loss: 3585.5535
Epoch [38/500], Loss: 3380.7815
Epoch [39/500], Loss: 3368.8374
Epoch [40/500], Loss: 3381.6723
Epoch [41/500], Loss: 3603.8180
Epoch [42/500], Loss: 3375.2730
Epoch [43/500], Loss: 3293.8695
Epoch [44/500], Loss: 3262.5488
Epoch [45/500], Loss: 3314.6630
Epoch [46/500], Loss: 3348.5597
Epoch [47/500], Loss: 3420.0932
```

```
Epoch [48/500], Loss: 3337.0474
Epoch [49/500], Loss: 3206.0237
Epoch [50/500], Loss: 3389.5883
Epoch [51/500], Loss: 3249.3602
Epoch [52/500], Loss: 3288.1759
Epoch [53/500], Loss: 3320.1180
Epoch [54/500], Loss: 3274.9884
Epoch [55/500], Loss: 3225.3386
Epoch [56/500], Loss: 3162.1187
Epoch [57/500], Loss: 3267.4120
Epoch [58/500], Loss: 3106.5679
Epoch [59/500], Loss: 3106.7855
Epoch [60/500], Loss: 3221.0451
Epoch [61/500], Loss: 3110.2726
Epoch [62/500], Loss: 3080.8462
Epoch [63/500], Loss: 3224.3818
Epoch [64/500], Loss: 3195.6819
Epoch [65/500], Loss: 3123.0848
Epoch [66/500], Loss: 3255.5551
Epoch [67/500], Loss: 3052.5855
Epoch [68/500], Loss: 3075.4068
Epoch [69/500], Loss: 3093.8584
Epoch [70/500], Loss: 3340.2563
Epoch [71/500], Loss: 3006.6293
Epoch [72/500], Loss: 3071.3343
Epoch [73/500], Loss: 2964.5609
Epoch [74/500], Loss: 3003.7375
Epoch [75/500], Loss: 3023.2023
Epoch [76/500], Loss: 3038.6608
Epoch [77/500], Loss: 2957.3546
Epoch [78/500], Loss: 2927.2908
Epoch [79/500], Loss: 3065.9851
Epoch [80/500], Loss: 3016.3453
Epoch [81/500], Loss: 3086.5870
Epoch [82/500], Loss: 2978.5011
Epoch [83/500], Loss: 2954.2867
Epoch [84/500], Loss: 2918.2729
Epoch [85/500], Loss: 2923.4326
Epoch [86/500], Loss: 2893.7437
Epoch [87/500], Loss: 3063.0645
Epoch [88/500], Loss: 2902.4179
Epoch [89/500], Loss: 2987.4475
Epoch [90/500], Loss: 2915.9030
Epoch [91/500], Loss: 2974.2373
Epoch [92/500], Loss: 2884.7771
Epoch [93/500], Loss: 3016.0783
Epoch [94/500], Loss: 2859.2668
Epoch [95/500], Loss: 2868.8790
Epoch [96/500], Loss: 2874.1237
```

```
Epoch [97/500], Loss: 2835.1902
Epoch [98/500], Loss: 2939.6246
Epoch [99/500], Loss: 2876.6966
Epoch [100/500], Loss: 2847.5150
Epoch [101/500], Loss: 2988.9412
Epoch [102/500], Loss: 2882.7598
Epoch [103/500], Loss: 2765.0497
Epoch [104/500], Loss: 2865.7561
Epoch [105/500], Loss: 2933.3009
Epoch [106/500], Loss: 2807.7992
Epoch [107/500], Loss: 2873.6208
Epoch [108/500], Loss: 2685.5379
Epoch [109/500], Loss: 2893.1497
Epoch [110/500], Loss: 2758.8304
Epoch [111/500], Loss: 2863.4759
Epoch [112/500], Loss: 2895.8516
Epoch [113/500], Loss: 2769.5344
Epoch [114/500], Loss: 2795.1974
Epoch [115/500], Loss: 2713.1882
Epoch [116/500], Loss: 2854.9806
Epoch [117/500], Loss: 2669.7054
Epoch [118/500], Loss: 2758.1958
Epoch [119/500], Loss: 2853.6906
Epoch [120/500], Loss: 2766.4571
Epoch [121/500], Loss: 2780.7060
Epoch [122/500], Loss: 2717.0497
Epoch [123/500], Loss: 2715.7707
Epoch [124/500], Loss: 2693.0774
Epoch [125/500], Loss: 2766.3810
Epoch [126/500], Loss: 2709.3773
Epoch [127/500], Loss: 2763.9472
Epoch [128/500], Loss: 2648.3426
Epoch [129/500], Loss: 2695.1273
Epoch [130/500], Loss: 2719.0968
Epoch [131/500], Loss: 2829.1855
Epoch [132/500], Loss: 2710.2338
Epoch [133/500], Loss: 2613.2719
Epoch [134/500], Loss: 2766.5387
Epoch [135/500], Loss: 2737.8736
Epoch [136/500], Loss: 2754.5443
Epoch [137/500], Loss: 2755.5221
Epoch [138/500], Loss: 2678.3986
Epoch [139/500], Loss: 2640.7316
Epoch [140/500], Loss: 2665.1157
Epoch [141/500], Loss: 2850.9909
Epoch [142/500], Loss: 2666.4149
Epoch [143/500], Loss: 2599.7122
Epoch [144/500], Loss: 2622.8579
Epoch [145/500], Loss: 2690.5822
```

```
Epoch [146/500], Loss: 2596.1037
Epoch [147/500], Loss: 2613.4831
Epoch [148/500], Loss: 2647.0136
Epoch [149/500], Loss: 2730.8201
Epoch [150/500], Loss: 2685.1826
Epoch [151/500], Loss: 2657.2109
Epoch [152/500], Loss: 2554.2798
Epoch [153/500], Loss: 2662.5334
Epoch [154/500], Loss: 2603.9888
Epoch [155/500], Loss: 2503.8884
Epoch [156/500], Loss: 2641.8816
Epoch [157/500], Loss: 2632.2279
Epoch [158/500], Loss: 2641.0234
Epoch [159/500], Loss: 2587.1128
Epoch [160/500], Loss: 2742.5282
Epoch [161/500], Loss: 2509.7177
Epoch [162/500], Loss: 2504.4134
Epoch [163/500], Loss: 2588.8318
Epoch [164/500], Loss: 2679.2075
Epoch [165/500], Loss: 2541.1713
Epoch [166/500], Loss: 2524.9761
Epoch [167/500], Loss: 2493.6767
Epoch [168/500], Loss: 2638.3781
Epoch [169/500], Loss: 2491.1411
Epoch [170/500], Loss: 2436.9878
Epoch [171/500], Loss: 2592.6425
Epoch [172/500], Loss: 2471.1237
Epoch [173/500], Loss: 2523.7971
Epoch [174/500], Loss: 2577.7601
Epoch [175/500], Loss: 2674.4446
Epoch [176/500], Loss: 2581.0843
Epoch [177/500], Loss: 2601.9766
Epoch [178/500], Loss: 2630.7310
Epoch [179/500], Loss: 2593.7542
Epoch [180/500], Loss: 2538.8622
Epoch [181/500], Loss: 2642.6250
Epoch [182/500], Loss: 2533.7783
Epoch [183/500], Loss: 2470.4532
Epoch [184/500], Loss: 2379.3033
Epoch [185/500], Loss: 2746.0887
Epoch [186/500], Loss: 2555.6553
Epoch [187/500], Loss: 2481.2769
Epoch [188/500], Loss: 2533.1284
Epoch [189/500], Loss: 2568.0682
Epoch [190/500], Loss: 2365.5416
Epoch [191/500], Loss: 2558.0581
Epoch [192/500], Loss: 2468.4825
Epoch [193/500], Loss: 2547.1917
Epoch [194/500], Loss: 2314.4830
```

