Hyperparameter Tuning Cookbook

A guide for scikit-learn, PyTorch, river, and spotpython

Thomas Bartz-Beielstein

Feb 12, 2025

Table of contents

Preface **1**
Book Structure . . . 1
Software Used in this Book . . . 2
Citation . . . 2

I. Optimization **5**

1. Introduction: Optimization **7**
1.1. Optimization, Simulation, and Surrogate Modeling . . . 7
1.2. Surrogates . . . 7
1.2.1. Costs of Simulation . . . 8
1.2.2. Mathematical Models and Meta-Models . . . 8
1.2.3. Surrogates = Trained Meta-models . . . 8
1.2.4. Computer Experiments . . . 8
1.2.5. Limits of Mathematical Modeling . . . 9
1.2.6. Example: Why Computer Simulations are Necessary . . . 9
1.2.7. Simulation Requirements . . . 9
1.3. Jupyter Notebook . . . 10

2. Aircraft Wing Weight Example **11**
2.1. AWWE Equation . . . 11
2.2. AWWE Parameters and Equations (Part 1) . . . 11
2.3. Goals: Understanding and Optimization . . . 12
2.4. Properties of the Python "Solver" . . . 13
2.5. Plot 1: Load Factor (N_z) and Aspect Ratio (A) . . . 14
2.6. Plot 2: Taper Ratio and Fuel Weight . . . 17
2.7. The Big Picture: Combining all Variables . . . 18
2.8. AWWE Landscape . . . 21
2.9. Summary of the First Experiments . . . 22
2.10. Exercise . . . 22
2.10.1. Adding Paint Weight . . . 22
2.11. Jupyter Notebook . . . 23

3. Introduction to `scipy.optimize` **25**
3.1. Derivative-free Optimization Algorithms . . . 26
3.1.1. Nelder-Mead Simplex Algorithm . . . 26

3.1.2. Powell's Method . . . 27
3.2. Gradient-based Optimization Algorithms . . . 29
3.2.1. An Introductory Example: Broyden-Fletcher-Goldfarb-Shanno Algorithm (BFGS) . . . 29
3.2.2. Background and Basics for Gradient-based Optimization . . . 30
3.2.3. Gradient . . . 30
3.2.4. Jacobian Matrix . . . 30
3.2.5. Hessian Matrix . . . 31
3.2.6. Gradient Descent . . . 32
3.2.7. Newton Method . . . 35
3.2.8. BFGS-Algorithm . . . 37
3.2.9. Procedure: . . . 37
3.2.10. Visualization BFGS for Rosenbrock . . . 39
3.3. Global Optimization . . . 39
3.3.1. Local vs Global Optimization . . . 39
3.3.2. Dual Annealing Optimization . . . 44
3.3.3. Differential Evolution . . . 48
3.3.4. Procedure . . . 49
3.3.5. Other global optimization algorithms . . . 51
3.3.6. DIRECT . . . 51
3.3.7. SHGO . . . 52
3.3.8. Basin-hopping . . . 52
3.4. Project: One-Mass Oscillator Optimization . . . 52
3.4.1. Introduction . . . 52
3.4.2. One-Mass Oscillator Model . . . 52
3.4.3. The Real-World Data . . . 53
3.4.4. Objective Function . . . 55
3.4.5. Results . . . 56
3.5. Exercises . . . 58
3.6. Jupyter Notebook . . . 59

4. Sequential Parameter Optimization: Using `scipy` Optimizers 61
4.1. The Objective Function Branin . . . 61
4.2. The Optimizer . . . 62
4.2.1. TensorBoard . . . 63
4.3. Print the Results . . . 64
4.4. Show the Progress . . . 65
4.5. Exercises . . . 66
4.5.1. `dual_annealing` . . . 66
4.5.2. `direct` . . . 67
4.5.3. `shgo` . . . 69
4.5.4. `basinhopping` . . . 71
4.5.5. Performance Comparison . . . 72
4.6. Jupyter Notebook . . . 73

II. Numerical Methods 75

5. Introduction: Numerical Methods 77
5.1. Response Surface Methods: What is RSM? . . . 77
5.1.1. Visualization: Problems in Practice . . . 80
5.1.2. RSM: Strategies . . . 80
5.1.3. RSM: Noise in the Empirical Model . . . 81
5.1.4. RSM: Natural and Coded Variables . . . 81
5.1.5. RSM Low-order Polynomials . . . 82
5.2. First-Order Models (Main Effects Model) . . . 82
5.2.1. First-Order Model Properties . . . 83
5.2.2. First-order Model with Interactions in python . . . 83
5.2.3. Observations: First-Order Model with Interactions . . . 85
5.3. Second-Order Models . . . 85
5.3.1. Second-Order Models: Properties . . . 86
5.3.2. Example: Stationary Ridge . . . 86
5.3.3. Observations: Second-Order Model (Ridge) . . . 87
5.3.4. Example: Rising Ridge . . . 88
5.3.5. Summary: Rising Ridge . . . 89
5.3.6. Falling Ridge . . . 89
5.3.7. Saddle Point . . . 89
5.3.8. Interpretation: Saddle Points . . . 91
5.3.9. Summary: Ridge Analysis . . . 91
5.4. General RSM Models . . . 91
5.4.1. Ordinary Least Squares . . . 91
5.5. General Linear Regression . . . 92
5.6. Designs . . . 93
5.6.1. Different Designs . . . 94
5.7. RSM Experimentation . . . 94
5.7.1. First Step . . . 94
5.7.2. Second Step . . . 94
5.7.3. Third Step . . . 94
5.8. RSM: Review and General Considerations . . . 95
5.8.1. Historical Considerations about RSM . . . 96
5.8.2. Status Quo . . . 96
5.8.3. The Role of Statistics . . . 96
5.8.4. New RSM is needed: DACE . . . 96
5.9. Exercises . . . 97
5.10. Jupyter Notebook . . . 97

6. Kriging (Gaussian Process Regression) 99
6.1. DACE and RSM . . . 99
6.1.1. Noise Handling in RSM and DACE . . . 100
6.2. Background: Expectation, Mean, Standard Deviation . . . 100
6.2.1. Calculation of the Standard Deviation with Python . . . 103

6.2.2. The Argument "axis" . . . 103
6.3. Data Types and Precision in Python . . . 104
6.4. Distributions and Random Numbers in Python . . . 105
6.4.1. The Uniform Distribution . . . 105
6.4.2. The Normal Distribution . . . 107
6.4.3. Visualization of the Standard Deviation . . . 109
6.4.4. Standardization of Random Variables . . . 109
6.4.5. Realizations of a Normal Distribution . . . 110
6.4.6. The Multivariate Normal Distribution . . . 110
6.5. Covariance . . . 113
6.6. Correlation . . . 115
6.6.1. Definitions . . . 115
6.6.2. Computations . . . 116
6.6.3. The Outer-product and the `np.outer` Function . . . 118
6.6.4. Correlation and Independence . . . 120
6.7. R-Squared in Simple Linear Regression . . . 122
6.8. Cholesky Decomposition and Positive Definite Matrices . . . 123
6.9. Maximum Likelihood Estimation: Multivariate Normal Distribution . . . 126
6.9.1. The Joint Probability Density Function of the Multivariate Normal Distribution . . . 126
6.9.2. The Log-Likelihood Function . . . 126
6.10. Constructing a Surrogate . . . 127
6.10.1. Stage One: Preparing the Data and Choosing a Modelling Approach . . . 128
6.10.2. Stage Two: Parameter Estimation and Training . . . 129
6.10.3. Stage Three: Model Testing . . . 131
6.11. Sampling Plans . . . 132
6.12. Kriging . . . 134
6.12.1. The Kriging Idea in a Nutshell . . . 134
6.12.2. Radial Basis Function (RBF) . . . 134
6.12.3. The Kriging Model . . . 136
6.12.4. The Condition Number . . . 142
6.12.5. MLE to estimate θ and p . . . 142
6.12.6. Implementing an MLE of the Model Parameters . . . 144
6.12.7. Kriging Prediction . . . 144
6.13. Kriging Example: Sinusoid Function . . . 145
6.13.1. Calculating the Correlation Matrix Ψ . . . 145
6.13.2. Computing the Ψ Matrix . . . 147
6.13.3. Selecting the New Locations . . . 148
6.13.4. Computing the ψ Vector . . . 148
6.13.5. Predicting at New Locations . . . 149
6.13.6. Visualization . . . 149
6.14. Cholesky Decomposition . . . 150
6.14.1. Example of Cholesky Decomposition . . . 150
6.14.2. Inverse Matrix Using Cholesky Decomposition . . . 151

6.15. Gaussian Processes—Some Background Information 152
6.15.1. Gaussian Process Prior 153
6.15.2. Covariance Function 153
6.15.3. Construction of the Covariance Matrix 155
6.15.4. Generation of Random Samples and Plotting the Realizations of the Random Function 158
6.15.5. Properties of the 1d Example 160
6.16. Jupyter Notebook 163

7. Introduction to spotpython 165
7.1. Advantages of the spotpython approach 165
7.2. Disadvantages of the spotpython approach 167
7.3. Example: `Spot` and the Sphere Function 167
7.3.1. The Objective Function: Sphere 167
7.3.2. The `Spot` Method as an Optimization Algorithm Using a Surrogate Model 168
7.4. `Spot` Parameters: `fun_evals`, `init_size` and `show_models` 170
7.5. Print the Results 172
7.6. Show the Progress 172
7.7. Visualizing the Optimization and Hyperparameter Tuning Process with TensorBoard 173
7.8. Jupyter Notebook 178

8. Multi-dimensional Functions 179
8.1. The Objective Function: 3-dim Sphere 179
8.1.1. Results 181
8.1.2. TensorBoard 184
8.1.3. Conclusion 185
8.2. Exercises 185
8.3. Selected Solutions 186
8.4. Jupyter Notebook 194

9. Isotropic and Anisotropic Kriging 195
9.1. Example: Isotropic `Spot` Surrogate and the 2-dim Sphere Function . . . 195
9.1.1. The Objective Function: 2-dim Sphere 195
9.1.2. Results 196
9.2. Example With Anisotropic Kriging 197
9.2.1. Taking a Look at the `theta` Values 199
9.3. Exercises 200
9.3.1. 1. The Branin Function `fun_branin` 200
9.3.2. 2. The Two-dimensional Sin-Cos Function `fun_sin_cos` 201
9.3.3. 3. The Two-dimensional Runge Function `fun_runge` 201
9.3.4. 4. The Ten-dimensional Wing-Weight Function `fun_wingwt` . . . 201
9.3.5. 5. The Two-dimensional Rosenbrock Function `fun_rosen` 202

9.4. Selected Solutions . . . 202
9.4.1. Solution to Exercise Section 9.3.5: The Two-dimensional Rosenbrock Function `fun_rosen` . . . 202
9.5. Jupyter Notebook . . . 208

10. Using `sklearn` Surrogates in `spotpython` 209
10.1. Example: Branin Function with `spotpython`'s Internal Kriging Surrogate 209
10.1.1. The Objective Function Branin . . . 209
10.1.2. Running the surrogate model based optimizer `Spot`: . . . 210
10.1.3. TensorBoard . . . 211
10.1.4. Print the Results . . . 212
10.1.5. Show the Progress and the Surrogate . . . 212
10.2. Example: Using Surrogates From scikit-learn . . . 213
10.2.1. GaussianProcessRegressor as a Surrogate . . . 214
10.3. Example: One-dimensional Sphere Function With `spotpython`'s Kriging 215
10.3.1. Results . . . 223
10.4. Example: `Sklearn` Model GaussianProcess . . . 224
10.5. Exercises . . . 233
10.5.1. 1. A decision tree regressor: `DecisionTreeRegressor` . . . 233
10.5.2. 2. A random forest regressor: `RandomForestRegressor` . . . 234
10.5.3. 3. Ordinary least squares Linear Regression: `LinearRegression` 234
10.5.4. 4. Linear least squares with l2 regularization: `Ridge` . . . 234
10.5.5. 5. Gradient Boosting: `HistGradientBoostingRegressor` . . . 234
10.5.6. 6. Comparison of Surrogates . . . 234
10.6. Selected Solutions . . . 235
10.6.1. Solution to Exercise Section 10.5.5: Gradient Boosting . . . 235
10.7. Jupyter Notebook . . . 257

