ML 2024 Project Presentation

Giuseppe Gabriele Russo Mario Mogavero

Team Rocket
Master degree in Artificial Intelligence

g.russo55@studenti.unipi.it
m.mogavero1@studenti.unipi.it

Project B

29 Gennaio 2025

Objectives

- Develop and compare neural networks using Keras, PyTorch, and Scikit-learn.
- Explore SVMs and KNNs with Scikit-learn for additional comparison.
- Focus on analyzing performance and deepening expertise in neural networks.

Neural networks' Monks

Fully connected networks with the same number of hidden nodes in each layer tested with 4 activation functions: **tanh, linear, ReLU, Leaky ReLU**.

Possibility of using **Tikhonov regularization** and **Nesterov momentum**.

Implemented **early stopping** to prevent overfitting.

Output layer with **sigmoid activation**.

SVM and KNN

- KNN implemented with **different distance metrics** and **varying values of K** (the number of nearest neighbors considered).
- SVM implemented with **different values of C** (controlling the trade-off between maximizing the margin and minimizing classification error).

Neural Networks' CUP

For the regression task, we used a similar fully connected architecture with the same number of hidden nodes, but with some key differences:

- early stopping now includes tolerance;
- the output layer was changed to a linear activation function to suit the regression task.

Loss and learning algorithm

For classification, Binary Cross-Entropy (**BCE**) loss was used, while for regression, Mean Squared Error (**MSE**) loss was applied.
The learning algorithm employs Stochastic Gradient Descent (**SGD**) with **minibatch**, and Adam was also tested.

Validation schema

Greedy search with k-fold cross-validation was performed to optimize parameters ($K = 5$).

After obtaining the best parameters, **nested k-fold cross-validation** was used to estimate the risk ($inner = 5, outer = 5$)

Finally, model performance was assessed on the test file.

Preprocessing data

Classification Problems

- Applied one-hot encoding to input data.

CUP

- Input data standardized with Standard Scaler.
- Labels normalized with Min-Max Scaler.

Monk 1

Among the Monk1, Monk2, and Monk3 problems, **Monk1** is considered the simplest, as it has no noise in the training data and the decision patterns are clear. This makes it a good starting point for comparing the performance of different frameworks and algorithms.

The Tables 1, 2 summarize the results obtained for each experiment. Instead, the Graphs 3, 4, 5, allow you to visualize the differences in performance

Model	Hyperparameters	TR / TS loss	TR / TS accuracy	Accuracy variance
Scikit-learn	$\eta = 0.3$, epochs= 340, batch_size= 64, hidden_size= 3, hidden_layers= 1, $\alpha = 0.8$, patience= 30	0.017 / 0.001	1.0 / 0.98	0.0004
PyTorch	$\eta = 0.9$, epochs= 180, batch_size= 9, hidden_size= 3, hidden_layers= 1, $\alpha = 0.4$, patience= 30	0.012 / 0.004	1.0 / 1.0	0.0
Keras	$\eta = 0.4$, epochs= 180, batch_size= 4, hidden_size= 3, hidden_layers= 1, $\alpha = 0.4$, patience= 30	0.007 / 0.017	1.0 / 0.95	0.003

Figura 1: Average prediction results obtained for the MONK 1

Model	Hyperparameters	TR / TS accuracy	Accuracy variance
SVM	C = 31, type = poly	1.0 / 1.0	0.0006
K-nn	K = 8, P = 5, type = manhattan	1.0 / 0.78	0.006