```
Epoch [195/500], Loss: 2453.0341
Epoch [196/500], Loss: 2575.8539
Epoch [197/500], Loss: 2485.6687
Epoch [198/500], Loss: 2512.3165
Epoch [199/500], Loss: 2480.9626
Epoch [200/500], Loss: 2467.5569
Epoch [201/500], Loss: 2420.9561
Epoch [202/500], Loss: 2412.4003
Epoch [203/500], Loss: 2568.2590
Epoch [204/500], Loss: 2443.2315
Epoch [205/500], Loss: 2430.0233
Epoch [206/500], Loss: 2451.1175
Epoch [207/500], Loss: 2408.0079
Epoch [208/500], Loss: 2441.1846
Epoch [209/500], Loss: 2471.2928
Epoch [210/500], Loss: 2448.0211
Epoch [211/500], Loss: 2378.2195
Epoch [212/500], Loss: 2401.8159
Epoch [213/500], Loss: 2437.9303
Epoch [214/500], Loss: 2272.3903
Epoch [215/500], Loss: 2473.2563
Epoch [216/500], Loss: 2360.6628
Epoch [217/500], Loss: 2573.5435
Epoch [218/500], Loss: 2470.0984
Epoch [219/500], Loss: 2414.3931
Epoch [220/500], Loss: 2436.3155
Epoch [221/500], Loss: 2429.4119
Epoch [222/500], Loss: 2332.5698
Epoch [223/500], Loss: 2302.2110
Epoch [224/500], Loss: 2411.4492
Epoch [225/500], Loss: 2409.6519
Epoch [226/500], Loss: 2464.5200
Epoch [227/500], Loss: 2241.7741
Epoch [228/500], Loss: 2368.0580
Epoch [229/500], Loss: 2450.8059
Epoch [230/500], Loss: 2427.5117
Epoch [231/500], Loss: 2320.0411
Epoch [232/500], Loss: 2358.6145
Epoch [233/500], Loss: 2207.2958
Epoch [234/500], Loss: 2342.3911
Epoch [235/500], Loss: 2345.8556
Epoch [236/500], Loss: 2379.2786
Epoch [237/500], Loss: 2321.6780
Epoch [238/500], Loss: 2417.2920
Epoch [239/500], Loss: 2283.9997
Epoch [240/500], Loss: 2232.9793
Epoch [241/500], Loss: 2295.3952
Epoch [242/500], Loss: 2358.4865
Epoch [243/500], Loss: 2352.9853
```

```
Epoch [244/500], Loss: 2256.3809
Epoch [245/500], Loss: 2367.8476
Epoch [246/500], Loss: 2377.7551
Epoch [247/500], Loss: 2451.3671
Epoch [248/500], Loss: 2210.1934
Epoch [249/500], Loss: 2425.9869
Epoch [250/500], Loss: 2302.9612
Epoch [251/500], Loss: 2161.3521
Epoch [252/500], Loss: 2325.5830
Epoch [253/500], Loss: 2310.7760
Epoch [254/500], Loss: 2255.7153
Epoch [255/500], Loss: 2250.7434
Epoch [256/500], Loss: 2226.7105
Epoch [257/500], Loss: 2251.1275
Epoch [258/500], Loss: 2183.1008
Epoch [259/500], Loss: 2172.9411
Epoch [260/500], Loss: 2166.1711
Epoch [261/500], Loss: 2132.5334
Epoch [262/500], Loss: 2244.6537
Epoch [263/500], Loss: 2275.8807
Epoch [264/500], Loss: 2405.6937
Epoch [265/500], Loss: 2257.2780
Epoch [266/500], Loss: 2153.7363
Epoch [267/500], Loss: 2284.1818
Epoch [268/500], Loss: 2197.8321
Epoch [269/500], Loss: 2306.2704
Epoch [270/500], Loss: 2100.6698
Epoch [271/500], Loss: 2239.7511
Epoch [272/500], Loss: 2231.2202
Epoch [273/500], Loss: 2118.3276
Epoch [274/500], Loss: 2234.1089
Epoch [275/500], Loss: 2129.9997
Epoch [276/500], Loss: 2316.2012
Epoch [277/500], Loss: 2248.2970
Epoch [278/500], Loss: 2234.1359
Epoch [279/500], Loss: 2172.5023
Epoch [280/500], Loss: 2238.7112
Epoch [281/500], Loss: 2218.9955
Epoch [282/500], Loss: 2238.1648
Epoch [283/500], Loss: 2205.4981
Epoch [284/500], Loss: 2185.7444
Epoch [285/500], Loss: 2204.5215
Epoch [286/500], Loss: 2229.1982
Epoch [287/500], Loss: 2200.2704
Epoch [288/500], Loss: 2146.0352
Epoch [289/500], Loss: 2202.5329
Epoch [290/500], Loss: 2163.6178
Epoch [291/500], Loss: 2090.7217
Epoch [292/500], Loss: 2199.4350
```