11. Sequential Parameter Optimization: Gaussian Process Models 259
11.1. Gaussian Processes Regression: Basic Introductory `scikit-learn` Example . . . 259
11.1.1. Train and Test Data . . . 260
11.1.2. Building the Surrogate With `Sklearn` . . . 260
11.1.3. Plotting the `Sklearn`Model . . . 260
11.1.4. The `spotpython` Version . . . 261
11.1.5. Visualizing the Differences Between the `spotpython` and the `sklearn` Model Fits . . . 262
11.2. Exercises . . . 263
11.2.1. `Schonlau Example Function` . . . 263
11.2.2. `Forrester Example Function` . . . 263
11.2.3. `fun_runge Function (1-dim)` . . . 264
11.2.4. `fun_cubed (1-dim)` . . . 265
11.2.5. The Effect of Noise . . . 265

12. Expected Improvement **267**
12.1. Example: `Spot` and the 1-dim Sphere Function . . . 267
12.1.1. The Objective Function: 1-dim Sphere . . . 267
12.1.2. Results . . . 268
12.2. Same, but with EI as infill_criterion . . . 269
12.3. Non-isotropic Kriging . . . 271
12.4. Using `sklearn` Surrogates . . . 273
12.4.1. The spot Loop . . . 273
12.4.2. spot: The Initial Model . . . 274
12.4.3. Init: Build Initial Design . . . 275
12.4.4. Evaluate . . . 277
12.4.5. Build Surrogate . . . 277
12.4.6. A Simple Predictor . . . 277
12.5. Gaussian Processes regression: basic introductory example . . . 278
12.6. The Surrogate: Using scikit-learn models . . . 280
12.7. Additional Examples . . . 283
12.7.1. Optimize on Surrogate . . . 285
12.7.2. Evaluate on Real Objective . . . 285
12.7.3. Impute / Infill new Points . . . 285
12.8. Tests . . . 285
12.9. EI: The Famous Schonlau Example . . . 287
12.10. EI: The Forrester Example . . . 289
12.11. Noise . . . 292
12.12. Cubic Function . . . 295
12.13. Modifying Lambda Search Space . . . 301

13. Handling Noise **303**
13.1. Example: `Spot` and the Noisy Sphere Function . . . 303
13.1.1. The Objective Function: Noisy Sphere . . . 303
13.1.2. Reproducibility: Noise Generation and Seed Handling . . . 305
13.2. spotpython's Noise Handling Approaches . . . 307
13.3. Print the Results . . . 313
13.4. Noise and Surrogates: The Nugget Effect . . . 314
13.4.1. The Noisy Sphere . . . 314
13.5. Exercises . . . 317
13.5.1. Noisy `fun_cubed` . . . 317
13.5.2. `fun_runge` . . . 317
13.5.3. `fun_forrester` . . . 317
13.5.4. `fun_xsin` . . . 317

14. Optimal Computational Budget Allocation in `Spot` **319**
14.1. Example: `Spot`, OCBA, and the Noisy Sphere Function . . . 319
14.1.1. The Objective Function: Noisy Sphere . . . 319
14.2. Print the Results . . . 327

14.3. Noise and Surrogates: The Nugget Effect . . . 328
14.3.1. The Noisy Sphere . . . 328
14.4. Exercises . . . 331
14.4.1. Noisy `fun_cubed` . . . 331
14.4.2. `fun_runge` . . . 331
14.4.3. `fun_forrester` . . . 331
14.4.4. `fun_xsin` . . . 331

15. Kriging with Varying Correlation-p 333
15.1. Example: `Spot` Surrogate and the 2-dim Sphere Function . . . 333
15.1.1. The Objective Function: 2-dim Sphere . . . 333
15.1.2. Results . . . 334
15.2. Example With Modified p . . . 335
15.2.1. Taking a Look at the `p` Values . . . 337
15.3. Optimization of the `p` Values . . . 338
15.4. Optimization of Multiple `p` Values . . . 339
15.5. Exercises . . . 341
15.5.1. `fun_branin` . . . 341
15.5.2. `fun_sin_cos` . . . 342
15.5.3. `fun_runge` . . . 342
15.5.4. `fun_wingwt` . . . 342
15.6. Jupyter Notebook . . . 342

16. Factorial Variables 343
16.1. Jupyter Notebook . . . 346

17. User-Specified Functions: Extending the Analytical Class 347
17.1. The Objective Function: User Specified . . . 347
17.2. Results . . . 349
17.3. A Contour Plot . . . 350
17.4. Jupyter Notebook . . . 351

III. Data-Driven Modeling and Optimization 353

18. Data-Driven Modeling and Optimization 355
18.1. StatQuest Videos . . . 355
18.1.1. June, 11th 2024 . . . 355
18.1.2. Mathematical Models . . . 356
18.1.3. Hypothesis Testing and the Null-Hypothesis . . . 357
18.1.4. June, 18th 2024 . . . 359
18.1.5. Maximum Likelihood . . . 361
18.1.6. June, 25th 2024 . . . 365
18.1.7. t-SNE . . . 365
18.1.8. K-means clustering . . . 366

18.1.9. DBSCAN . . . 366
18.1.10. K-nearest neighbors . . . 366
18.1.11. Naive Bayes . . . 366
18.1.12. Gaussian Naive Bayes . . . 367
18.1.13. July, 2nd 2024 . . . 367
18.1.14. Additional Videos . . . 368
18.2. Introduction to Statistical Learning . . . 368
18.2.1. Opening Remarks and Examples . . . 368
18.3. Basics . . . 370
18.3.1. Histograms . . . 370
18.3.2. Probability Distributions . . . 373
18.3.3. Discrete Distributions . . . 375
18.4. Continuous Distributions . . . 377
18.4.1. Distribution functions: PDFs and CDFs . . . 378
18.4.2. Expectation (Continuous) . . . 378
18.4.3. Variance and Standard Deviation (Continuous) . . . 378
18.4.4. Uniform Distribution . . . 378
18.4.5. Normal Distribution . . . 379
18.4.6. The Mean, the Median, and the Mode . . . 382
18.4.7. The Exponential Distribution . . . 382
18.4.8. Population and Estimated Parameters . . . 383
18.4.9. Calculating the Mean, Variance, and Standard Deviation . . . 383
18.4.10. What is a Mathematical Model? . . . 383
18.4.11. Sampling from a Distribution . . . 383
18.4.12. Hypothesis Testing and the Null Hypothesis . . . 383
18.4.13. Alternative Hypotheses . . . 383
18.4.14. p-values: What They Are and How to Interpret Them . . . 383
18.4.15. How to Calculate p-values . . . 383
18.4.16. p-hacking: What It Is and How to Avoid It . . . 383
18.4.17. Covariance . . . 383
18.4.18. Pearson's Correlation . . . 383
18.4.19. Boxplots . . . 383
18.4.20. R-squared . . . 383
18.4.21. The Main Ideas of Fitting a Line to Data . . . 383
18.4.22. Linear Regression . . . 383
18.4.23. Multiple Regression . . . 383
18.5. Supervised Learning . . . 383
18.5.1. Statistical Learning and Regression . . . 385
18.5.2. Optimal Predictor . . . 386
18.5.3. Curse of Dimensionality and Parametric Models . . . 387
18.5.4. Assessing Model Accuracy and Bias-Variance Trade-off . . . 392
18.5.5. Classification Problems and K-Nearest Neighbors . . . 397
18.5.6. k-Nearest Neighbor Classification . . . 398
18.5.7. Minkowski Distance . . . 399
18.5.8. Unsuperivsed Learning: Classification . . . 399

IV. Machine Learning and AI 407

19. Machine Learning and Artificial Intelligence 409
19.1. Jupyter Notebooks . . . 409
19.2. Videos . . . 409
19.2.1. June, 11th 2024 . . . 409
19.2.2. June, 18th 2024 . . . 409
19.2.3. June, 25th 2024 . . . 410
19.2.4. CNNs . . . 410
19.2.5. RNN . . . 411
19.2.6. LSTM . . . 411
19.2.7. Pytorch/Lightning . . . 411
19.2.8. July, 2nd 2024 . . . 412
19.2.9. Additional Lecture (July, 9th 2024)? . . . 412
19.2.10. Additional Videos . . . 413
19.2.11. All Videos in a Playlist . . . 413
19.3. The StatQuest Introduction to PyTorch . . . 414
19.3.1. Build a Simple Neural Network in PyTorch . . . 414
19.3.2. Use the Neural Network and Graph the Output . . . 416
19.3.3. Optimize (Train) a Parameter in the Neural Network and Graph the Output . . . 417
19.4. Build a Long Short-Term Memory unit by hand using PyTorch + Lightning . . . 423
19.4.1. Train the LSTM unit and use Lightning and TensorBoard to evaluate: Part 1 - Getting Started . . . 428
19.4.2. Optimizing (Training) the Weights and Biases in the LSTM that we made by hand: Part 2 - Adding More Epochs without Starting Over . . . 430
19.5. Using and optimzing the PyTorch LSTM, nn.LSTM() . . . 432

V. Introduction to Hyperparameter Tuning 437

20. Hyperparameter Tuning 439
20.1. Structure of the Hyperparameter Tuning Chapters . . . 439
20.2. Goals of Hyperparameter Tuning . . . 439

VI. Hyperparameter Tuning with Sklearn 441

21. HPT: sklearn 443
21.1. Introduction to sklearn . . . 443

22. HPT: sklearn SVC on Moons Data 445
22.1. Step 1: Setup . . . 445

22.2. Step 2: Initialization of the Empty `fun_control` Dictionary 445
22.3. Step 3: SKlearn Load Data (Classification) 446
22.4. Step 4: Specification of the Preprocessing Model 448
22.5. Step 5: Select Model (`algorithm`) and `core_model_hyper_dict` 449
22.6. Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model` 451
22.6.1. Modify hyperparameter of type numeric and integer (boolean) . 452
22.6.2. Modify hyperparameter of type factor 452
22.6.3. Optimizers 453
22.7. Step 7: Selection of the Objective (Loss) Function 453
22.7.1. Predict Classes or Class Probabilities 453
22.8. Step 8: Calling the SPOT Function 454
22.8.1. The Objective Function 454
22.8.2. Run the `Spot` Optimizer 454
22.8.3. TensorBoard 456
22.9. Step 9: Results 457
22.10. Get Default Hyperparameters 459
22.11. Get SPOT Results 459
22.11.1. Plot: Compare Predictions 460
22.11.2. Detailed Hyperparameter Plots 462
22.11.3. Parallel Coordinates Plot 468
22.11.4. Plot all Combinations of Hyperparameters 468

23. Step 2: Initialization of the Empty `fun_control` Dictionary 469
23.1. Step 3: SKlearn Load Data (Classification) 470
23.2. Step 4: Specification of the Preprocessing Model 471
23.3. Step 5: Select Model (`algorithm`) and `core_model_hyper_dict` 471
23.4. Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model` 474
23.4.1. Modify hyperparameter of type numeric and integer (boolean) . 474
23.4.2. Modify hyperparameter of type factor 475
23.4.3. Optimizers 475
23.5. Step 7: Selection of the Objective (Loss) Function 475
23.5.1. Predict Classes or Class Probabilities 476
23.6. Step 8: Calling the SPOT Function 476
23.6.1. The Objective Function 476
23.6.2. Run the `Spot` Optimizer 476
23.6.3. TensorBoard 479
23.7. Get Default Hyperparameters 481
23.8. Get SPOT Results 481
23.8.1. Plot: Compare Predictions 482
23.8.2. Detailed Hyperparameter Plots 483
23.8.3. Parallel Coordinates Plot 487

VII. Hyperparameter Tuning with River 489

24. HPT: River 491
24.1. Introduction to River . . . 491

25. Simplifying Hyperparameter Tuning in Online Machine Learning—The spotRiverGUI 493
25.1. Introduction . . . 493
25.2. Installation and Starting . . . 494
25.2.1. Installation . . . 494
25.2.2. Starting the GUI . . . 495
25.3. Binary Classification . . . 495
25.3.1. Binary Classification Options . . . 495
25.3.2. Experiment Options . . . 498
25.3.3. Evaluation Options . . . 499
25.3.4. Online Machine Learning Model Options . . . 501
25.4. Regression . . . 503
25.5. Showing the Data . . . 503
25.6. Saving and Loading . . . 507
25.6.1. Saving the Experiment . . . 507
25.6.2. Loading an Experiment . . . 508
25.7. Running a New Experiment . . . 508
25.7.1. Starting and Stopping Tensorboard . . . 508
25.8. Performing the Analysis . . . 508
25.9. Summary and Outlook . . . 512
25.10 Appendix . . . 513
25.10.1 Adding new Tasks . . . 513