Figura 2: Results SVM, K-nn Monk 1

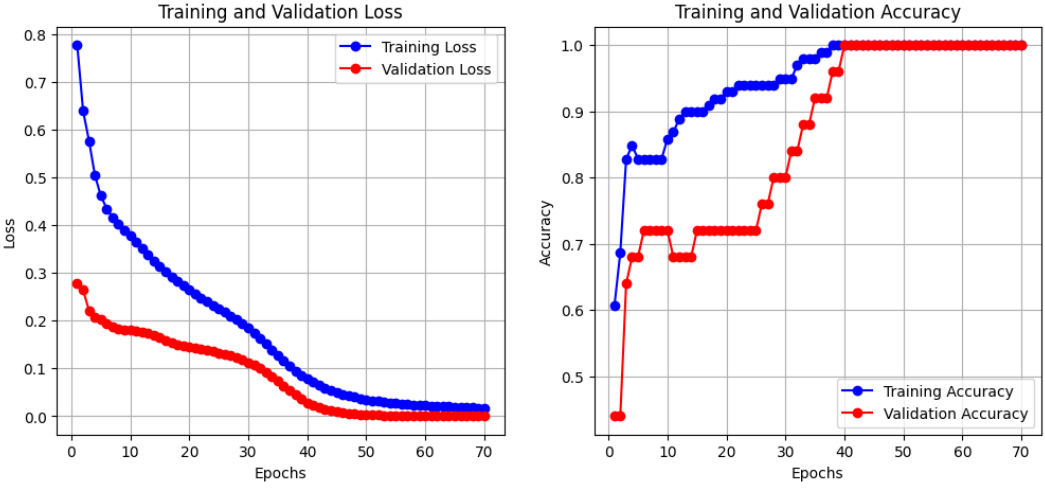

Figura 3: Plot of loss and accuracy Scikit-learn on Monk 1

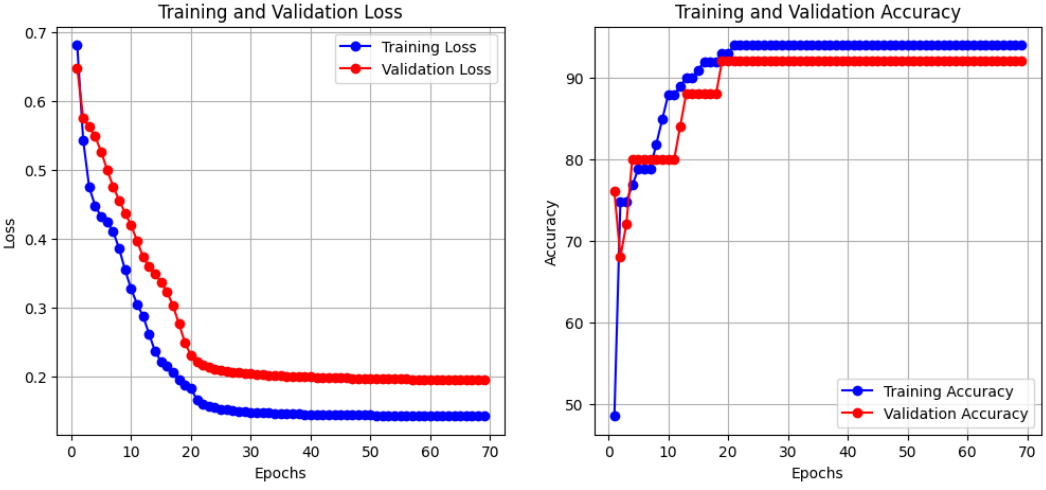

Figura 4: Plot of loss and accuracy Pytorch on Monk 1

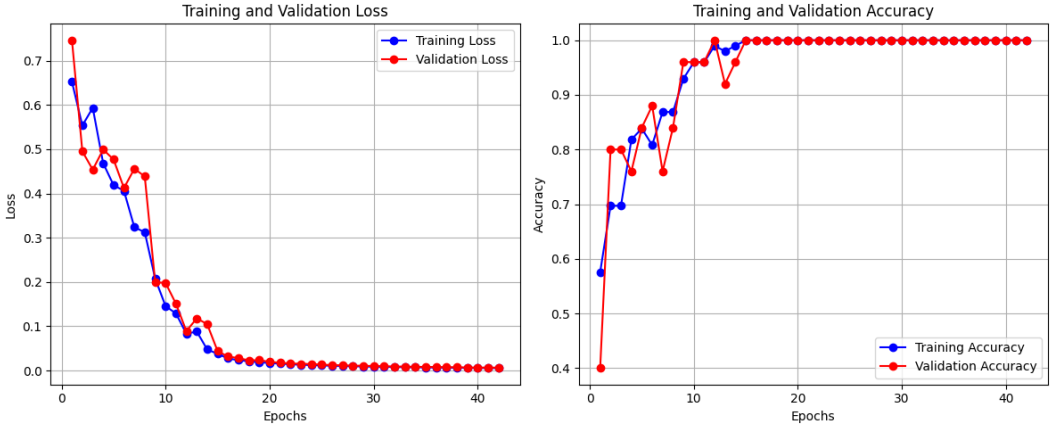

Figura 5: Plot of loss and accuracy Keras on Monk 1

Monk 2

Monk2 is considered more complex than Monk1 due to the presence of noise in the training data. As a result, Monk2 tests the robustness of different frameworks, requiring them to handle this noise and still deliver accurate predictions.

The Tables 6, 7 summarize the results obtained for each experiment. Instead, the Graphs 8, 9, 10, allow you to visualize the differences in performance

Model	Hyperparameters	TR / TS loss	TR / TS accuracy	Accuracy variance
Scikit-learn	$\eta = 0.9$, epochs= 180, batch_size= 16, hidden_size= 3, hidden_layers= 1, $\alpha = 0.6$, patience= 30	0.005 / 0.001	1.0 / 1.0	0.0
PyTorch	$\eta = 0.2$, epochs= 180, batch_size= 10, hidden_size= 3, hidden_layers= 1, $\alpha = 0.6$, patience= 30	0.002 / 0.002	1.0 / 1.0	0.0
Keras	$\eta = 0.4$, epochs= 180, batch_size= 16, hidden_size= 3, hidden_layers= 1, $\alpha = 0.9$, patience= 30	0.002 / 0.002	1.0 / 1.0	0.0

Figura 6: Average prediction results obtained for the MONK 2

Model	Hyperparameters	TR / TS accuracy	Accuracy variance
SVM	C = 31, type = poly	1.0 / 0.77	0.001
K-nn	K = 26, P = 10, type = euclidean	1.0/ 0.65	0.001