```
Epoch [293/500], Loss: 2247.4354
Epoch [294/500], Loss: 2099.1900
Epoch [295/500], Loss: 2183.5129
Epoch [296/500], Loss: 2218.2231
Epoch [297/500], Loss: 2130.3327
Epoch [298/500], Loss: 2094.0930
Epoch [299/500], Loss: 2101.7884
Epoch [300/500], Loss: 2147.7355
Epoch [301/500], Loss: 2124.7002
Epoch [302/500], Loss: 2058.4284
Epoch [303/500], Loss: 2050.9608
Epoch [304/500], Loss: 2237.8130
Epoch [305/500], Loss: 2094.0225
Epoch [306/500], Loss: 2060.9965
Epoch [307/500], Loss: 2162.8378
Epoch [308/500], Loss: 2128.2147
Epoch [309/500], Loss: 2009.5850
Epoch [310/500], Loss: 2220.6780
Epoch [311/500], Loss: 2204.4912
Epoch [312/500], Loss: 2118.9375
Epoch [313/500], Loss: 2127.2874
Epoch [314/500], Loss: 2040.9875
Epoch [315/500], Loss: 1912.1669
Epoch [316/500], Loss: 2131.2912
Epoch [317/500], Loss: 2117.8949
Epoch [318/500], Loss: 1970.4073
Epoch [319/500], Loss: 2089.1612
Epoch [320/500], Loss: 2090.4944
Epoch [321/500], Loss: 2247.8748
Epoch [322/500], Loss: 2051.0224
Epoch [323/500], Loss: 2127.1336
Epoch [324/500], Loss: 2162.7250
Epoch [325/500], Loss: 2054.7989
Epoch [326/500], Loss: 2085.5882
Epoch [327/500], Loss: 2111.9569
Epoch [328/500], Loss: 2039.1014
Epoch [329/500], Loss: 2099.5322
Epoch [330/500], Loss: 1964.1898
Epoch [331/500], Loss: 1956.4593
Epoch [332/500], Loss: 1985.9542
Epoch [333/500], Loss: 1941.2565
Epoch [334/500], Loss: 1952.0217
Epoch [335/500], Loss: 1992.1219
Epoch [336/500], Loss: 1980.9500
Epoch [337/500], Loss: 2015.3063
Epoch [338/500], Loss: 1997.0151
Epoch [339/500], Loss: 1894.2463
Epoch [340/500], Loss: 1956.9394
Epoch [341/500], Loss: 2028.8103
```

```
Epoch [342/500], Loss: 2094.5031
Epoch [343/500], Loss: 1900.6862
Epoch [344/500], Loss: 1839.1451
Epoch [345/500], Loss: 2082.5817
Epoch [346/500], Loss: 2098.2102
Epoch [347/500], Loss: 1966.5328
Epoch [348/500], Loss: 1952.9995
Epoch [349/500], Loss: 1998.4393
Epoch [350/500], Loss: 2074.6342
Epoch [351/500], Loss: 1993.4183
Epoch [352/500], Loss: 1964.1117
Epoch [353/500], Loss: 1958.8746
Epoch [354/500], Loss: 2104.0586
Epoch [355/500], Loss: 2035.9787
Epoch [356/500], Loss: 1937.7076
Epoch [357/500], Loss: 1950.2650
Epoch [358/500], Loss: 1994.3245
Epoch [359/500], Loss: 2001.7006
Epoch [360/500], Loss: 1973.8647
Epoch [361/500], Loss: 1947.2977
Epoch [362/500], Loss: 1958.6678
Epoch [363/500], Loss: 2011.2557
Epoch [364/500], Loss: 1791.3986
Epoch [365/500], Loss: 1972.9401
Epoch [366/500], Loss: 1978.7444
Epoch [367/500], Loss: 1980.6750
Epoch [368/500], Loss: 1865.7227
Epoch [369/500], Loss: 1872.4342
Epoch [370/500], Loss: 1842.5752
Epoch [371/500], Loss: 1867.1660
Epoch [372/500], Loss: 2028.6238
Epoch [373/500], Loss: 1885.3570
Epoch [374/500], Loss: 1907.5974
Epoch [375/500], Loss: 1888.3931
Epoch [376/500], Loss: 1943.6515
Epoch [377/500], Loss: 1999.4481
Epoch [378/500], Loss: 1848.9745
Epoch [379/500], Loss: 1951.8967
Epoch [380/500], Loss: 1883.6128
Epoch [381/500], Loss: 1815.1949
Epoch [382/500], Loss: 1975.5147
Epoch [383/500], Loss: 2048.4680
Epoch [384/500], Loss: 1971.7219
Epoch [385/500], Loss: 1919.0411
Epoch [386/500], Loss: 1849.5365
Epoch [387/500], Loss: 1863.8547
Epoch [388/500], Loss: 1958.9429
Epoch [389/500], Loss: 1863.7494
Epoch [390/500], Loss: 1824.7334
```

```
Epoch [391/500], Loss: 1934.7120
Epoch [392/500], Loss: 1774.5746
Epoch [393/500], Loss: 1884.3492
Epoch [394/500], Loss: 1888.7124
Epoch [395/500], Loss: 1915.0471
Epoch [396/500], Loss: 1730.6133
Epoch [397/500], Loss: 1780.5719
Epoch [398/500], Loss: 1833.6808
Epoch [399/500], Loss: 1801.4333
Epoch [400/500], Loss: 1758.8076
Epoch [401/500], Loss: 1807.9520
Epoch [402/500], Loss: 1829.1902
Epoch [403/500], Loss: 1792.7499
Epoch [404/500], Loss: 1969.5285
Epoch [405/500], Loss: 1812.7976
Epoch [406/500], Loss: 1883.7206
Epoch [407/500], Loss: 1889.8121
Epoch [408/500], Loss: 1892.5653
Epoch [409/500], Loss: 1793.5052
Epoch [410/500], Loss: 1794.4681
Epoch [411/500], Loss: 1760.8629
Epoch [412/500], Loss: 1809.0963
Epoch [413/500], Loss: 1794.1723
Epoch [414/500], Loss: 1746.3659
Epoch [415/500], Loss: 1808.1686
Epoch [416/500], Loss: 1718.5291
Epoch [417/500], Loss: 1786.4752
Epoch [418/500], Loss: 1807.7736
Epoch [419/500], Loss: 1760.1557
Epoch [420/500], Loss: 1714.4806
Epoch [421/500], Loss: 1801.0207
Epoch [422/500], Loss: 1719.1750
Epoch [423/500], Loss: 1651.4006
Epoch [424/500], Loss: 1808.7979
Epoch [425/500], Loss: 1855.5246
Epoch [426/500], Loss: 1908.9558
Epoch [427/500], Loss: 1606.7321
Epoch [428/500], Loss: 1759.4498
Epoch [429/500], Loss: 1692.9073
Epoch [430/500], Loss: 1722.9143
Epoch [431/500], Loss: 1673.9960
Epoch [432/500], Loss: 1736.1084
Epoch [433/500], Loss: 1800.6127
Epoch [434/500], Loss: 1681.0660
Epoch [435/500], Loss: 1713.0716
Epoch [436/500], Loss: 1640.8029
Epoch [437/500], Loss: 1656.4019
Epoch [438/500], Loss: 1783.6526
Epoch [439/500], Loss: 1713.4239
```