26. `river` Hyperparameter Tuning: Hoeffding Tree Regressor with Friedman Drift Data 515
26.1. The Friedman Drift Data Set . . . 515
26.2. Setup . . . 516
26.2.1. General Experiment Setup . . . 516
26.2.2. Data Setup . . . 517
26.2.3. Evaluation Setup . . . 518
26.2.4. River-Specific Setup . . . 518
26.2.5. Model Setup . . . 519
26.2.6. Objective Function Setup . . . 520
26.2.7. Surrogate Model Setup . . . 520
26.2.8. Summary: Setting up the Experiment . . . 520
26.2.9. Run the `Spot` Optimizer . . . 521
26.3. Using the `spotgui` . . . 523
26.4. Results . . . 523
26.5. Performance of the Model with Default Hyperparameters . . . 525
26.5.1. Get Default Hyperparameters and Fit the Model . . . 525

26.5.2. Evaluate the Model with Default Hyperparameters 525
26.5.3. Show Predictions of the Model with Default Hyperparameters . . 526
26.6. Get SPOT Results 527
26.7. Visualize Regression Trees 529
26.7.1. Spot Model 530
26.8. Detailed Hyperparameter Plots 531
26.9. Parallel Coordinates Plots 532

27. The Friedman Drift Data Set 535
27.1. Setup 536
27.1.1. Select a User Hyperdictionary 536
27.2. Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model` 537
27.2.1. Run the `Spot` Optimizer 538
27.3. Results 539
27.4. Performance of the Model with Default Hyperparameters 540
27.4.1. Get Default Hyperparameters and Fit the Model 540
27.4.2. Evaluate the Model with Default Hyperparameters 541
27.4.3. Show Predictions of the Model with Default Hyperparameters . . 542
27.5. Get SPOT Results 543
27.6. Detailed Hyperparameter Plots 545
27.7. Parallel Coordinates Plots 546

VIII Hyperparameter Tuning with PyTorch Lightning 547

28. Basic Lightning Module 549
28.1. Introduction 549
28.2. Starter Example: Transformer 550
28.3. Lightning Core Methods 551
28.3.1. Training Step 551
28.3.2. Validation Step 553
28.3.3. Test Step 554
28.3.4. Predict Step 554
28.4. Lightning Extras 556
28.4.1. Lightning: Save Hyperparameters 556
28.4.2. Lightning: Model Loading 556
28.5. Starter Example: Linear Neural Network 557
28.5.1. Hidden Layers 557
28.5.2. Hyperparameters 557
28.5.3. The LightningBasic Class 557
28.5.4. The Data Set: Diabetes 562
28.5.5. The DataLoaders 563
28.5.6. The Trainer 563
28.5.7. Using a DataModule 564

28.6. Using spotpython with Pytorch Lightning 565

29. Details of the Lightning Module Integration in spotpython 571
29.1. Introduction 571
29.2. 1. spotpython.fun.hyperlight.HyperLight.fun() 571
29.3. 2. spotpython.light.trainmodel.train_model() 572
29.4. 3. Trainer: fit and validate 574
29.4.1. Commented: Using the fun_control dictionary 575
29.4.2. Using the source code: 575
29.4.3. DataModule 582

30. User Specified Basic Lightning Module With spotpython 587
30.1. Introduction 587
30.1.1. Dataset 587
30.1.2. DataModule 589
30.2. The Neural Network: MyRegressor 594
30.3. Calling the Neural Network With spotpython 600
30.4. Looking at the Results 602
30.4.1. Tuning Progress 602
30.4.2. Tuned Hyperparameters and Their Importance 602
30.4.3. Get the Tuned Architecture 602

31. HPT PyTorch Lightning: Data 603
31.1. Setup 603
31.2. Initialization of the `fun_control` Dictionary 604
31.3. Loading the Diabetes Data Set 604
31.3.1. Data Set and Data Loader 604
31.3.2. Preparing Training, Validation, and Test Data 605
31.3.3. Dataset for spotpython 607
31.4. The LightDataModule 608
31.4.1. The `prepare_data()` Method 608
31.4.2. The `setup()` Method 608
31.4.3. The `train_dataloader()` Method 612
31.4.4. The `val_dataloader()` Method 613
31.4.5. The `test_dataloader()` Method 614
31.4.6. The `predict_dataloader()` Method 615
31.5. Using the `LightDataModule` in the `train_model()` Method 616
31.6. Further Information 617
31.6.1. Preprocessing 617

32. Hyperparameter Tuning with `spotpython` and `PyTorch` Lightning for the Diabetes Data Set 619
32.1. The Basic Setting 619
32.2. Looking at the Results 624
32.2.1. Tuning Progress 624

32.2.2. Tuned Hyperparameters and Their Importance 625
32.2.3. Get the Tuned Architecture 627
32.2.4. Test on the full data set 628
32.3. Cross Validation With Lightning 628
32.4. Extending the Basic Setup 629
32.4.1. General Experiment Setup 630
32.4.2. Data Setup 630
32.4.3. Objective Function `fun` 630
32.4.4. Core-Model Setup 630
32.4.5. Hyperdict Setup 631
32.4.6. Other Settings 631
32.5. Tensorboard 631
32.6. Loading the Saved Experiment and Getting the Hyperparameters of the Tuned Model 631
32.7. Using the `spotgui` 632
32.8. Summary 633

33. Hyperparameter Tuning with PyTorch Lightning and User Data Sets 635
33.1. Loading a User Specified Data Set 635
33.2. Summary 640

34. Hyperparameter Tuning with PyTorch Lightning and User Models 641
34.1. Using a User Specified Model 641
34.2. Details 645
34.2.1. Model Setup 645
34.2.2. The `my_hyper_dict.py` File 645
34.2.3. The `my_hyper_dict.json` File 645
34.2.4. The `my_regressor.py` File 648
34.3. Connection with the LightDataModule 652
34.3.1. The `prepare_data()` Method 653
34.3.2. The `setup()` Method 653
34.3.3. The `train_dataloader()` Method 658
34.3.4. The `val_dataloader()` Method 659
34.3.5. The `test_dataloader()` Method 660
34.3.6. The `predict_dataloader()` Method 661
34.4. Using the `LightDataModule` in the `train_model()` Method 663
34.5. The Last Connection: The `HyperLight` Class 664
34.6. Further Information 665
34.6.1. Preprocessing 665

35. Hyperparameter Tuning with PyTorch Lightning: ResNets 667
35.1. Residual Neural Networks 667
35.1.1. Residual Connections 667
35.1.2. Implementation of the Original ResNet Block 668
35.1.3. Implementation of the Pre-Activation ResNet Block 672

35.1.4. The Overall ResNet Architecture 675

36. Neural ODEs 681
36.1. Neural Ordinary Differential Equations 681
36.2. Regression Example 687
36.2.1. Specifying the Dynamics Layer 687
36.3. Further Reading 690

37. Neural ODE Example 693
37.1. Implementation of a Neural ODE 693

38. Physics Informed Neural Networks 707
38.1. PINNs 707
38.2. Generation and Visualization of the Training Data and the Ground Truth (Function) 707
38.3. Gradient With Autograd 710
38.4. Network 710
38.5. Basic Neutral Network 713
38.6. PINNs 716
38.6.1. PINNs: Parameter Estimation 719
38.7. Summary 724

39. Hyperparameter Tuning with PyTorch Lightning: Physics Informed Neural Networks 725
39.1. PINNs 725
39.1.1. The Ground Truth Model 725
39.1.2. Required Files 727
39.1.3. The New `pinn_hyperdict.py` File 728
39.1.4. The New `pinn_regressor.py` File 729
39.1.5. The New `pinn_hyperdict.json` File 736

40. Explainable AI with SpotPython and Pytorch 737
40.1. Running the Hyperparameter Tuning or Loading the Existing Model . . 738
40.2. Results from the Hyperparameter Tuning Experiment 739
40.2.1. Getting the Best Model, i.e, the Tuned Architecture 740
40.3. Training the Tuned Architecture on the Test Data 741
40.4. Visualizing the Neural Network Architecture 743
40.5. XAI Methods 744
40.5.1. Weights 744
40.5.2. Activations 745
40.5.3. Gradients 745
40.5.4. Getting the Weights 745
40.5.5. Getting the Activations 753
40.5.6. Getting the Gradients 756

40.6. Feature Attributions . . . 765
40.6.1. Integrated Gradients . . . 765
40.6.2. Deep Lift . . . 766
40.6.3. Feature Ablation . . . 768
40.7. Conductance . . . 770

41. HPT PyTorch Lightning Transformer: Introduction 773
41.1. Transformer Basics . . . 773
41.1.1. Embedding . . . 773
41.1.2. Attention . . . 774
41.1.3. Self-Attention . . . 776
41.1.4. Masked Self-Attention . . . 776
41.1.5. Generation of Outputs . . . 776
41.1.6. End-Of-Sequence-Token . . . 777
41.2. Details of the Implementation . . . 777
41.2.1. Dot Product Attention . . . 779
41.2.2. Scaled Dot Product Attention . . . 780
41.3. Example: Transformer in Lightning . . . 781
41.3.1. Downloading the Pretrained Models . . . 782
41.3.2. The Transformer Architecture . . . 783
41.3.3. Attention Mechanism . . . 783
41.3.4. Multi-Head Attention . . . 784
41.3.5. Permutation Equivariance . . . 787
41.3.6. Transformer Encoder . . . 787
41.3.7. Layer Normalization and Feed-Forward Network . . . 789
41.3.8. Positional Encoding . . . 791
41.3.9. Learning Rate Warm-up . . . 794
41.3.10. PyTorch Lightning Module . . . 797
41.4. Experiment: Sequence to Sequence . . . 799
41.4.1. Dataset and Data Loaders . . . 799
41.4.2. The Reverse Predictor Class . . . 801
41.4.3. Gradient Clipping . . . 801
41.4.4. Implementation of the Lightning Trainer . . . 802
41.4.5. Training the Model . . . 803
41.5. Visualizing Attention Maps . . . 804
41.6. Conclusion . . . 806
41.7. Additional Considerations . . . 806
41.7.1. Complexity and Path Length . . . 806
41.8. Further Reading . . . 807

42. Hyperparameter Tuning of a Transformer Network with PyTorch Lightning 809
42.1. Basic Setup . . . 809
42.2. Looking at the Results . . . 812
42.2.1. Tuning Progress . . . 812
42.2.2. Tuned Hyperparameters and Their Importance . . . 813

42.3. Hyperparameter Considerations . . . 813
42.3.1. Important: Constraints and Interconnections: . . . 814
42.3.2. Practical Considerations: . . . 815
42.4. Summary . . . 816

43. Saving and Loading 817
43.1. spotpython: Saving and Loading Optimization Experiments . . . 817
43.1.1. Getting the Tuned Hyperparameters . . . 819
43.2. spotpython as a Hyperparameter Tuner . . . 820
43.2.1. The Diabetes Data Set . . . 820
43.3. Saving and Loading PyTorch Lightning Models . . . 823
43.3.1. Get the Tuned Architecture . . . 823
43.3.2. Load a Model from Checkpoint . . . 824
43.4. Converting a Lightning Model to a Plain Torch Model . . . 825
43.4.1. The Function `get_removed_attributes_and_base_net` . . . 825
43.4.2. An Example how to use the Plain Torch Net . . . 825

44. Hyperparameter Tuning with `spotpython` and `PyTorch` Lightning for the Diabetes Data Set Using a ResNet Model 827
44.1. Looking at the Results . . . 831
44.1.1. Tuning Progress . . . 831
44.1.2. Tuned Hyperparameters and Their Importance . . . 831
44.1.3. Get the Tuned Architecture . . . 833
44.1.4. Test on the full data set . . . 833
44.1.5. Cross Validation With Lightning . . . 834
44.2. Summary . . . 835

45. Hyperparameter Tuning with `spotpython` and `PyTorch` Lightning for the Diabetes Data Set Using a User Specified ResNet Model 837
45.1. Looking at the Results . . . 841
45.1.1. Tuning Progress . . . 841
45.1.2. Tuned Hyperparameters and Their Importance . . . 842
45.1.3. Get the Tuned Architecture . . . 844
45.2. Details of the User-Specified ResNet Model . . . 844
45.2.1. `my_resnet.py` . . . 845
45.2.2. `my_hyperdict.py` . . . 849
45.2.3. `my_hyperdict.json` . . . 850
45.3. Summary . . . 853