Figura 7: Results SVM, K-nn Monk 2

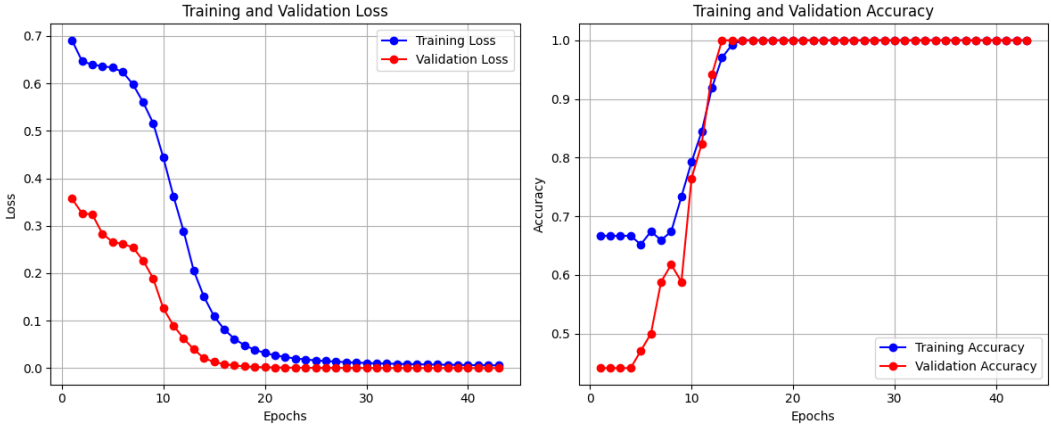

Figura 8: Plot of loss and accuracy Scikit-learn on Monk 2

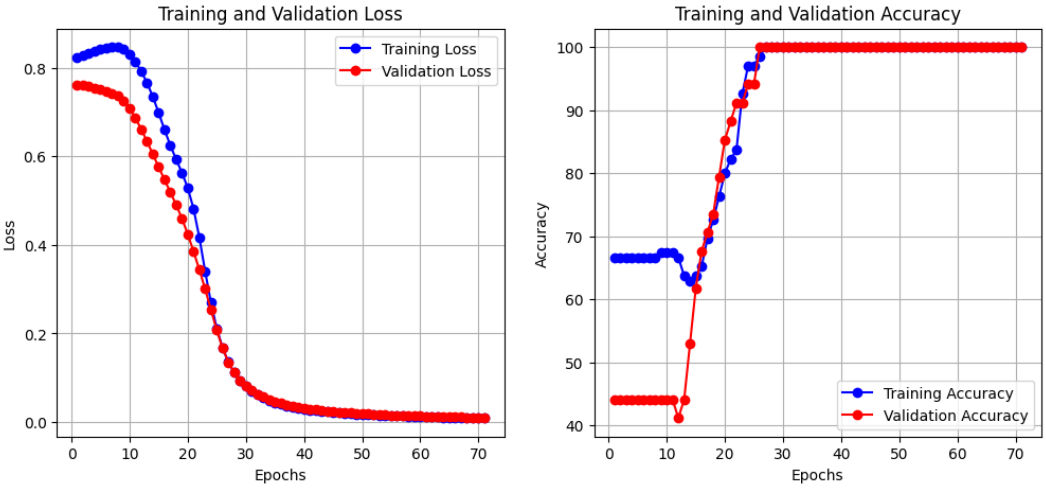

Figura 9: Plot of loss and accuracy Pytorch on Monk 2

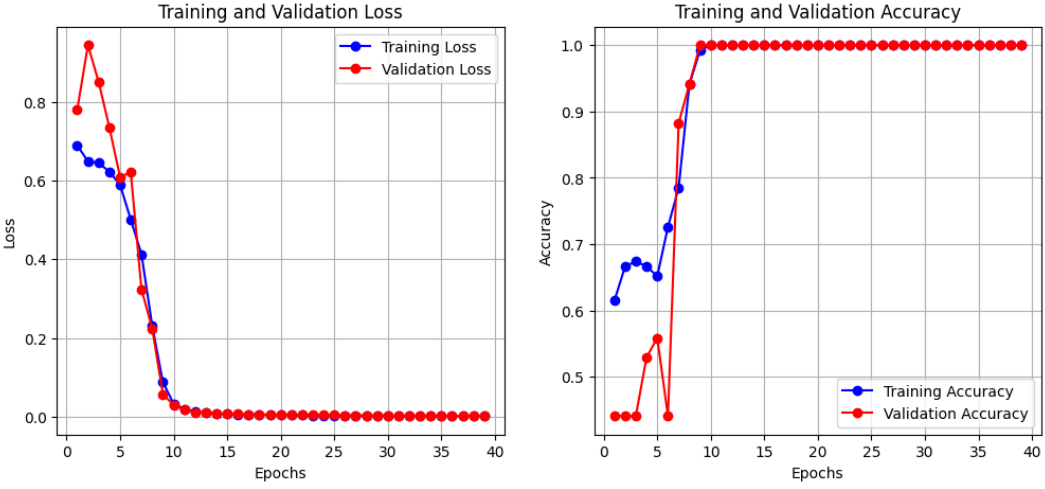

Figura 10: Plot of loss and accuracy Keras on Monk 2

Monk 3

Monk3 is the most difficult due to the significant noise in the training data. Consequently, Monk3 pushes frameworks to their limits, as they must deal with a high degree of ambiguity while striving to maintain performance and accuracy.