```
Epoch [440/500], Loss: 1809.7644
Epoch [441/500], Loss: 1743.5266
Epoch [442/500], Loss: 1682.6921
Epoch [443/500], Loss: 1677.0661
Epoch [444/500], Loss: 1736.6464
Epoch [445/500], Loss: 1807.2656
Epoch [446/500], Loss: 1725.9234
Epoch [447/500], Loss: 1657.5855
Epoch [448/500], Loss: 1754.9121
Epoch [449/500], Loss: 1752.7892
Epoch [450/500], Loss: 1744.2254
Epoch [451/500], Loss: 1825.9842
Epoch [452/500], Loss: 1689.4929
Epoch [453/500], Loss: 1598.4587
Epoch [454/500], Loss: 1692.1117
Epoch [455/500], Loss: 1617.5338
Epoch [456/500], Loss: 1675.3965
Epoch [457/500], Loss: 1670.8938
Epoch [458/500], Loss: 1647.6798
Epoch [459/500], Loss: 1753.4851
Epoch [460/500], Loss: 1758.1408
Epoch [461/500], Loss: 1700.7597
Epoch [462/500], Loss: 1679.9935
Epoch [463/500], Loss: 1709.8305
Epoch [464/500], Loss: 1710.7260
Epoch [465/500], Loss: 1711.0959
Epoch [466/500], Loss: 1598.6013
Epoch [467/500], Loss: 1478.0789
Epoch [468/500], Loss: 1665.6838
Epoch [469/500], Loss: 1653.2387
Epoch [470/500], Loss: 1697.1327
Epoch [471/500], Loss: 1669.8118
Epoch [472/500], Loss: 1593.1218
Epoch [473/500], Loss: 1676.3799
Epoch [474/500], Loss: 1736.9578
Epoch [475/500], Loss: 1595.8201
Epoch [476/500], Loss: 1522.8760
Epoch [477/500], Loss: 1706.4085
Epoch [478/500], Loss: 1714.9787
Epoch [479/500], Loss: 1546.3933
Epoch [480/500], Loss: 1628.3445
Epoch [481/500], Loss: 1628.6630
Epoch [482/500], Loss: 1684.8953
Epoch [483/500], Loss: 1628.4327
Epoch [484/500], Loss: 1491.2651
Epoch [485/500], Loss: 1505.6517
Epoch [486/500], Loss: 1628.0004
Epoch [487/500], Loss: 1560.2391
Epoch [488/500], Loss: 1561.3844
```

```
Epoch [489/500], Loss: 1682.5492
Epoch [490/500], Loss: 1724.4385
Epoch [491/500], Loss: 1701.9223
Epoch [492/500], Loss: 1573.9546
Epoch [493/500], Loss: 1554.6167
Epoch [494/500], Loss: 1602.8807
Epoch [495/500], Loss: 1512.3147
Epoch [496/500], Loss: 1441.6916
Epoch [497/500], Loss: 1704.3566
Epoch [498/500], Loss: 1572.4130
Epoch [499/500], Loss: 1662.7552
Epoch [500/500], Loss: 1650.1271
```

```
model.eval()
y_pred=model(Xte)
#print(f'ACC:
{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.5)}')
#classification
print(f'MSE:
{mean_squared_error(yte.detach().numpy(),y_pred.detach().numpy())}')
#regression
```

```
MSE:2760.140869140625
```

ANFIS - Classificação

```
import numpy as np
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import
mean_squared_error,accuracy_score,classification_report
import skfuzzy as fuzz
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import pandas
```

Dataset Selection and Preparation

This section loads and prepares the dataset for training:

Dataset choice: The code selects a classification dataset (diabetes from OpenML).

Feature and label extraction: X contains input features, y contains target labels.

Binary encoding: For classification, labels are converted to binary values (0 = negative, 1 = positive) to suit supervised learning.

Tensor conversion: Labels are converted into PyTorch tensors with shape (n_samples, 1) for compatibility with the model.

The binary encoding, and tensor conversion is not needed when dealing with regression, where the labels are continuous target values, but is necessary in classification.

```
# CHOOSE DATASET

# Regression dataset
#data = datasets.load_diabetes(as_frame=True)

# Classification dataset
data = datasets.fetch_openml("diabetes",version=1, as_frame=True)

X = data.data.values
y = data.target.values

# Converter labels em binário (0 = negativo, 1 = positivo) (só usado
em classification)
y= np.array([1 if val == "tested_positive" else 0 for val in y])

# Converter para tensor PyTorch (coluna) ( só usado em classification)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)


X.shape
```

```
(768, 8)
```

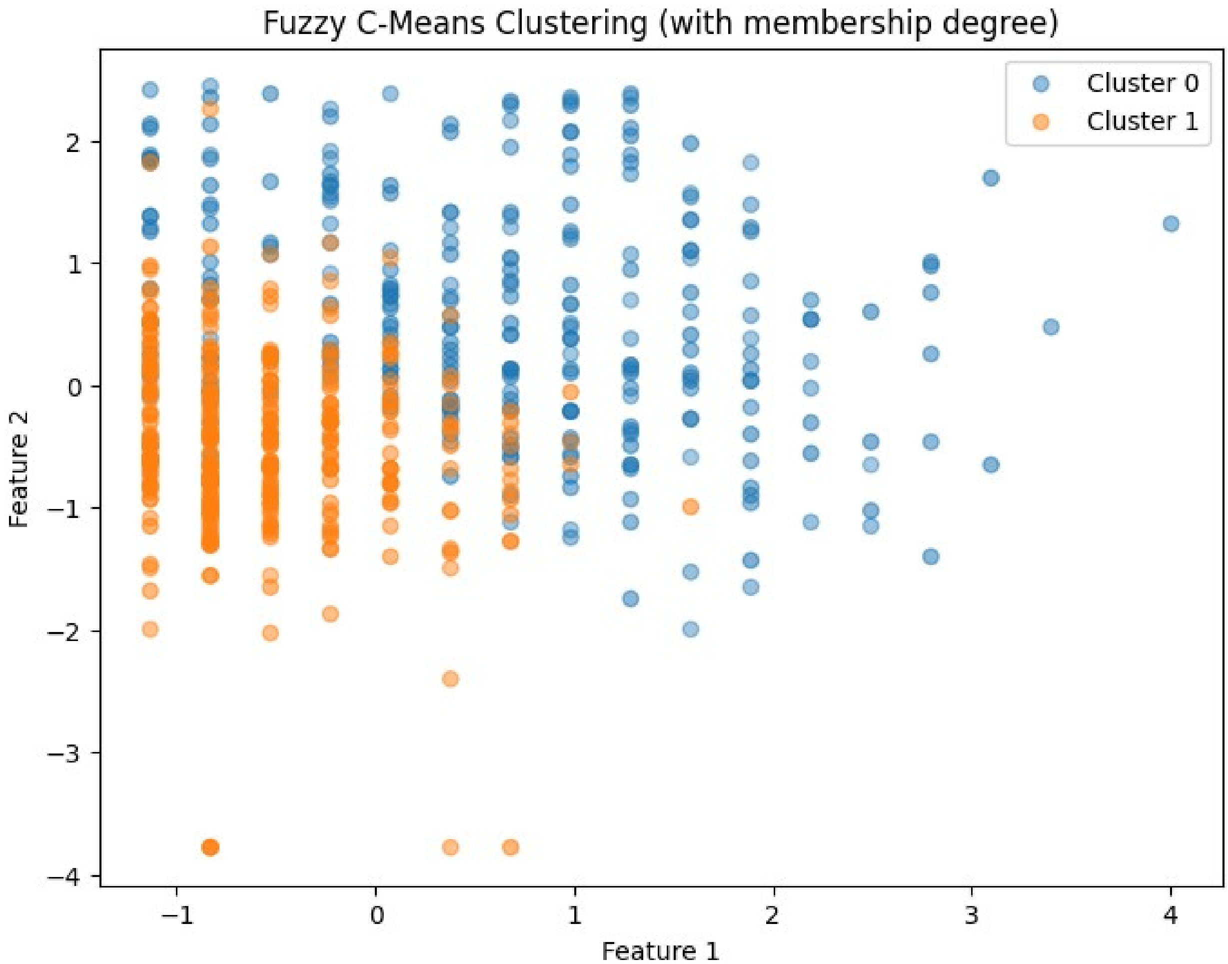
```
#train test spliting
test_size=0.2
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=test_size,
random_state=42)