46. Hyperparameter Tuning with `spotpython` and `PyTorch` Lightning Using a CondNet Model 855
46.1. Looking at the Results . . . 858
46.1.1. Tuning Progress . . . 858
46.1.2. Tuned Hyperparameters and Their Importance . . . 859
46.1.3. Get the Tuned Architecture . . . 861

Appendices 863

A. Introduction to Jupyter Notebook **863**
A.1. Different Notebook cells . . . 863
A.1.1. Code cells . . . 863
A.1.2. Markdown cells . . . 863
A.1.3. Raw cells . . . 864
A.2. Install Packages . . . 864
A.3. Load Packages . . . 865
A.4. Functions in Python . . . 865
A.5. List of Useful Jupyter Notebook Shortcuts . . . 866

B. Git Introduction **869**
B.1. Learning Objectives . . . 869
B.2. Basics of Git . . . 869
B.2.1. Initializing a Repository: `git init` . . . 869
B.2.2. Ignoring Files: `.gitignore` . . . 870
B.2.3. Adding Changes to the Staging Area: `git add` . . . 870
B.2.4. Transferring Changes to Memory: `git commit` . . . 871
B.2.5. Check the Status of Your Repository: `git status` . . . 872
B.2.6. Review Your Repository's History: `git log` . . . 873
B.3. Branches (Timelines) . . . 873
B.3.1. Creating an Alternative Timeline: `git branch` . . . 873
B.3.2. The Pointer to the Current Branch: `HEAD` . . . 874
B.3.3. Switching to an Alternative Timeline: `git switch` . . . 874
B.3.4. Switching to an Alternative Timeline and Making Changes: `git checkout` . . . 874
B.3.5. The Difference Between `checkout` and `switch` . . . 875
B.4. Merging Branches and Resolving Conflicts . . . 876
B.4.1. git `merge`: Merging Two Timelines . . . 876
B.4.2. Resolving Conflicts When Merging . . . 877
B.4.3. git `revert`: Undoing Something . . . 879
B.5. Downloading from GitLab . . . 881
B.6. Advanced . . . 881
B.6.1. git `rebase`: Moving the Base of a Branch . . . 881
B.7. Exercises . . . 883
B.7.1. Create project folder . . . 883
B.8. Initialize repo . . . 883
B.8.1. Do not upload / ignore certain file types . . . 884
B.8.2. Create file and stage it . . . 884
B.8.3. Create another file and check status . . . 884
B.8.4. Commit changes . . . 884
B.8.5. Create a new branch and switch to it . . . 884
B.8.6. Commit changes in the new branch . . . 885
B.8.7. Merge branch into main . . . 885

B.8.8. Resolve merge conflict . . . 885

C. Python Introduction **887**
C.1. Recommendations . . . 887

D. Documentation of the Sequential Parameter Optimization **889**
D.1. An Initial Example . . . 889
D.2. Organization . . . 891
D.3. The Spot Object . . . 892
D.4. Run . . . 892
D.5. Print the Results . . . 893
D.6. Show the Progress . . . 893
D.7. Visualize the Surrogate . . . 893
D.8. Run With a Specific Start Design . . . 894
D.9. Init: Build Initial Design . . . 895
D.10. Replicability . . . 896
D.11. Surrogates . . . 897
D.11.1. A Simple Predictor . . . 897
D.12. Tensorboard Setup . . . 897
D.12.1. Tensorboard Configuration . . . 897
D.12.2. Starting TensorBoard . . . 898
D.13. Demo/Test: Objective Function Fails . . . 898
D.14. Handling Results: Printing, Saving, and Loading . . . 899
D.15. `spotpython` as a Hyperparameter Tuner . . . 899
D.15.1. Modifying Hyperparameter Levels . . . 899

E. Datasets **905**
E.1. The Diabetes Data Set . . . 905
E.1.1. Data Exploration of the sklearn Diabetes Data Set . . . 905
E.1.2. Generating the PyTorch Data Set . . . 906
E.2. The Friedman Drift Dataset . . . 907
E.2.1. The Friedman Drift Dataset as Implemented in `river` . . . 907
E.2.2. The Friedman Drift Data Set from `spotpython` . . . 910

F. Using Slurm **913**
F.1. Introduction . . . 913
F.2. Prepare the Slurm Scripts on the Remote Machine . . . 913
F.3. Generate a `spotpython` Configuration . . . 915
F.4. Copy the Configuration to the Remote Machine . . . 916
F.5. Run the `spotpython` Code on the Remote Machine . . . 916
F.6. Copy the Results to the Local Machine . . . 916
F.7. Analyze the Results on the Local Machine . . . 917
F.7.1. Visualizing the Tuning Progress . . . 917
F.7.2. Design Table with Default and Tuned Hyperparameters . . . 917
F.7.3. Plotting Important Hyperparameters . . . 917

F.7.4. The Tuned Hyperparameters 918

G. Python Package Building **919**
Introduction 919
G.1. Create a Conda Environment 919
G.2. Download the User Package 919
G.3. Build the User Package 919
G.4. Open the Documentation of the User Package 920

H. Solutions to Selected Exercises **921**
H.1. Data-Driven Modeling and Optimization 921
H.1.1. Histograms 921
H.1.2. The Normal Distribution 921
H.1.3. The mean, the media, and the mode 922
H.1.4. The exponential distribution 922
H.1.5. Population and Estimated Parameters 922
H.1.6. Calculating the Mean, Variance and Standard Deviation 922
H.1.7. Hypothesis Testing and the Null-Hypothesis 922
H.1.8. Alternative Hypotheses, Main Ideas 923
H.1.9. p-values: What they are and how to interpret them 923
H.1.10. How to calculate p-values 923
H.1.11. p-hacking: What it is and how to avoid it 924
H.1.12. Covariance 924
H.1.13. Pearson's Correlation 924
H.1.14. Boxplots 925
H.1.15. Power Analysis 925
H.1.16. The Central Limit Theorem 925
H.1.17. Boxplots 925
H.1.18. R-squared 926
H.1.19. Linear Regression 926
H.1.20. Multiple Regression 926
H.1.21. A Gentle Introduction to Machine Learning 926
H.1.22. Maximum Likelihood 926
H.1.23. Probability is not Likelihood 926
H.1.24. Cross Validation 927
H.1.25. The Confusion Matrix 927
H.1.26. Sensitivity and Specificity 927
H.1.27. Bias and Variance 927
H.1.28. Mutual Information 927
H.1.29. Principal Component Analysis (PCA) 928
H.1.30. t-SNE 928
H.1.31. K-means clustering 929
H.1.32. DBSCAN 929
H.1.33. K-nearest neighbors 930
H.1.34. Naive Bayes 930

H.1.35. Gaussian Naive Bayes 930
H.1.36. Trees 930
H.2. Machine Learning and Artificial Intelligence 931
H.2.1. Backpropagation 931
H.2.2. Gradient Descent 931
H.2.3. ReLU 931
H.2.4. CNNs 931
H.2.5. RNN 932
H.2.6. LSTM 932
H.2.7. Pytorch/Lightning 932
H.2.8. Embeddings 932
H.2.9. Sequence to Sequence Models 933
H.2.10. Transformers 933

References **935**

Preface

This document provides a comprehensive guide to hyperparameter tuning using spotpython for scikit-learn, scipy-optimize, River, and PyTorch. The first part introduces fundamental ideas from optimization. The second part discusses numerical issues and introduces spotpython's surrogate model-based optimization process. The thirs part focuses on hyperparameter tuning. Several case studies are presented, including hyperparameter tuning for sklearn models such as Support Vector Classification, Random Forests, Gradient Boosting (XGB), and K-nearest neighbors (KNN), as well as a Hoeffding Adaptive Tree Regressor from river. The integration of spotpython into the PyTorch and PyTorch Lightning training workflow is also discussed. With a hands-on approach and step-by-step explanations, this cookbook serves as a practical starting point for anyone interested in hyperparameter tuning with Python. Highlights include the interplay between Tensorboard, PyTorch Lightning, spotpython, spotriver, and River. This publication is under development, with updates available on the corresponding webpage.

! Important: This book is still under development.

The most recent version of this book is available at https://sequential-parameter-optimization.github.io/Hyperparameter-Tuning-Cookbook/

Book Structure

This document is structured in three parts. The first part presents an introduction to optimization. The second part describes numerical methods, and the third part presents hyperparameter tuning.

Hyperparameter Tuning Reference

- The open access book Bartz et al. (2022) provides a comprehensive overview of hyperparameter tuning. It can be downloaded from https://link.springer.com/book/10.1007/978-981-19-5170-1.

> **Note**
>
> The `.ipynb` notebook (Bartz-Beielstein 2023) is updated regularly and reflects updates and changes in the `spotpython` package. It can be downloaded from https://github.com/sequential-parameter-optimization/spotpython/blob/main/notebooks/14_spot_ray_hpt_torch_cifar10.ipynb.

Software Used in this Book

scikit-learn is a Python module for machine learning built on top of SciPy and is distributed under the 3-Clause BSD license. The project was started in 2007 by David Cournapeau as a Google Summer of Code project, and since then many volunteers have contributed.

PyTorch is an optimized tensor library for deep learning using GPUs and CPUs. Lightning is a lightweight PyTorch wrapper for high-performance AI research. It allows you to decouple the research from the engineering.

River is a Python library for online machine learning. It is designed to be used in real-world environments, where not all data is available at once, but streaming in.

spotpython ("Sequential Parameter Optimization Toolbox in Python") is the Python version of the well-known hyperparameter tuner SPOT, which has been developed in the R programming environment for statistical analysis for over a decade. The related open-access book is available here: Hyperparameter Tuning for Machine and Deep Learning with R—A Practical Guide.

spotriver provides an interface between spotpython and River.