The Tables 11, 12 summarize the results obtained for each experiment. Instead, the Graphs 13, 14, 15, allow you to visualize the differences in performance

Model	Hyperparameters	TR / TS loss	TR / TS accuracy	Accuracy variance
Scikit-learn	$\eta = 0.1$, epochs= 180, batch_size= 16, hidden_size= 3, hidden_layers= 1, $\alpha = 0.3$, patience= 30, $\lambda = 0.1$	0.277 / 0.08	0.93 / 0.92	0.003
PyTorch	$\eta = 0.2$, epochs= 180, batch_size= 9, hidden_size= 3, hidden_layers= 1, $\alpha = 0.4$, patience= 30, $\lambda = 0.001$	0.374 / 0.4	0.94 / 0.96	0.239
Keras	$\eta = 0.1$, epochs= 180, batch_size= 16, hidden_size= 3, hidden_layers= 1, $\alpha = 0.6$, patience= 30, $\lambda = 0.001$	0.184 / 0.330	0.94 / 0.92	0.002

Figura 11: Average prediction results obtained for the MONK 3

Model	Hyperparameters	TR / TS accuracy	Accuracy variance
SVM	C = 31, type = linear	0.93 / 0.97	0.002
K-nn	K = 57, P = 3 type = minkowski	1.0 / 0.92	0.003