# Standardize features
scaler=StandardScaler()
Xtr= scaler.fit_transform(Xtr)
Xte= scaler.transform(Xte)

 # Number of clusters
n_clusters = 2
m=2

# Concatenate target for clustering
Xexp=np.concatenate([Xtr, ytr.reshape(-1, 1)], axis=1)
#Xexp=Xtr

# Transpose data for skfuzzy (expects features x samples)
Xexp_T = Xexp.T

# Fuzzy C-means clustering
centers, u, u0, d, jm, p, fpc = fuzz.cluster.cmeans(
```

```
    Xexp_T, n_clusters, m=m, error=0.005, maxiter=1000, init=None,
)

centers.shape
```

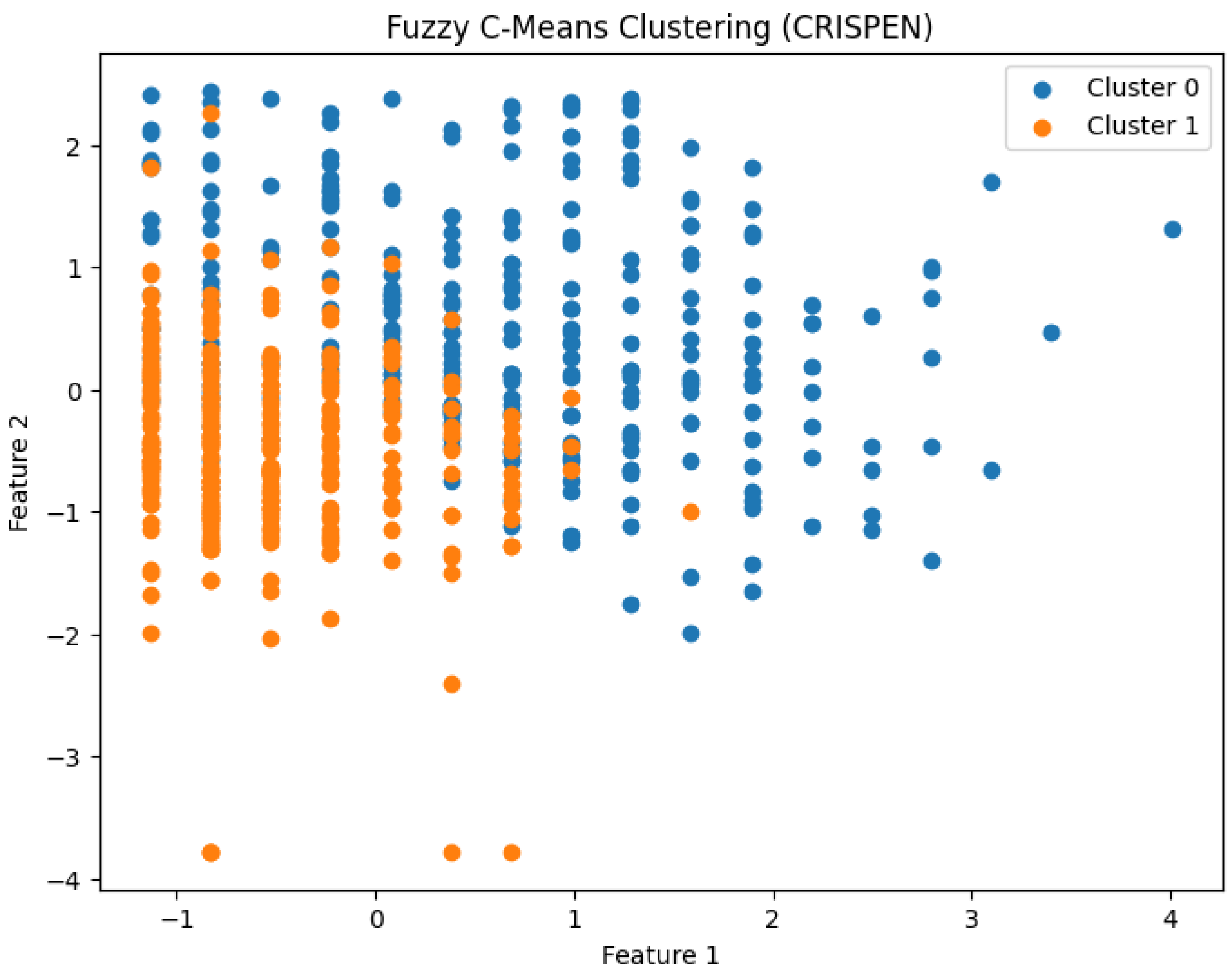
```
(2, 9)
```

```
# Compute sigma (spread) for each cluster
sigmas = []
for j in range(n_clusters):
    # membership weights for cluster j, raised to m
    u_j = u[j, :] ** m
    # weighted variance for each feature
    var_j = np.average((Xexp - centers[j])**2, axis=0, weights=u_j)
    sigma_j = np.sqrt(var_j)
    sigmas.append(sigma_j)
sigmas=np.array(sigmas)

# Hard clustering from fuzzy membership
cluster_labels = np.argmax(u, axis=0)
print("Fuzzy partition coefficient (FPC):", fpc)

# Plot first two features with fuzzy membership
plt.figure(figsize=(8,6))
for j in range(n_clusters):
    plt.scatter(
        Xexp[cluster_labels == j, 0],             # Feature 1
        Xexp[cluster_labels == j, 1],             # Feature 2
        alpha=u[j, :],          # transparency ~ membership
        label=f'Cluster {j}'
    )

plt.title("Fuzzy C-Means Clustering (with membership degree)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
```