Citation

If this document has been useful to you and you wish to cite it in a scientific publication, please refer to the following paper, which can be found on arXiv: https://arxiv.org/abs/2307.10262.

```
@ARTICLE{bart23iArXiv,
      author = {{Bartz-Beielstein}, Thomas},
      title = "{Hyperparameter Tuning Cookbook:
          A guide for scikit-learn, PyTorch, river, and spotpython}",
     journal = {arXiv e-prints},
    keywords = {Computer Science - Machine Learning,
      Computer Science - Artificial Intelligence, 90C26, I.2.6, G.1.6},
```

```
          year = 2023,
         month = jul,
           eid = {arXiv:2307.10262},
         pages = {arXiv:2307.10262},
           doi = {10.48550/arXiv.2307.10262},
archivePrefix = {arXiv},
        eprint = {2307.10262},
 primaryClass = {cs.LG},
        adsurl = {https://ui.adsabs.harvard.edu/abs/2023arXiv230710262B},
       adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

Part I.

Optimization

1. Introduction: Optimization

1.1. Optimization, Simulation, and Surrogate Modeling

- We will consider the interplay between
  - mathematical models,
  - numerical approximation,
  - simulation,
  - computer experiments, and
  - field data
- Experimental design will play a key role in our developments, but not in the classical regression and response surface methodology sense
- Challenging real-data/real-simulation examples benefiting from modern surrogate modeling methodology
- We will consider the classical, response surface methodology (RSM) approach, and then move on to more modern approaches
- All approaches are based on surrogates

1.2. Surrogates

- Gathering data is **expensive**, and sometimes getting exactly the data you want is impossible or unethical
- **Surrogate**: substitute for the real thing
- In statistics, draws from predictive equations derived from a fitted model can act as a surrogate for the data-generating mechanism
- Benefits of the surrogate approach:
  - Surrogate could represent a cheaper way to explore relationships, and entertain "what ifs?"
  - Surrogates favor faithful yet pragmatic reproduction of dynamics:
    - interpretation,
    - establishing causality, or
    - identification
  - Many numerical simulators are **deterministic**, whereas field observations are noisy or have measurement error

1.2.1. Costs of Simulation

- Computer simulations are generally cheaper (but not always!) than physical observation
- Some computer simulations can be just as expensive as field experimentation, but computer modeling is regarded as easier because:
  - the experimental apparatus is better understood
  - more aspects may be controlled.

1.2.2. Mathematical Models and Meta-Models

- Use of mathematical models leveraging numerical solvers has been commonplace for some time
- Mathematical models became more complex, requiring more resources to simulate/solve numerically
- Practitioners increasingly relied on **meta-models** built off of limited simulation campaigns

1.2.3. Surrogates = Trained Meta-models

- Data collected via expensive computer evaluations tuned flexible functional forms that could be used in lieu of further simulation to
  - save money or computational resources;
  - cope with an inability to perform future runs (expired licenses, off-line or over-impacted supercomputers)
- Trained meta-models became known as **surrogates**

1.2.4. Computer Experiments

- **Computer experiment**: design, running, and fitting meta-models.
  - Like an ordinary statistical experiment, except the data are generated by computer codes rather than physical or field observations, or surveys
- **Surrogate modeling** is statistical modeling of computer experiments

1.2.5. Limits of Mathematical Modeling

- Mathematical biologists, economists and others had reached the limit of equilibrium-based mathematical modeling with cute closed-form solutions
- **Stochastic simulations replace deterministic solvers** based on FEM, Navier–Stokes or Euler methods
- Agent-based simulation models are used to explore predator-prey (Lotka–Voltera) dynamics, spread of disease, management of inventory or patients in health insurance markets
- Consequence: the distinction between surrogate and statistical model is all but gone

1.2.6. Example: Why Computer Simulations are Necessary

- You can't seed a real community with Ebola and watch what happens
- If there's (real) field data, say on a historical epidemic, further experimentation may be almost entirely limited to the mathematical and computer modeling side
- Classical statistical methods offer little guidance

1.2.7. Simulation Requirements

- Simulation should
  - enable rich **diagnostics** to help criticize that models
  - **understanding** its sensitivity to inputs and other configurations
  - providing the ability to **optimize** and
  - refine both **automatically** and with expert intervention
- And it has to do all that while remaining **computationally tractable**
- One perspective is so-called **response surface methods** (RSMs):
- a poster child from industrial statistics' heyday, well before information technology became a dominant industry

! Goals

- How to choose models and optimizers for solving real-world problems
- How to use simulation to understand and improve processes

1.3. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

2. Aircraft Wing Weight Example

2.1. AWWE Equation

- Example from Forrester et al.
- Understand the **weight** of an unpainted light aircraft wing as a function of nine design and operational parameters:

$$W = 0.036 S_W^{0.758} \times W_{fw}^{0.0035} \left(\frac{A}{\cos^2 \Lambda}\right)^{0.6} \times q^{0.006} \times \lambda^{0.04}$$
$$\times \left(\frac{100 R_{tc}}{\cos \Lambda}\right)^{-0.3} \times (N_z W_{dg})^{0.49}$$

2.2. AWWE Parameters and Equations (Part 1)

Table 2.1.: Aircraft Wing Weight Parameters

Symbol	Parameter	Baseline	Minimum	Maximum
S_W	Wing area (ft^2)	174	150	200
W_{fw}	Weight of fuel in wing (lb)	252	220	300
A	Aspect ratio	7.52	6	10
Λ	Quarter-chord sweep (deg)	0	-10	10
q	Dynamic pressure at cruise (lb/ft^2)	34	16	45
λ	Taper ratio	0.672	0.5	1
R_{tc}	Aerofoil thickness to chord ratio	0.12	0.08	0.18
N_z	Ultimate load factor	3.8	2.5	6
W_{dg}	Flight design gross weight (lb)	2000	1700	2500
W_p	paint weight (lb/ft^2)	0.064	0.025	0.08

The study begins with a baseline Cessna C172 Skyhawk Aircraft as its reference point. It aims to investigate the impact of wing area and fuel weight on the overall weight of the aircraft. Two crucial parameters in this analysis are the aspect ratio (A), defined as the ratio of the wing's length to the average chord (thickness of the airfoil), and the

taper ratio (λ), which represents the ratio of the maximum to the minimum thickness of the airfoil or the maximum to minimum chord.

It's important to note that the equation used in this context is not a computer simulation but will be treated as one for the purpose of illustration. This approach involves employing a true mathematical equation, even if it's considered unknown, as a useful tool for generating realistic settings to test the methodology. The functional form of this equation was derived by "calibrating" known physical relationships to curves obtained from existing aircraft data, as referenced in Raymer 2012. Essentially, it acts as a surrogate for actual measurements of aircraft weight.

Examining the mathematical properties of the AWWE (Aircraft Weight With Wing Area and Fuel Weight Equation), it is evident that the response is highly nonlinear concerning its inputs. While it's common to apply the logarithm to simplify equations with complex exponents, even when modeling the logarithm, which transforms powers into slope coefficients and products into sums, the response remains nonlinear due to the presence of trigonometric terms. Given the combination of nonlinearity and high input dimension, simple linear and quadratic response surface approximations are likely to be inadequate for this analysis.

2.3. Goals: Understanding and Optimization

The primary goals of this study revolve around understanding and optimization:

1. **Understanding**: One of the straightforward objectives is to gain a deep understanding of the input-output relationships in this context. Given the global perspective implied by this setting, it becomes evident that a more sophisticated model is almost necessary. At this stage, let's focus on this specific scenario to establish a clear understanding.
2. **Optimization**: Another application of this analysis could be optimization. There may be an interest in minimizing the weight of the aircraft, but it's likely that there will be constraints in place. For example, the presence of wings with a nonzero area is essential for the aircraft to be capable of flying. In situations involving (constrained) optimization, a global perspective and, consequently, the use of flexible modeling are vital.

The provided Python code serves as a genuine computer implementation that "solves" a mathematical model. It accepts arguments encoded in the unit cube, with defaults used to represent baseline settings, as detailed in the table labeled as Table 2.1. To map values from the interval $[a, b]$ to the interval $[0, 1]$, the following formula can be employed:

$$y = f(x) = \frac{x - a}{b - a}.$$

To reverse this mapping and obtain the original values, the formula

$$g(y) = a + (b - a)y$$

can be used.

```
import numpy as np

def wingwt(Sw=0.48, Wfw=0.4, A=0.38, L=0.5, q=0.62, l=0.344,  Rtc=0.4, Nz=0.37, Wdg=0.38):
    # put coded inputs back on natural scale
    Sw = Sw * (200 - 150) + 150
    Wfw = Wfw * (300 - 220) + 220
    A = A * (10 - 6) + 6
    L = (L * (10 - (-10)) - 10) * np.pi/180
    q = q * (45 - 16) + 16
    l = l * (1 - 0.5) + 0.5
    Rtc = Rtc * (0.18 - 0.08) + 0.08
    Nz = Nz * (6 - 2.5) + 2.5
    Wdg = Wdg*(2500 - 1700) + 1700
    # calculation on natural scale
    W = 0.036 * Sw**0.758 * Wfw**0.0035 * (A/np.cos(L)**2)**0.6 * q**0.006
    W = W * l**0.04 * (100*Rtc/np.cos(L))**(-0.3) * (Nz*Wdg)**(0.49)
    return(W)
```

2.4. Properties of the Python "Solver"

The compute time required by the "wingwt" solver is extremely short and can be considered trivial in terms of computational resources. The approximation error is exceptionally small, effectively approaching machine precision, which indicates the high accuracy of the solver's results.

To simulate time-consuming evaluations, a deliberate delay is introduced by incorporating a `sleep(3600)` command, which effectively synthesizes a one-hour execution time for a particular evaluation.

Moving on to the AWWE visualization, plotting in two dimensions is considerably simpler than dealing with nine dimensions. To aid in creating visual representations, the code provided below establishes a grid within the unit square to facilitate the generation of sliced visuals. This involves generating a "meshgrid" as outlined in the code.

```
import numpy as np
x = np.linspace(0, 1, 3)
y = np.linspace(0, 1, 3)
```

```
X, Y = np.meshgrid(x, y)
zp = zip(np.ravel(X), np.ravel(Y))
list(zp)
```

```
[(np.float64(0.0), np.float64(0.0)),
 (np.float64(0.5), np.float64(0.0)),
 (np.float64(1.0), np.float64(0.0)),
 (np.float64(0.0), np.float64(0.5)),
 (np.float64(0.5), np.float64(0.5)),
 (np.float64(1.0), np.float64(0.5)),
 (np.float64(0.0), np.float64(1.0)),
 (np.float64(0.5), np.float64(1.0)),
 (np.float64(1.0), np.float64(1.0))]
```

The coding used to transform inputs from natural units is largely a matter of taste, so long as it's easy to undo for reporting back on original scales

```
%matplotlib inline
import matplotlib.pyplot as plt
# plt.style.use('seaborn-white')
import numpy as np
x = np.linspace(0, 1, 100)
y = np.linspace(0, 1, 100)
X, Y = np.meshgrid(x, y)
```

2.5. Plot 1: Load Factor (N_z) and Aspect Ratio (A)

We will vary N_z and A, with other inputs fixed at their baseline values.

```
z = wingwt(A = X, Nz = Y)
fig = plt.figure(figsize=(7., 5.))
plt.contourf(X, Y, z, 20, cmap='jet')
plt.xlabel("A")
plt.ylabel("Nz")
plt.title("Load factor (Nz) vs. Aspect Ratio (A)")
plt.colorbar()
plt.show()
```

Contour plots can be refined, e.g., by adding explicit contour lines as shown in the following figure.

```
contours = plt.contour(X, Y, z, 4, colors='black')
plt.clabel(contours, inline=True, fontsize=8)
plt.xlabel("A")
plt.ylabel("Nz")

plt.imshow(z, extent=[0, 1, 0, 1], origin='lower',
           cmap='jet', alpha=0.9)
plt.colorbar()
```

The interpretation of the AWWE plot can be summarized as follows:

- The figure displays the weight response as a function of two variables, N_z and A, using an image-contour plot.
- The slight curvature observed in the contours suggests an interaction between these two variables.
- Notably, the range of outputs depicted in the figure, spanning from approximately 160 to 320, nearly encompasses the entire range of outputs observed from various input settings within the full 9-dimensional input space.
- The plot indicates that aircraft wings tend to be heavier when the aspect ratios (A) are high.
- This observation aligns with the idea that wings are designed to withstand and accommodate high gravitational forces (g-forces, large N_z), and there may be a compounding effect where larger values of N_z contribute to increased wing weight.
- It's plausible that this phenomenon is related to the design considerations of fighter jets, which cannot have the efficient and lightweight glider-like wings typically found in other types of aircraft.

2.6. Plot 2: Taper Ratio and Fuel Weight

- The same experiment for two other inputs, e.g., taper ratio λ and fuel weight W_{fw}

```
z = wingwt(Wfw = X,  Nz = Y)
contours = plt.contour(X, Y, z, 4, colors='black')
plt.clabel(contours, inline=True, fontsize=8)
plt.xlabel("WfW")
plt.ylabel("l")

plt.imshow(z, extent=[0, 1, 0, 1], origin='lower',
           cmap='jet', alpha=0.9)
plt.colorbar();
```

- Interpretation of Taper Ratio (l) and Fuel Weight (W_{fw})
  - Apparently, neither input has much effect on wing weight:
    - with λ having a marginally greater effect, covering less than 4 percent of the span of weights observed in the $A \times N_z$ plane
  - There's no interaction evident in $\lambda \times W_{fw}$

2.7. The Big Picture: Combining all Variables

```
pl = ["Sw", "Wfw", "A", "L", "q", "l",  "Rtc", "Nz", "Wdg"]
```

```
import math

Z = []
Zlab = []
l = len(pl)
# lc = math.comb(l,2)
for i in range(l):
    for j in range(i+1, l):
    # for j in range(l):
        # print(pl[i], pl[j])
        d = {pl[i]: X, pl[j]: Y}
        Z.append(wingwt(**d))
        Zlab.append([pl[i],pl[j]])
```

Now we can generate all 36 combinations, e.g., our first example is combination `p = 19`.

```
p = 19
Zlab[p]
```

```
['A', 'Nz']
```

To help interpret outputs from experiments such as this one—to level the playing field when comparing outputs from other pairs of inputs—code below sets up a color palette that can be re-used from one experiment to the next. We use the arguments `vmin=180` and `vmax =360` to implement comparibility

```
plt.contourf(X, Y, Z[p], 20, cmap='jet', vmin=180, vmax=360)
plt.xlabel(Zlab[p][0])
plt.ylabel(Zlab[p][1])
plt.colorbar()
```

- Let's plot the second example, taper ratio λ and fuel weight W_{fw}
- This is combination `11`:

```
p = 11
Zlab[p]
```

```
['Wfw', 'l']
```

```
plt.contourf(X, Y, Z[p], 20, cmap='jet', vmin=180, vmax=360)
plt.xlabel(Zlab[p][0])
plt.ylabel(Zlab[p][1])
plt.colorbar()
```

- Using a global colormap indicates that these variables have minor effects on the wing weight.
- Important factors can be detected by visual inspection
- Plotting the Big Picture: we can plot all 36 combinations in one figure.