Figura 12: Results SVM, K-nn Monk 3

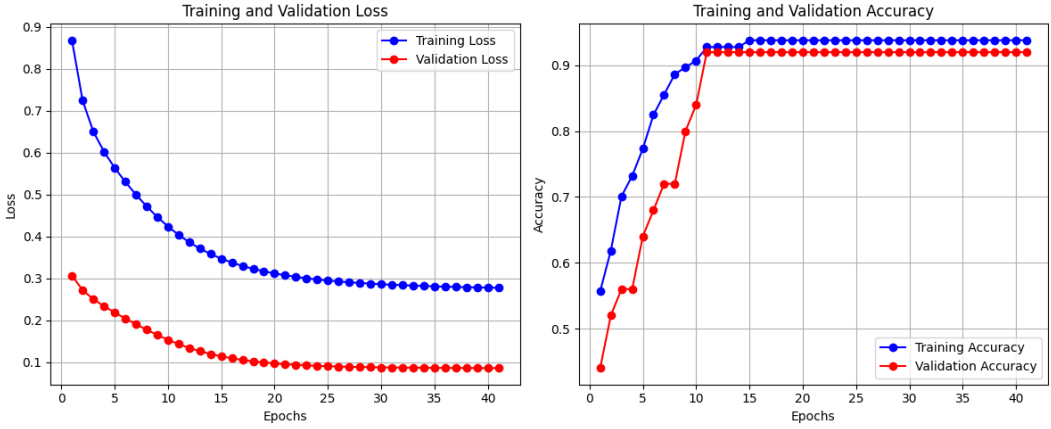

Figura 13: Plot of loss and accuracy Scikit-learn on Monk 3

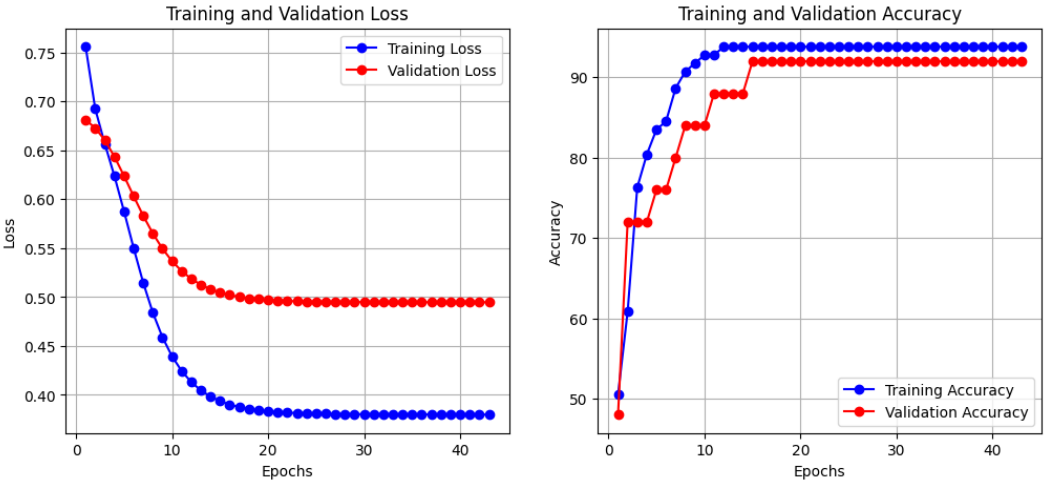

Figura 14: Plot of loss and accuracy Pytorch on Monk 3

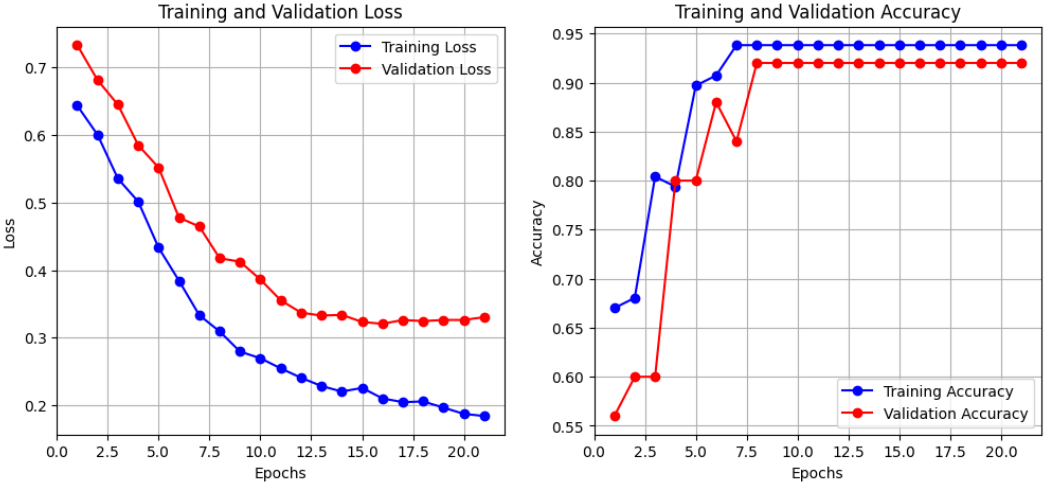

Figura 15: Plot of loss and accuracy Keras on Monk 3

CUP setup

We assume early stopping tolerance on validation loss of 0.05, and the greedy search was executed with the range reported in figure 16. All the models are tested with the activation functions introduced before. Time estimated: 90 min.

η	λ	α	Hidden size	Hidden layers	Patience	Epochs	Batch size
[0.1 - 0.9]	[0.0001 - 0.1]	[0.1 - 0.9]	[10 - 80]	[1 -3]	[15 - 30]	[180 - 700]	[4 - 128]

Figura 16: Range of parameters for greedy search

CUP validation schema

The validation schema is the following:

For all models, greedy search with k-fold cross-validation was performed to optimize parameters, and after a nested k-fold cross-validation was used to estimate the risk. Finally, the model performance was assessed on the test file.

CUP results

Model	TR MEE	VL MEE	TS MEE	Variance MEE
Scikit-learn	0.07235	0.085827	0.071308	0.000213
Pytorch	0.020309	0.029251	0.038588	3.77226 e-06
Keras	0.115760	0.145479	0.088829	4.777090 e-05

Figura 17: Average prediction results obtained for the CUP

Best model for CUP

On the basis of the results obtained on validation considering the variance, we chose **PyTorch** as the best model.

Model	Pytorch
Hyperparameters	$\eta = 0.01$, epochs= 700, batch_size= 128, hidden_size= 80, hidden_layers= 2, $\alpha = 0.8$, patience= 30, $\lambda = 0.01$
TR MEE	0.02030986314379303
VL MEE	0.02925143790967537
TS MEE	0.038588407039642336