```
Fuzzy partition coefficient (FPC): 0.5049376838853796
```

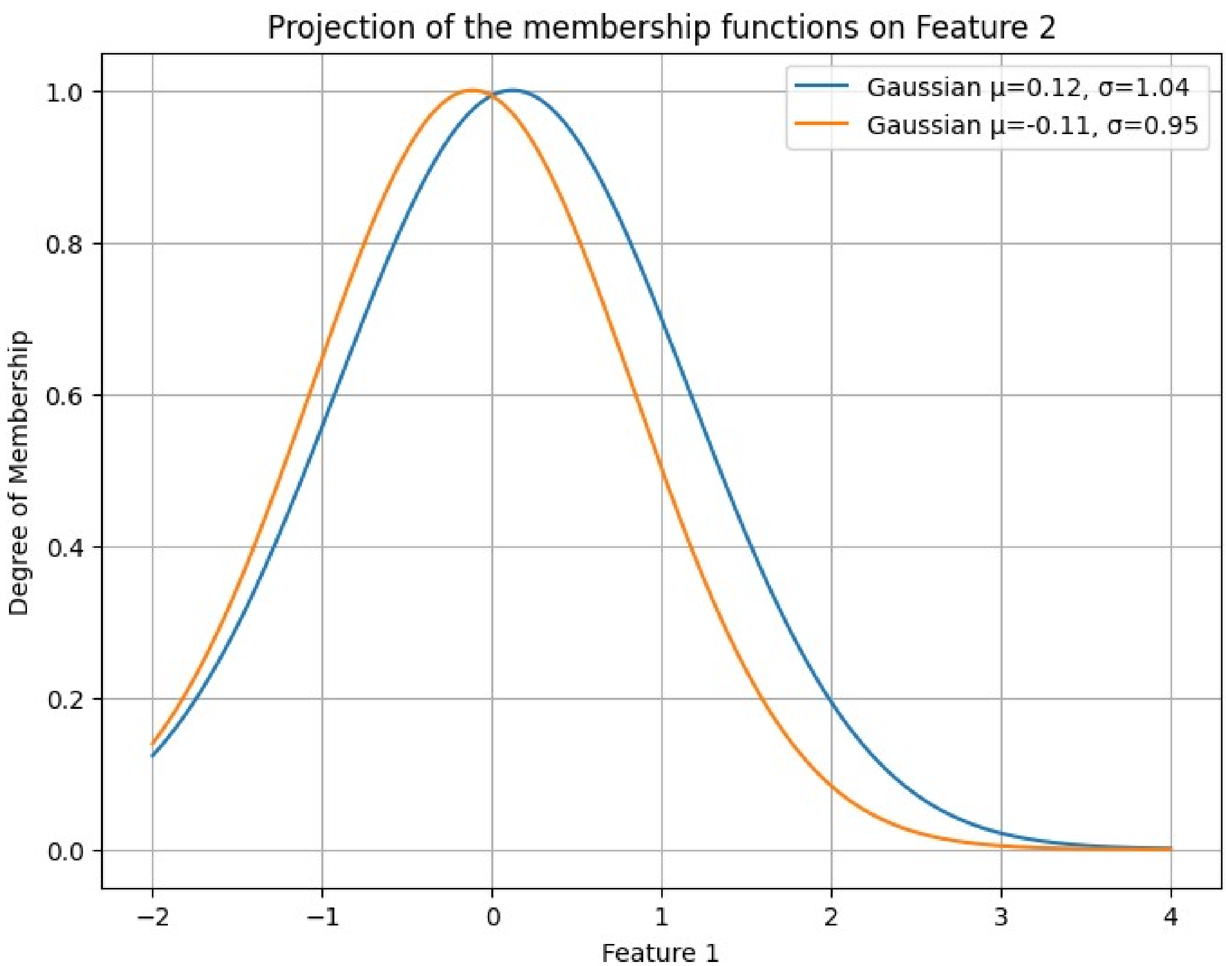



```
# Plot first two features with cluster assignments
plt.figure(figsize=(8,6))
for j in range(n_clusters):
    plt.scatter(
        Xexp[cluster_labels == j, 0],
        Xexp[cluster_labels == j, 1],
        label=f'Cluster {j}'
    )

plt.title("Fuzzy C-Means Clustering (CRISPEN)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
```




```
# Gaussian formula
def gaussian(x, mu, sigma):
    return np.exp(-0.5 * ((x - mu)/sigma)**2)

lin=np.linspace(-2, 4, 500)
plt.figure(figsize=(8,6))

y_aux=[]
feature=0
for j in range(n_clusters):
# Compute curves
    y_aux.append(gaussian(lin, centers[j,feature], sigmas[j,feature]))

# Plot
    plt.plot(lin, y_aux[j], label=f"Gaussian
μ={np.round(centers[j,feature],2)},
σ={np.round(sigmas[j,feature],2)}")

plt.title("Projection of the membership functions on Feature 2")
plt.xlabel("Feature 1")
```

```
plt.ylabel("Degree of Membership")
plt.legend()
plt.grid(True)
plt.show()
```




```
# ---------------------------
# Gaussian Membership Function
# ---------------------------
class GaussianMF(nn.Module):
    def __init__(self, centers, sigmas, agg_prob):
        super().__init__()
        self.centers = nn.Parameter(torch.tensor(centers,
dtype=torch.float32))
        self.sigmas = nn.Parameter(torch.tensor(sigmas,
dtype=torch.float32))
        self.agg_prob=agg_prob

    def forward(self, x):
        # Expand for broadcasting
```

```
        # x: (batch, 1, n_dims), centers: (1, n_rules, n_dims),
sigmas: (1, n_rules, n_dims)
        diff = abs((x.unsqueeze(1) -
self.centers.unsqueeze(0))/self.sigmas.unsqueeze(0)) #(batch, n_rules,
n_dims)

        # Aggregation
        if self.agg_prob:
            dist = torch.norm(diff, dim=-1)  # (batch, n_rules) #
probablistic intersection
        else:
            dist = torch.max(diff, dim=-1).values  # (batch, n_rules)
# min intersection (min instersection of normal funtion is the same as
the max on dist)

        return torch.exp(-0.5 * dist ** 2)


# -----------------------------
# TSK Model
# -----------------------------
class TSK(nn.Module):
    def __init__(self, n_inputs, n_rules, centers,
sigmas,agg_prob=False):
        super().__init__()
        self.n_inputs = n_inputs
        self.n_rules = n_rules

        # Antecedents (Gaussian MFs)

        self.mfs=GaussianMF(centers, sigmas,agg_prob)

        # Consequents (linear functions of inputs)
        # Each rule has coeffs for each input + bias
        self.consequents = nn.Parameter(
            torch.randn(n_inputs + 1,n_rules)
        )

    def forward(self, x):
        # x: (batch, n_inputs)
        batch_size = x.shape[0]