```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
import numpy as np

fig = plt.figure(figsize=(20., 20.))
grid = ImageGrid(fig, 111,  # similar to subplot(111)
                 nrows_ncols=(6,6),  # creates 2x2 grid of axes
                 axes_pad=0.5,  # pad between axes in inch.
                 share_all=True,
                 label_mode="all",
                 )
i = 0
for ax, im in zip(grid, Z):
    # Iterating over the grid returns the Axes.
    ax.set_xlabel(Zlab[i][0])
    ax.set_ylabel(Zlab[i][1])
    # ax.set_title(Zlab[i][1] + " vs. " + Zlab[i][0])
    ax.contourf(X, Y, im, 30, cmap = "jet",  vmin = 180, vmax = 360)
```

```
    i = i + 1

plt.show()
```

2.8. AWWE Landscape

- Our Observations
  1. The load factor N_z, which determines the magnitude of the maximum aerodynamic load on the wing, is very active and involved in interactions with other variables.

  - Classic example: the interaction of N_z with the aspect ratio A indicates a heavy wing for high aspect ratios and large g-forces
  - This is the reaon why highly manoeuvrable fighter jets cannot have very efficient, glider wings)

  2. Aspect ratio A and airfoil thickness to chord ratio R_{tc} have nonlinear interactions.
  3. Most important variables:

  - Ultimate load factor N_z, wing area S_w, and flight design gross weightW_{dg}.

  4. Little impact: dynamic pressure q, taper ratio l, and quarter-chord sweep L.

- Expert Knowledge
  - Aircraft designers know that the overall weight of the aircraft and the wing area must be kept to a minimum
  - the latter usually dictated by constraints such as required stall speed, landing distance, turn rate, etc.

2.9. Summary of the First Experiments

- First, we considered two pairs of inputs, out of 36 total pairs
- Then, the "Big Picture":
  - For each pair we evaluated `wingwt` 10,000 times
- Doing the same for all pairs would require 360K evaluations:
  - not a reasonable number with a real computer simulation that takes any non-trivial amount of time to evaluate
  - Only 1s per evaluation: > 100 hours
- Many solvers take minutes/hours/days to execute a single run
- And: three-way interactions?
- Consequence: a different strategy is needed

2.10. Exercise

2.10.1. Adding Paint Weight

- Paint weight is not considered.
- Add Paint Weight W_p to formula (the updated formula is shown below) and update the functions and plots in the notebook.

$$W = 0.036 S_W^{0.758} \times W_{fw}^{0.0035} \times \left(\frac{A}{\cos^2 \Lambda}\right)^{0.6} \times q^{0.006} \times \lambda^{0.04}$$
$$\times \left(\frac{100 R_{tc}}{\cos \Lambda}\right)^{-0.3} \times (N_z W_{dg})^{0.49} + S_w W_p$$

2.11. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

3. Introduction to `scipy.optimize`

SciPy provides algorithms for optimization, integration, interpolation, eigenvalue problems, algebraic equations, differential equations, statistics and many other classes of problems. SciPy is a collection of mathematical algorithms and convenience functions built on NumPy. It adds significant power to Python by providing the user with high-level commands and classes for manipulating and visualizing data.

SciPy optimize provides functions for minimizing (or maximizing) objective functions, possibly subject to constraints. It includes solvers for nonlinear problems (with support for both local and global optimization algorithms), linear programing, constrained and nonlinear least-squares, root finding, and curve fitting.

In this notebook, we will learn how to use the `scipy.optimize` module to solve optimization problems. See: https://docs.scipy.org/doc/scipy/tutorial/optimize.html

> **Note**
>
> - This content is based on information from the scipy.optimize package.
> - The `scipy.optimize` package provides several commonly used optimization algorithms. A detailed listing is available in `scipy.optimize` (can also be found by `help(scipy.optimize)`).

Common functions and objects, shared across different SciPy optimize solvers, are shown in Table 3.1.

Table 3.1.: Common functions and objects, shared across different SciPy optimize solvers

Function or Object	Description
show_options([solver, method, disp])	Show documentation for additional options of optimization solvers.
OptimizeResult	Represents the optimization result.
OptimizeWarning	Warning issued by solvers.

We will introduce unconstrained minimization of multivariate scalar functions in this chapter. The minimize function provides a common interface to unconstrained and constrained minimization algorithms for multivariate scalar functions in `scipy.optimize`.

To demonstrate the minimization function, consider the problem of minimizing the Rosenbrock function of N variables:

$$f(J) = \sum_{i=1}^{N-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2$$

The minimum value of this function is 0, which is achieved when (x_i = 1).

Note that the Rosenbrock function and its derivatives are included in `scipy.optimize`. The implementations shown in the following sections provide examples of how to define an objective function as well as its Jacobian and Hessian functions. Objective functions in `scipy.optimize` expect a numpy array as their first parameter, which is to be optimized and must return a float value. The exact calling signature must be `f(x, *args)`, where `x` represents a numpy array, and `args` is a tuple of additional arguments supplied to the objective function.

3.1. Derivative-free Optimization Algorithms

Section 3.1.1 and Section 3.1.2 present two approaches that do not need gradient information to find the minimum. They use function evaluations to find the minimum.

3.1.1. Nelder-Mead Simplex Algorithm

The Nelder Mead is a simple local optimization algorithm. It requires only function evaluations and is a good choice for simple minimization problems. However, because it does not use any gradient evaluations, it may take longer to find the minimum. It can be devided into the following steps:

1. Initialize the simplex
2. Evaluate the function at each vertex of the simplex
3. Order the vertices by function value
4. **Reflect** the worst point through the centroid of the remaining points
5. If the reflected point is better than the second worst, replace the worst point with the reflected point
6. If the reflected point is worse than the worst point, try **contracting** the simplex
7. If the reflected point is better than the best point, try **expanding** the simplex
8. If none of the above steps improve the simplex, **shrink** the simplex towards the best point
9. Check for convergence

`method='Nelder-Mead'`: In the example below, the `minimize` routine is used with the *Nelder-Mead* simplex algorithm (selected through the `method` parameter):

```python
import numpy as np
from scipy.optimize import minimize

def rosen(x):
    """The Rosenbrock function"""
    return sum(100.0 * (x[1:] - x[:-1]**2.0)**2.0 + (1 - x[:-1])**2.0)

x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize(rosen, x0, method='nelder-mead',
               options={'xatol': 1e-8, 'disp': True})

print(res.x)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 339
         Function evaluations: 571
[1. 1. 1. 1. 1.]
```

The simplex algorithm is probably the simplest way to minimize a well-behaved function. It requires only function evaluations and is a good choice for simple minimization problems. However, because it does not use any gradient evaluations, it may take longer to find the minimum.

3.1.2. Powell's Method

Another optimization algorithm that needs only function calls to find the minimum is *Powell*'s method, which can be selected by setting the `method` parameter to `'powell'` in the `minimize` function. This algorithm consists of a conjugate direction method. It performs sequential one-dimensional minimizations along each vector of the directions set, which is updated at each iteration of the main minimization loop. It can be described by the following steps:

1. Initialization
2. Minimization along each direction
3. Create conjugate direction
4. Line search along the conjugate direction
5. Check for convergence

Example 3.1. To demonstrate how to supply additional arguments to an objective function, let's consider minimizing the Rosenbrock function with an additional scaling factor a and an offset b:

$$f(J, a, b) = \sum_{i=1}^{N-1} a(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 + b$$

You can achieve this using the `minimize` routine with the example parameters $a = 0.5$ and $b = 1$:

```
def rosen_with_args(x, a, b):
    """The Rosenbrock function with additional arguments"""
    return sum(a * (x[1:] - x[:-1]**2.0)**2.0 + (1 - x[:-1])**2.0) + b

x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize(rosen_with_args, x0, method='nelder-mead',
               args=(0.5, 1.), options={'xatol': 1e-8, 'disp': True})

print(res.x)
```

```
Optimization terminated successfully.
         Current function value: 1.000000
         Iterations: 319
         Function evaluations: 525
[1.         1.         1.         1.         0.99999999]
```

As an alternative to using the `args` parameter of `minimize`, you can wrap the objective function in a new function that accepts only `x`. This approach is also useful when it is necessary to pass additional parameters to the objective function as keyword arguments.

```
def rosen_with_args(x, a, *, b):  # b is a keyword-only argument
    return sum(a * (x[1:] - x[:-1]**2.0)**2.0 + (1 - x[:-1])**2.0) + b

def wrapped_rosen_without_args(x):
    return rosen_with_args(x, 0.5, b=1.)  # pass in `a` and `b`

x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize(wrapped_rosen_without_args, x0, method='nelder-mead',
               options={'xatol': 1e-8,})

print(res.x)
```

```
[1.         1.         1.         1.         0.99999999]
```

Another alternative is to use `functools.partial`.

```
from functools import partial

partial_rosen = partial(rosen_with_args, a=0.5, b=1.)
res = minimize(partial_rosen, x0, method='nelder-mead',
               options={'xatol': 1e-8,})

print(res.x)
```

```
[1.         1.         1.         1.         0.99999999]
```

3.2. Gradient-based Optimization Algorithms

3.2.1. An Introductory Example: Broyden-Fletcher-Goldfarb-Shanno Algorithm (BFGS)

This section introduces an optimization algorithm that uses gradient information to find the minimum. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm (selected by setting `method='BFGS'`) is an optimization algorithm that aims to converge quickly to the solution. This algorithm uses the gradient of the objective function. If the gradient is not provided by the user, it is estimated using first-differences. The BFGS method typically requires fewer function calls compared to the simplex algorithm, even when the gradient needs to be estimated.

Example 3.2 (BFGS)**.** To demonstrate the BFGS algorithm, let's use the Rosenbrock function again. The gradient of the Rosenbrock function is a vector described by the following mathematical expression:

$$\frac{\partial f}{\partial x_j} = \sum_{i=1}^{N} 200(x_i - x_{i-1}^2)(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}) - 2(1 - x_{i-1})\delta_{i-1,j} \tag{3.1}$$

$$= 200(x_j - x_{j-1}^2) - 400x_j(x_{j+1} - x_j^2) - 2(1 - x_j) \tag{3.2}$$

This expression is valid for interior derivatives, but special cases are:

$$\frac{\partial f}{\partial x_0} = -400x_0(x_1 - x_0^2) - 2(1 - x_0)$$

$$\frac{\partial f}{\partial x_{N-1}} = 200(x_{N-1} - x_{N-2}^2)$$

Here's a Python function that computes this gradient:

```
def rosen_der(x):
    xm = x[1:-1]
    xm_m1 = x[:-2]
    xm_p1 = x[2:]
    der = np.zeros_like(x)
    der[1:-1] = 200*(xm-xm_m1**2) - 400*(xm_p1 - xm**2)*xm - 2*(1-xm)
    der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
    der[-1] = 200*(x[-1]-x[-2]**2)
    return der
```

You can specify this gradient information in the minimize function using the jac parameter as illustrated below:

```
res = minimize(rosen, x0, method='BFGS', jac=rosen_der,
               options={'disp': True})

print(res.x)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 25
         Function evaluations: 30
         Gradient evaluations: 30
[1.00000004 1.0000001  1.00000021 1.00000044 1.00000092]
```

3.2.2. Background and Basics for Gradient-based Optimization

3.2.3. Gradient

The gradient $\nabla f(J)$ for a scalar function $f(J)$ with n different variables is defined by its partial derivatives:

$$\nabla f(J) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n}\right]$$

3.2.4. Jacobian Matrix

The Jacobian matrix $J(J)$ for a vector-valued function $F(J) = [f_1(J), f_2(J), \dots, f_m(J)]$ is defined as:

$$J(J) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

It consists of the first order partial derivatives and gives therefore an overview about the gradients of a vector valued function.

Example 3.3 (acobian matrix)**.** Consider a vector-valued function $f : \mathbb{R}^2 \to \mathbb{R}^3$ defined as follows:

$$f(J) = \begin{bmatrix} x_1^2 + 2x_2 \\ 3x_1 - \sin(x_2) \\ e^{x_1+x_2} \end{bmatrix}$$

Let's compute the partial derivatives and construct the Jacobian matrix:

$\frac{\partial f_1}{\partial x_1} = 2x_1, \quad \frac{\partial f_1}{\partial x_2} = 2$

$\frac{\partial f_2}{\partial x_1} = 3, \quad \frac{\partial f_2}{\partial x_2} = -\cos(x_2)$

$\frac{\partial f_3}{\partial x_1} = e^{x_1+x_2}, \quad \frac{\partial f_3}{\partial x_2} = e^{x_1+x_2}$

So, the Jacobian matrix is:

$$J(J) = \begin{bmatrix} 2x_1 & 2 \\ 3 & -\cos(x_2) \\ e^{x_1+x_2} & e^{x_1+x_2} \end{bmatrix}$$

This Jacobian matrix provides information about how small changes in the input variables x_1 and x_2 affect the corresponding changes in each component of the output vector.

3.2.5. Hessian Matrix

The Hessian matrix $H(J)$ for a scalar function $f(J)$ is defined as:

$$H(J) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

The Hessian matrix consists of the second order derivatives of the function. It provides information about the local curvature of the function with respect to changes in the input variables.

Example 3.4 (Hessian matrix)**.** Consider a scalar-valued function:

$$f(J) = x_1^2 + 2x_2^2 + \sin(x_1 x_2)$$

The Hessian matrix of this scalar-valued function is the matrix of its second-order partial derivatives with respect to the input variables:

$$H(J) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix}$$

Let's compute the second-order partial derivatives and construct the Hessian matrix:

$$\frac{\partial^2 f}{\partial x_1^2} = 2 + \cos(x_1 x_2) x_2^2 \tag{3.3}$$

$$\frac{\partial^2 f}{\partial x_1 \partial x_2} = 2x_1 x_2 \cos(x_1 x_2) - \sin(x_1 x_2) \tag{3.4}$$

$$\frac{\partial^2 f}{\partial x_2 \partial x_1} = 2x_1 x_2 \cos(x_1 x_2) - \sin(x_1 x_2) \tag{3.5}$$

$$\frac{\partial^2 f}{\partial x_2^2} = 4x_2^2 + \cos(x_1 x_2) x_1^2 \tag{3.6}$$

So, the Hessian matrix is:

$$H(J) = \begin{bmatrix} 2 + \cos(x_1 x_2) x_2^2 & 2x_1 x_2 \cos(x_1 x_2) - \sin(x_1 x_2) \\ 2x_1 x_2 \cos(x_1 x_2) - \sin(x_1 x_2) & 4x_2^2 + \cos(x_1 x_2) x_1^2 \end{bmatrix}$$

3.2.6. Gradient Descent

In optimization, the goal is to find the minimum or maximum of a function. Gradient-based optimization methods utilize information about the gradient (or derivative) of the function to guide the search for the optimal solution. This is particularly useful when dealing with complex, high-dimensional functions where an exhaustive search is impractical.

The gradient descent method can be divided in the following steps:

- **Initialize:** start with an initial guess for the parameters of the function to be optimized.
- **Compute Gradient:** Calculate the gradient (partial derivatives) of the function with respect to each parameter at the current point. The gradient indicates the direction of the steepest increase in the function.

- **Update Parameters:** Adjust the parameters in the opposite direction of the gradient, scaled by a learning rate. This step aims to move towards the minimum of the function:
  - $x_{k+1} = x_k - \alpha \times \nabla f(x_k)$
  - x_x is current parameter vector or point in the parameter space.
  - α is the learning rate, a positive scalar that determines the step size in each iteration.
  - $\nabla f(x)$ is the gradient of the objective function.
- **Iterate:** Repeat the above steps until convergence or a predefined number of iterations. Convergence is typically determined when the change in the function value or parameters becomes negligible.

Example 3.5 (Gradient Descent)**.** We consider a simple quadratic function as an example:

$$f(x) = x^2 + 4x + y^2 + 2y + 4.$$

We'll use gradient descent to find the minimum of this function.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Define the quadratic function
def quadratic_function(x, y):
    return x**2 + 4*x + y**2 + 2*y + 4

# Define the gradient of the quadratic function
def gradient_quadratic_function(x, y):
    grad_x = 2*x + 4
    grad_y = 2*y + 2
    return np.array([grad_x, grad_y])

# Gradient Descent for optimization in 2D
def gradient_descent(initial_point, learning_rate, num_iterations):
    points = [np.array(initial_point)]
    for _ in range(num_iterations):
        current_point = points[-1]
        gradient = gradient_quadratic_function(*current_point)
        new_point = current_point - learning_rate * gradient
        points.append(new_point)
    return points

# Visualization of optimization process with 3D surface and consistent arrow sizes
```

```
def plot_optimization_process_3d_consistent_arrows(points):
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')

    x_vals = np.linspace(-10, 2, 100)
    y_vals = np.linspace(-10, 2, 100)
    X, Y = np.meshgrid(x_vals, y_vals)
    Z = quadratic_function(X, Y)

    ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.6)
    ax.scatter(*zip(*points), [quadratic_function(*p) for p in points], c='red', label

    for i in range(len(points) - 1):
        x, y = points[i]
        dx, dy = points[i + 1] - points[i]
        dz = quadratic_function(*(points[i + 1])) - quadratic_function(*points[i])
        gradient_length = 0.5

        ax.quiver(x, y, quadratic_function(*points[i]), dx, dy, dz, color='blue', leng

    ax.set_title('Gradient-Based Optimization with 2D Quadratic Function')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('f(x, y)')
    ax.legend()
    plt.show()

# Initial guess and parameters
initial_guess = [-9.0, -9.0]
learning_rate = 0.2
num_iterations = 10

# Run gradient descent in 2D and visualize the optimization process with 3D surface ar
trajectory = gradient_descent(initial_guess, learning_rate, num_iterations)
plot_optimization_process_3d_consistent_arrows(trajectory)
```

3.2.7. Newton Method

Initialization: Start with an initial guess for the optimal solution: x_0.

Iteration: Repeat the following three steps until convergence or a predefined stopping criterion is met:

1. Calculate the gradient (∇) and the Hessian matrix (∇^2) of the objective function at the current point:

$$\nabla f(x_k) \quad \text{and} \quad \nabla^2 f(x_k)$$

2. Update the current solution using the Newton-Raphson update formula

$$x_{k+1} = x_k - [\nabla^2 f(x_k)]^{-1} \nabla f(x_k),$$

where

```
* $\nabla f(x_k)$ is the gradient (first derivative) of the objective function wi
```

- $\nabla^2 f(x_k)$: The Hessian matrix (second derivative) of the objective function with respect to x, evaluated at the current solution x_k.
- x_k: The current solution or point in the optimization process.
- $\nabla^2 f(x_k)]^{-1}$: The inverse of the Hessian matrix at the current point, representing the approximation of the curvature of the objective function.
- x_{k+1}: The updated solution or point after applying the Newton-Raphson update.

3. Check for convergence.

Example 3.6 (Newton Method)**.** We want to optimize the Rosenbrock function and use the Hessian and the Jacobian (which is equal to the gradient vector for scalar objective function) to the `minimize` function.

```
def rosenbrock(x):
    return 100 * (x[1] - x[0]**2)**2 + (1 - x[0])**2

def rosenbrock_gradient(x):
    dfdx0 = -400 * x[0] * (x[1] - x[0]**2) - 2 * (1 - x[0])
    dfdx1 = 200 * (x[1] - x[0]**2)
    return np.array([dfdx0, dfdx1])

def rosenbrock_hessian(x):
    d2fdx0 = 1200 * x[0]**2 - 400 * x[1] + 2
    d2fdx1 = -400 * x[0]
    return np.array([[d2fdx0, d2fdx1], [d2fdx1, 200]])

def classical_newton_optimization_2d(initial_guess, tol=1e-6, max_iter=100):
    x = initial_guess.copy()

    for i in range(max_iter):
        gradient = rosenbrock_gradient(x)
        hessian = rosenbrock_hessian(x)

        # Solve the linear system H * d = -g for d
        d = np.linalg.solve(hessian, -gradient)

        # Update x
```

```
        x += d

        # Check for convergence
        if np.linalg.norm(gradient, ord=np.inf) < tol:
            break

    return x

# Initial guess
initial_guess_2d = np.array([0.0, 0.0])

# Run classical Newton optimization for the 2D Rosenbrock function
result_2d = classical_newton_optimization_2d(initial_guess_2d)

# Print the result
print("Optimal solution:", result_2d)
print("Objective value:", rosenbrock(result_2d))
```

```
Optimal solution: [1. 1.]
Objective value: 0.0
```

3.2.8. BFGS-Algorithm

BFGS is an optimization algorithm designed for unconstrained optimization problems. It belongs to the class of quasi-Newton methods and is known for its efficiency in finding the minimum of a smooth, unconstrained objective function.

3.2.9. Procedure:

1. **Initialization:**
   - Start with an initial guess for the parameters of the objective function.
   - Initialize an approximation of the Hessian matrix (inverse) denoted by H.

2. **Iterative Update:**
   - At each iteration, compute the gradient vector at the current point.
   - Update the parameters using the BFGS update formula, which involves the inverse Hessian matrix approximation, the gradient, and the difference in parameter vectors between successive iterations:

$$x_{k+1} = x_k - H_k^{-1} \nabla f(x_k).$$

- Update the inverse Hessian approximation using the BFGS update formula for the inverse Hessian.

$$H_{k+1} = H_k + \frac{\Delta x_k \Delta x_k^T}{\Delta x_k^T \Delta g_k} - \frac{H_k g_k g_k^T H_k}{g_k^T H_k g_k},$$

  where:
- x_k and x_{k+1} are the parameter vectors at the current and updated iterations, respectively.
- $\nabla f(x_k)$ is the gradient vector at the current iteration.
- $\Delta x_k = x_{k+1} - x_k$ is the change in parameter vectors.
- $\Delta g_k = \nabla f(x_{k+1}) - \nabla f(x_k)$ is the change in gradient vectors.

3. **Convergence:**

   - Repeat the iterative update until the optimization converges. Convergence is typically determined by reaching a sufficiently low gradient or parameter change.

Example 3.7 (BFGS for Rosenbrock)**.**

```python
import numpy as np
from scipy.optimize import minimize

# Define the 2D Rosenbrock function
def rosenbrock(x):
    return (1 - x[0])**2 + 100 * (x[1] - x[0]**2)**2

# Initial guess
initial_guess = np.array([0.0, 0.0])

# Minimize the Rosenbrock function using BFGS
minimize(rosenbrock, initial_guess, method='BFGS')
```

```
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 2.8440052847381483e-11
        x: [ 1.000e+00  1.000e+00]
      nit: 19
      jac: [ 3.987e-06 -2.844e-06]
 hess_inv: [[ 4.948e-01  9.896e-01]
            [ 9.896e-01  1.984e+00]]
     nfev: 72
     njev: 24
```

3.2.10. Visualization BFGS for Rosenbrock

A visualization of the BFGS search process on Rosenbrock's function can be found here: https://upload.wikimedia.org/wikipedia/de/f/ff/Rosenbrock-bfgs-animation.gif

3.3. Global Optimization

Global optimization aims to find the global minimum of a function within given bounds, in the presence of potentially many local minima. Typically, global minimizers efficiently search the parameter space, while using a local minimizer (e.g., minimize) under the hood.