Figura 18: MEE result for the best model

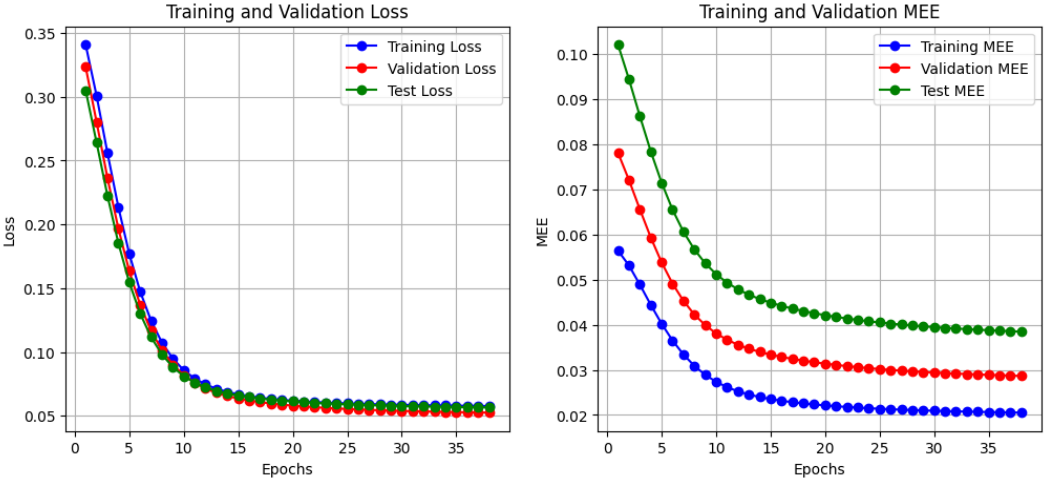

Figura 19: MSE and MEE for the best model on the CUP

Discussion: Performance

- **Scalability and Speed:** PyTorch $\approx$ Keras $>$ Scikit-learn.
- **Flexibility:** PyTorch $>$ Keras $>$ Scikit-learn.

Our studies on Monk datasets have shown that PyTorch consistently outperforms Keras and Scikit implementations in both accuracy and convergence time.

Discussion: Ease of Use Comparison

Keras: Simplicity and High-Level Abstractions

- Straightforward building: layers are added sequentially. All configurations are specified in model compilation.
- Training and validation are handled with a single function call: `model.fit()`.

PyTorch: Flexibility and Explicit Control

- NN are defined as a Python class, with `__init__()` for layers and `forward()` for forward propagation.
- Requires the manual implementation of a `fit()` function for training and validation.

Scikit-learn: Manual Implementation

- Training is done with straightforward methods like `fit()`, but lacks built-in GPU acceleration or tensor-based computations.
- Designed for classic machine learning workflows, not optimized for deep learning tasks or large-scale datasets.

SVM, K-nn: Simplicity but...

- **SVM** is effective in high-dimensional spaces and suitable for datasets with clear margin separation like Monk 1 but not for others.
- **K-nn** is highly dependent on the choice of distance metric and the value of `K`.

Conclusion

- Frameworks simplify NN implementation, but with a tradeoff between ease of use and flexibility.
- Problems like MONK datasets are sensitive to initial weights, and more flexibility leads to better performance optimization.
- Greater flexibility requires more code and time, but enhances training optimization.
- Scikit−learn's preprocessing libraries are convenient and well-integrated with all frameworks.

File name : `team_rocket_ML-CUP24-TS.csv`

Team name: Team Rocket

Appendix

Introducing a New Comparison

We implemented a neural network for the Monk problems manually using only `numpy`.

The goal is to highlight the differences in complexity and the time required to implement and train neural networks when using frameworks versus building them from scratch.

This manual implementation serves as a baseline!