        # Compute membership values for each input feature
        # firing_strengths: (batch, n_rules)
        firing_strengths = self.mfs(x)

        # Normalize memberships
        # norm_fs: (batch, n_rules)
        norm_fs = firing_strengths / (firing_strengths.sum(dim=1,
keepdim=True) + 1e-9)
```

```
        # Consequent output (linear model per rule)
        x_aug = torch.cat([x, torch.ones(batch_size, 1)], dim=1)  #
add bias

        rule_outputs = torch.einsum("br,rk->bk", x_aug,
self.consequents)  # (batch, rules)
        # Weighted sum
        output = torch.sum(norm_fs * rule_outputs, dim=1,
keepdim=True)

        return output, norm_fs, rule_outputs


# ---------------------------
# Least Squares Solver for Consequents (TSK)
# ---------------------------
def train_ls(model, X, y):
    with torch.no_grad():
        _, norm_fs, _ = model(X)

        # Design matrix for LS: combine normalized firing strengths
with input
        X_aug = torch.cat([X, torch.ones(X.shape[0], 1)], dim=1)

        Phi = torch.einsum("br,bi->bri", X_aug,
norm_fs).reshape(X.shape[0], -1)

        # Solve LS: consequents = (Phi^T Phi)^-1 Phi^T y

        theta= torch.linalg.lstsq(Phi, y).solution


        model.consequents.data =
theta.reshape(model.consequents.shape)



# ---------------------------
# Gradient Descent Training
# ---------------------------
def train_gd(model, X, y, epochs=100, lr=1e-3):
    optimizer = optim.Adam(model.parameters(), lr=lr)
    criterion = nn.MSELoss()
    for _ in range(epochs):
        optimizer.zero_grad()
        y_pred, _, _ = model(X)
        loss = criterion(y_pred, y)
        #print(loss)
```

```
        loss.backward()
        optimizer.step()

# ---------------------------
# Hybrid Training (Classic ANFIS)
# ---------------------------
def train_hybrid_anfis(model, X, y, max_iters=10, gd_epochs=20, lr=1e-
3):
    train_ls(model, X, y)
    for _ in range(max_iters):
        # Step A: GD on antecedents (freeze consequents)
        model.consequents.requires_grad = False
        train_gd(model, X, y, epochs=gd_epochs, lr=lr)

        # Step B: LS on consequents (freeze antecedents)
        model.consequents.requires_grad = True
        model.mfs.requires_grad = False
        train_ls(model, X, y)

        # Re-enable antecedents
        model.mfs.requires_grad = True

# ---------------------------
# Alternative Hybrid Training (LS+ gradient descent on all)
# ---------------------------
def train_hybrid(model, X, y, epochs=100, lr=1e-4):
    # Step 1: LS for consequents
    train_ls(model, X, y)
    # Step 2: GD fine-tuning
    train_gd(model, X, y, epochs=epochs, lr=lr)

# Build model
model = TSK(n_inputs=Xtr.shape[1], n_rules=n_clusters,
centers=centers[:,:-1], sigmas=sigmas[:,:-1])

Xtr = torch.tensor(Xtr, dtype=torch.float32)
ytr = torch.tensor(ytr, dtype=torch.float32).reshape(-1,1)
Xte = torch.tensor(Xte, dtype=torch.float32)
yte = torch.tensor(yte, dtype=torch.float32).reshape(-1,1)
```

```
C:\Users\Murtaghy\AppData\Local\Temp\ipykernel_15480\1902815775.py:5:
UserWarning: To copy construct from a tensor, it is recommended to use
sourceTensor.detach().clone() or
sourceTensor.detach().clone().requires_grad_(True), rather than
torch.tensor(sourceTensor).
  ytr = torch.tensor(ytr, dtype=torch.float32).reshape(-1,1)
C:\Users\Murtaghy\AppData\Local\Temp\ipykernel_15480\1902815775.py:7:
UserWarning: To copy construct from a tensor, it is recommended to use
sourceTensor.detach().clone() or
sourceTensor.detach().clone().requires_grad_(True), rather than
```

```
torch.tensor(sourceTensor).
  yte = torch.tensor(yte, dtype=torch.float32).reshape(-1,1)

# Training with LS:
train_hybrid_anfis(model, Xtr, ytr, max_iters=10, gd_epochs=20, lr=1e-
3)

y_pred, _, _=model(Xte)
#performance metric for classification
print(f'ACC:
{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.5)}')
#classification
#performance metric for regression
#print(f'MSE:
{mean_squared_error(yte.detach().numpy(),y_pred.detach().numpy())}')
#regression

ACC:0.7402597402597403
```

Redes Neuronais - Classificação

```
import numpy as np
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import
mean_squared_error,accuracy_score,classification_report
import matplotlib.pyplot as plt
import torch.nn.functional as F
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
import pandas
```

This section loads and prepares the dataset for the model:

Dataset selection: The commented line offers a regression example (load_diabetes), while the active code uses a classification dataset (diabetes from OpenML).

Feature extraction: X contains the input features as a NumPy array.

Target extraction: y contains the raw labels.

Binary encoding: For classification tasks, labels are converted into binary format (0 = negative, 1 = positive) to suit supervised learning.

Tensor conversion: Labels are converted to PyTorch tensors with shape (n_samples, 1), ensuring compatibility with the neural network.

```
# CHOOSE DATASET

# Regression dataset
#data = datasets.load_diabetes(as_frame=True)

# Classification dataset
data = datasets.fetch_openml("diabetes",version=1, as_frame=True)

X = data.data.values
y = data.target.values

# Converter labels em binário (0 = negativo, 1 = positivo) (só usado
em classification)
y= np.array([1 if val == "tested_positive" else 0 for val in y])

# Converter para tensor PyTorch (coluna) ( só usado em classification)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)


X.shape
```

```
(768, 8)
```

```
#train test spliting
test_size=0.2
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=test_size,
random_state=42)

# Standardize features
scaler=StandardScaler()
Xtr= scaler.fit_transform(Xtr)
Xte= scaler.transform(Xte)

class MLP(nn.Module):
    def __init__(self, input_size, output_size=1, dropout_prob=0.5):
        super(MLP, self).__init__()

        self.fc1 = nn.Linear(input_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 64)
        self.fc4 = nn.Linear(64, 64)
        self.out = nn.Linear(64, output_size)

        self.dropout = nn.Dropout(p=dropout_prob)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)

        x = F.relu(self.fc2(x))
        x = self.dropout(x)
```

```
        x = F.relu(self.fc3(x))
        x = self.dropout(x)

        x = F.relu(self.fc4(x))
        x = self.dropout(x)

        x = self.out(x)
        return x

num_epochs=100
lr=0.00025
dropout=0.1
batch_size=64

Xtr = torch.tensor(Xtr, dtype=torch.float32)
ytr = torch.tensor(ytr, dtype=torch.float32)
Xte = torch.tensor(Xte, dtype=torch.float32)
yte = torch.tensor(yte, dtype=torch.float32)

# Wrap Xtr and ytr into a dataset
train_dataset = TensorDataset(Xtr, ytr)

# Create DataLoader
train_dataloader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
```

```
C:\Users\Murtaghy\AppData\Local\Temp\ipykernel_17000\3341141645.py:2:
UserWarning: To copy construct from a tensor, it is recommended to use
sourceTensor.detach().clone() or
sourceTensor.detach().clone().requires_grad_(True), rather than
torch.tensor(sourceTensor).
  ytr = torch.tensor(ytr, dtype=torch.float32)
C:\Users\Murtaghy\AppData\Local\Temp\ipykernel_17000\3341141645.py:4:
UserWarning: To copy construct from a tensor, it is recommended to use
sourceTensor.detach().clone() or
sourceTensor.detach().clone().requires_grad_(True), rather than
torch.tensor(sourceTensor).
  yte = torch.tensor(yte, dtype=torch.float32)
```

```
# Model, Loss, Optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = MLP(input_size=Xtr.shape[1], dropout_prob=dropout).to(device)
criterion = nn.BCEWithLogitsLoss()  # for binary classification
criterion = nn.MSELoss() #for regression
optimizer = optim.Adam(model.parameters(), lr=lr)

# Training loop
for epoch in range(num_epochs):
    model.train()
```

```python
    epoch_loss = 0.0

    for batch_x, batch_y in train_dataloader:
        batch_x = batch_x.to(device)
        batch_y = batch_y.to(device)

        logits = model(batch_x)
        loss = criterion(logits, batch_y.view(-1, 1))

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()

    avg_loss = epoch_loss / len(train_dataloader)
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {avg_loss:.4f}")
```

```
Epoch [1/100], Loss: 0.2716
Epoch [2/100], Loss: 0.2480
Epoch [3/100], Loss: 0.2309
Epoch [4/100], Loss: 0.2135
Epoch [5/100], Loss: 0.2027
Epoch [6/100], Loss: 0.1948
Epoch [7/100], Loss: 0.1877
Epoch [8/100], Loss: 0.1765
Epoch [9/100], Loss: 0.1717
Epoch [10/100], Loss: 0.1659
Epoch [11/100], Loss: 0.1676
Epoch [12/100], Loss: 0.1665
Epoch [13/100], Loss: 0.1620
Epoch [14/100], Loss: 0.1599
Epoch [15/100], Loss: 0.1559
Epoch [16/100], Loss: 0.1575
Epoch [17/100], Loss: 0.1509
Epoch [18/100], Loss: 0.1586
Epoch [19/100], Loss: 0.1562
Epoch [20/100], Loss: 0.1522
Epoch [21/100], Loss: 0.1603
Epoch [22/100], Loss: 0.1532
Epoch [23/100], Loss: 0.1576
Epoch [24/100], Loss: 0.1461
Epoch [25/100], Loss: 0.1510
Epoch [26/100], Loss: 0.1503
Epoch [27/100], Loss: 0.1491
Epoch [28/100], Loss: 0.1480
Epoch [29/100], Loss: 0.1485
Epoch [30/100], Loss: 0.1470
Epoch [31/100], Loss: 0.1454
Epoch [32/100], Loss: 0.1447
```

```
Epoch [33/100], Loss: 0.1517
Epoch [34/100], Loss: 0.1474
Epoch [35/100], Loss: 0.1474
Epoch [36/100], Loss: 0.1424
Epoch [37/100], Loss: 0.1449
Epoch [38/100], Loss: 0.1463
Epoch [39/100], Loss: 0.1415
Epoch [40/100], Loss: 0.1451
Epoch [41/100], Loss: 0.1446
Epoch [42/100], Loss: 0.1374
Epoch [43/100], Loss: 0.1425
Epoch [44/100], Loss: 0.1435
Epoch [45/100], Loss: 0.1452
Epoch [46/100], Loss: 0.1421
Epoch [47/100], Loss: 0.1374
Epoch [48/100], Loss: 0.1394
Epoch [49/100], Loss: 0.1401
Epoch [50/100], Loss: 0.1366
Epoch [51/100], Loss: 0.1392
Epoch [52/100], Loss: 0.1333
Epoch [53/100], Loss: 0.1425
Epoch [54/100], Loss: 0.1392
Epoch [55/100], Loss: 0.1408
Epoch [56/100], Loss: 0.1363
Epoch [57/100], Loss: 0.1356
Epoch [58/100], Loss: 0.1311
Epoch [59/100], Loss: 0.1407
Epoch [60/100], Loss: 0.1353
Epoch [61/100], Loss: 0.1369
Epoch [62/100], Loss: 0.1342
Epoch [63/100], Loss: 0.1321
Epoch [64/100], Loss: 0.1388
Epoch [65/100], Loss: 0.1331
Epoch [66/100], Loss: 0.1270
Epoch [67/100], Loss: 0.1354
Epoch [68/100], Loss: 0.1305
Epoch [69/100], Loss: 0.1311
Epoch [70/100], Loss: 0.1300
Epoch [71/100], Loss: 0.1393
Epoch [72/100], Loss: 0.1280
Epoch [73/100], Loss: 0.1321
Epoch [74/100], Loss: 0.1247
Epoch [75/100], Loss: 0.1295
Epoch [76/100], Loss: 0.1266
Epoch [77/100], Loss: 0.1310
Epoch [78/100], Loss: 0.1285
Epoch [79/100], Loss: 0.1258
Epoch [80/100], Loss: 0.1234
Epoch [81/100], Loss: 0.1267
```

```
Epoch [82/100], Loss: 0.1256
Epoch [83/100], Loss: 0.1265
Epoch [84/100], Loss: 0.1329
Epoch [85/100], Loss: 0.1294
Epoch [86/100], Loss: 0.1254
Epoch [87/100], Loss: 0.1277
Epoch [88/100], Loss: 0.1247
Epoch [89/100], Loss: 0.1250
Epoch [90/100], Loss: 0.1257
Epoch [91/100], Loss: 0.1237
Epoch [92/100], Loss: 0.1197
Epoch [93/100], Loss: 0.1231
Epoch [94/100], Loss: 0.1203
Epoch [95/100], Loss: 0.1194
Epoch [96/100], Loss: 0.1234
Epoch [97/100], Loss: 0.1226
Epoch [98/100], Loss: 0.1161
Epoch [99/100], Loss: 0.1196
Epoch [100/100], Loss: 0.1181
```

```
y_pred=model(Xte)
print(f'ACC:
{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.5)}')
#classification
#print(f'MSE:
{mean_squared_error(yte.detach().numpy(),y_pred.detach().numpy())}')
#regression
```

```
ACC:0.7337662337662337
```