Forward Propagation Implementation

```
# Forward propagation
input_data[count] = X[i]

# Copy previous weight updates
if any(x is not None for x in dW):
    dW_old = copy.deepcopy(dW)

# Anticipate momentum-based updates for weights
if anticipate:
    anticipate_weights = copy.deepcopy(self.weights)
    anticipate = False
    for i in range(self.hidden_layers + 1):
        if i == self.hidden_layers:
            anticipate_weights[i][0] = self.weights[i][0] + self.momentum * dW_old[i][0]
            anticipate_weights[i][1] = self.weights[i][1] + self.momentum * dW_old[i][1]
        else:
            anticipate_weights[i][0] = self.weights[i][0] + self.momentum * dW_old[i][0]
            sumdb = np.sum(dW_old[i][1], axis=0)
            anticipate_weights[i][1] = self.weights[i][1] + self.momentum * sumdb

# Forward propagation through the hidden layers
for j in range(self.hidden_layers):
    if j == 0:  # Input to first hidden layer
        a = self.relu(
            np.dot(anticipate_weights[j][0], X[i][:, np.newaxis]).T
            + anticipate_weights[j][1]
        )
        atot[j][count] = a
    else:  # Hidden layer to hidden layer
        a = self.relu(
            np.dot(atot[j - 1][count], anticipate_weights[j][0])
            + anticipate_weights[j][1]
            )
        atot[j][count] = a
```

```
# Output layer
z1 = self.sigmoid(
    np.dot(
        anticipate_weights[self.hidden_layers][0],
        atot[self.hidden_layers - 1][count].T,
    )
    + anticipate_weights[self.hidden_layers][1]
)[0]
z1 = np.clip(z1, 1e-15, 1 - 1e-15)  # Avoid numerical instability
output[count] = z1
```

Back Propagation Implementation

```
# Backpropagation to compute gradients
for i in range(self.hidden_layers, -1, -1):
    if i == self.hidden_layers:  # Output layer
        dZ = loss * self.sigmoid_derivate(output)
        dw = self.GradientProductOutput(dZ, atot[i - 1])
        db = np.sum(dZ)
        dW[i] = (dw, db, dZ)

    elif i == (self.hidden_layers - 1):  # Last hidden layer
        if self.hidden_layers != 1:
            dA = self.error_hidden(anticipate_weights[i + 1][0], dW[i + 1][2])
            dZ = dA * self.relu_derivative(atot[i])
            dw = self.GradientProductHidden(dZ, atot[i - 1])
            db = dZ
            dW[i] = (dw, db, dZ)
        else:
            dA = self.error_hidden(anticipate_weights[i + 1][0], dW[i + 1][2])
            dZ = dA * self.relu_derivative(atot[i])
            dw = self.GradientProductInput(dZ, input_data)
            db = dZ
            dW[i] = (dw, db, dZ)

    elif i == 0:  # Input to first hidden layer
        dA = self.error_hidden_matrix(
            anticipate_weights[i + 1][0], dW[i + 1][2]
        )
        dZ = dA * self.relu_derivative(atot[i])
        dw = self.GradientProductInput(dZ, input_data)
        db = dZ
        dW[i] = (dw, db, dZ)

    else:  # Hidden layer to hidden layer
        dA = self.error_hidden_matrix(
            anticipate_weights[i + 1][0], dW[i + 1][2]
        )
        dZ = dA * self.relu_derivative(atot[i])
        dw = self.GradientProductHidden(dZ, atot[i - 1])
        db = dZ
        dW[i] = (dw, db, dZ)
```

Update weights Implementation

```
# Update weights using gradients, momentum, and regularization
for i in range(self.hidden_layers + 1):
    tikhonov_weight = self.lamb * self.weights[i][0]  # Regularization for weights
    tikhonov_bias = self.lamb * self.weights[i][1]  # Regularization for biases
    mom_weight = self.momentum * dW_old[i][0]
    mom_bias = self.momentum * dW_old[i][1]
    if i == self.hidden_layers:  # Output layer
        self.weights[i][0] = self.weights[i][0] + self.learning_rate * dW[i][0] - tikhonov_weight + mom_weight
        self.weights[i][1] = self.weights[i][1] + self.learning_rate * dW[i][1] - tikhonov_bias + mom_bias
    else:  # Hidden layers
        self.weights[i][0] = self.weights[i][0] + self.learning_rate * dW[i][0] - tikhonov_weight + mom_weight
        sumdb = np.sum(dW[i][1], axis=0)
        sumdbmom = np.sum(dW_old[i][1], axis=0)
        mom_bias = self.momentum * sumdbmom
        self.weights[i][1] = self.weights[i][1] + self.learning_rate * sumdb - tikhonov_bias + mom_bias
```