3.3.1. Local vs Global Optimization

3.3.1.1. Local Optimizater:

- Seeks the optimum in a **specific region** of the search space
- Tends to **exploit** the local environment, to find solutions in the immediate area
- Highly **sensitive to initial conditions**; may converge to different local optima based on the starting point
- Often **computationally efficient for low-dimensional problems** but may struggle with high-dimensional or complex search spaces
- Commonly used in situations where the objective is to refine and improve existing solutions

3.3.1.2. Global Optimizer:

- Explores the **entire search space** to find the global optimum
- Emphasize **exploration over exploitation**, aiming to search broadly and avoid premature convergence to local optima
- Aim to **mitigate the risk of premature convergence** to local optima by employing strategies for global exploration
- **Less sensitive to initial conditions**, designed to navigate diverse regions of the search space
- Equipped to handle **high-dimensional** and **complex** problems, though computational demands may vary depending on the specific algorithm
- Preferred for applications where a comprehensive search of the solution space is crucial, such as in parameter tuning, machine learning, and complex engineering design

Example 3.8 (Global Optimizers in SciPy)**.** SciPy contains a number of good global optimizers. Here, we'll use those on the same objective function, namely the (aptly named) eggholder function:

```
def eggholder(x):
    return (-(x[1] + 47) * np.sin(np.sqrt(abs(x[0]/2 + (x[1]  + 47))))
            -x[0] * np.sin(np.sqrt(abs(x[0] - (x[1]  + 47)))))

bounds = [(-512, 512), (-512, 512)]
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

x = np.arange(-512, 513)
y = np.arange(-512, 513)
xgrid, ygrid = np.meshgrid(x, y)
xy = np.stack([xgrid, ygrid])

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.view_init(45, -45)
ax.plot_surface(xgrid, ygrid, eggholder(xy), cmap='terrain')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('eggholder(x, y)')
plt.show()
```

We now use the global optimizers to obtain the minimum and the function value at the minimum. We'll store the results in a dictionary so we can compare different optimization results later.

```
from scipy import optimize
results = dict()
results['shgo'] = optimize.shgo(eggholder, bounds)
results['shgo']
```

```
 message: Optimization terminated successfully.
 success: True
     fun: -935.3379515605789
    funl: [-9.353e+02]
       x: [ 4.395e+02  4.540e+02]
      xl: [[ 4.395e+02  4.540e+02]]
     nit: 1
    nfev: 45
   nlfev: 40
   nljev: 10
   nlhev: 0
```

```
results['DA'] = optimize.dual_annealing(eggholder, bounds)
results['DA']
```

```
 message: ['Maximum number of iteration reached']
```

```
 success: True
  status: 0
     fun: -894.5789003904578
       x: [-4.657e+02  3.857e+02]
     nit: 1000
    nfev: 4025
    njev: 8
    nhev: 0
```

All optimizers return an `OptimizeResult`, which in addition to the solution contains information on the number of function evaluations, whether the optimization was successful, and more. For brevity, we won't show the full output of the other optimizers:

```
results['DE'] = optimize.differential_evolution(eggholder, bounds)
results['DE']
```

```
             message: Optimization terminated successfully.
             success: True
                 fun: -959.6406627208469
                   x: [ 5.120e+02  4.042e+02]
                 nit: 38
                nfev: 1185
          population: [[ 5.118e+02  4.040e+02]
                       [ 5.099e+02  3.994e+02]
                       ...
                       [ 5.088e+02  3.998e+02]
                       [ 5.118e+02  4.070e+02]]
 population_energies: [-9.596e+02 -9.423e+02 ... -9.445e+02 -9.496e+02]
                 jac: [-3.386e+00 -2.274e-05]
```

`shgo` has a second method, which returns all local minima rather than only what it thinks is the global minimum:

```
results['shgo_sobol'] = optimize.shgo(eggholder, bounds, n=200, iters=5,
                                      sampling_method='sobol')
results['shgo_sobol']
```

```
 message: Optimization terminated successfully.
 success: True
     fun: -959.640662720831
    funl: [-9.596e+02 -9.353e+02 ... -6.591e+01 -6.387e+01]
       x: [ 5.120e+02  4.042e+02]
      xl: [[ 5.120e+02  4.042e+02]
```

```
           [ 4.395e+02  4.540e+02]
           ...
           [ 3.165e+01 -8.523e+01]
           [ 5.865e+01 -5.441e+01]]
     nit: 5
    nfev: 3529
   nlfev: 2327
   nljev: 634
   nlhev: 0
```

We'll now plot all found minima on a heatmap of the function:

```
fig = plt.figure()
ax = fig.add_subplot(111)
im = ax.imshow(eggholder(xy), interpolation='bilinear', origin='lower',
               cmap='gray')
ax.set_xlabel('x')
ax.set_ylabel('y')

def plot_point(res, marker='o', color=None):
    ax.plot(512+res.x[0], 512+res.x[1], marker=marker, color=color, ms=10)

plot_point(results['DE'], color='c')  # differential_evolution - cyan
plot_point(results['DA'], color='w')  # dual_annealing.        - white

# SHGO produces multiple minima, plot them all (with a smaller marker size)
plot_point(results['shgo'], color='r', marker='+')
plot_point(results['shgo_sobol'], color='r', marker='x')
for i in range(results['shgo_sobol'].xl.shape[0]):
    ax.plot(512 + results['shgo_sobol'].xl[i, 0],
            512 + results['shgo_sobol'].xl[i, 1],
            'ro', ms=2)

ax.set_xlim([-4, 514*2])
ax.set_ylim([-4, 514*2])
plt.show()
```

3.3.2. Dual Annealing Optimization

This function implements the Dual-Annealing optimization, which is a variant of the famous simulated annealing optimization.

Simulated Annealing is a **probabilistic** optimization algorithm inspired by the annealing process in metallurgy. The algorithm is designed to find a good or optimal **global** solution to a problem by exploring the solution space in a controlled and adaptive manner.

> **ℹ Annealing in Metallurgy**
>
> Simulated Annealing draws inspiration from the physical process of annealing in metallurgy. Just as metals are gradually cooled to achieve a more stable state, Simulated Annealing uses a similar approach to explore solution spaces in the digital world.

Heating Phase: In metallurgy, a metal is initially heated to a high temperature. At this elevated temperature, the atoms or molecules in the material become more energetic and chaotic, allowing the material to overcome energy barriers and defects.

Analogy Simulated Annealing (Exploration Phase): In Simulated Annealing, the algorithm starts with a high "temperature," which encourages exploration of the

solution space. At this stage, the algorithm is more likely to accept solutions that are worse than the current one, allowing it to escape local optima and explore a broader region of the solution space.

Cooling Phase: The material is then gradually cooled at a controlled rate. As the temperature decreases, the atoms or molecules start to settle into more ordered and stable arrangements. The slow cooling rate is crucial to avoid the formation of defects and to ensure the material reaches a well-organized state.

Analogy Simulated Annealing (Exploitation Phase): As the algorithm progresses, the temperature is gradually reduced over time according to a cooling schedule. This reduction simulates the cooling process in metallurgy. With lower temperatures, the algorithm becomes more selective and tends to accept only better solutions, focusing on refining and exploiting the promising regions discovered during the exploration phase.

3.3.2.1. Key Concepts

Temperature: The temperature is a parameter that controls the likelihood of accepting worse solutions. We start with a high temperature, allowing the algorithm to explore the solution space braodly. The temperature decreases with the iterations of the algorithm.

Cooling Schedule: The temperature parameter is reduced according to this schedule. The analogy to the annealing of metals: a slower cooling rate allows the material to reach a more stable state.

Neighborhood Exploration: At each iteration, the algorithm explores the neighborhood of the current solution. The neighborhood is defined by small perturbations or changes to the current solution.

Acceptance Probability: The algorithm evaluates the objective function for the new solution in the neighborhood. If the new solution is better, it is accepted. If the new solution is worse, it may still be accepted with a certain probability. This probability is determined by both the difference in objective function values and the current temperature.

For minimization: If:

$$f(x_t) > f(x_{t+1})$$

Then:

$$P(accept_new_point) = 1$$

If:

$$f(x_t) < f(x_{t+1})$$

Then:

$$P(accept_new_point) = e^{-(\frac{f(x_{t+1})-f(x_t)}{Tt})}$$

Termination Criterion: The algorithm continues iterations until a termination condition is met. This could be a fixed number of iterations, reaching a specific temperature threshold, or achieving a satisfactory solution.

3.3.2.2. Steps

1. Initialization: Set an initial temperature (T_0) and an initial solution ($f(x_0)$). The temperature is typically set high initially to encourage exploration.

2. Generate a Neighbor: Perturb the current solution to generate a neighboring solution. The perturbation can be random or follow a specific strategy.

3. Evaluate the Neighbor: Evaluate the objective function for the new solution in the neighborhood.

4. Accept or Reject the Neighbor: + If the new solution is better (lower cost for minimization problems or higher for maximization problems), accept it as the new current solution. + If the new solution is worse, accept it with a probability determined by an acceptance probability function as mentioned above. The probability is influenced by the difference in objective function values and the current temperature.

5. Cooling: Reduce the temperature according to a cooling schedule. The cooling schedule defines how fast the temperature decreases over time. Common cooling schedules include exponential or linear decay.

6. Termination Criterion: Repeat the iterations (2-5) until a termination condition is met. This could be a fixed number of iterations, reaching a specific temperature threshold, or achieving a satisfactory solution.

3.3.2.3. Scipy Implementation of the Dual Annealing Algorithm

In Scipy, we utilize the Dual Annealing optimizer, an extension of the simulated annealing algorithm that is versatile for both discrete and continuous problems.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import dual_annealing

def rastrigin_function(x):
    return 20 + x[0]**2 - 10 * np.cos(2 * np.pi * x[0]) + x[1]**2 - 10 * np.cos(2 * np

# Define the Rastrigin function for visualization
def rastrigin_visualization(x, y):
    return 20 + x**2 - 10 * np.cos(2 * np.pi * x) + y**2 - 10 * np.cos(2 * np.pi * y)
```

```
# Create a meshgrid for visualization
x_vals = np.linspace(-10, 10, 100)
y_vals = np.linspace(-10, 10, 100)
x_mesh, y_mesh = np.meshgrid(x_vals, y_vals)
z_mesh = rastrigin_visualization(x_mesh, y_mesh)

# Visualize the Rastrigin function
plt.figure(figsize=(10, 8))
contour = plt.contour(x_mesh, y_mesh, z_mesh, levels=50, cmap='viridis')
plt.colorbar(contour, label='Rastrigin Function Value')
plt.title('Visualization of the 2D Rastrigin Function')

# Optimize the Rastrigin function using dual annealing
result = dual_annealing(func = rastrigin_function,
                        x0=[5.0,3.0],                       #Initial Guess
                        bounds= [(-10, 10), (-10, 10)],
                        initial_temp = 5230,                #Intial Value for temperature
                        restart_temp_ratio = 2e-05,         #Temperature schedule
                        seed=42)

# Plot the optimized point
optimal_x, optimal_y = result.x
plt.plot(optimal_x, optimal_y, 'ro', label='Optimal Point')

# Set labels and legend
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()

# Show the plot
plt.show()

# Display the optimization result
print("Optimal parameters:", result.x)
print("Minimum value of the Rastrigin function:", result.fun)
```

```
Optimal parameters: [-4.60133247e-09 -4.31928660e-09]
Minimum value of the Rastrigin function: 7.105427357601002e-15
```

3.3.3. Differential Evolution

Differential Evolution is an algorithm used for finding the global minimum of multivariate functions. It is stochastic in nature (does not use gradient methods), and can search large areas of candidate space, but often requires larger numbers of function evaluations than conventional gradient based techniques.

Differential Evolution (DE) is a versatile and global optimization algorithm inspired by natural selection and evolutionary processes. Introduced by Storn and Price in 1997, DE mimics the survival-of-the-fittest principle by evolving a population of candidate solutions through iterative mutation, crossover, and selection operations. This nature-inspired approach enables DE to efficiently explore complex and non-linear solution spaces, making it a widely adopted optimization technique in diverse fields such as engineering, finance, and machine learning.

3.3.4. Procedure

The procedure boils down to the following steps:

1. **Initialization:**
   - Create a population of candidate solutions randomly within the specified search space.
2. **Mutation:**
   - For each individual in the population, select three distinct individuals (vectors) randomly.
   - Generate a mutant vector `V` by combining these three vectors with a scaling factor.
3. **Crossover:**
   - Perform the crossover operation between the target vector `U` and the mutant vector `V`. Information from both vectors is used to create a trial vector `U´`

> Cross-Over Strategies in DE
>
> - There are several crossover strategies in the literature. Two examples are:
>
> **Binominal Crossover:**
> In this strategy, each component of the trial vector is selected from the mutant vector with a probability equal to the crossover rate (CR). This means that each element of the trial vector has an independent probability of being replaced by the corresponding element of the mutant vector.
>
> $$U_i' = \begin{cases} V_i, & \text{if a random number } \sim U(0,1) \leq CR \text{ (Crossover Rate)} \\ U_i, & \text{otherwise} \end{cases}$$
>
> **Exponential Crossover:**
> In exponential crossover, the trial vector is constructed by selecting a random starting point and copying elements from the mutant vector with a certain probability. The probability decreases exponentially with the distance from the starting point. This strategy introduces a correlation between neighboring elements in the trial vector.

4. **Selection:**
   - Evaluate the fitness of the trial vector obtained from the crossover.
   - Replace the target vector with the trial vector if its fitness is better.
5. **Termination:**
   - Repeat the mutation, crossover, and selection steps for a predefined number of generations or until convergence criteria are met.

6. **Result:**
   - The algorithm returns the best-found solution after the specified number of iterations.

The key parameters in DE include the population size, crossover probability, and the scaling factor. Tweak these parameters based on the characteristics of the optimization problem for optimal performance.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

# Define the Rastrigin function
def rastrigin(x):
    A = 10
    return A * len(x) + sum([(xi**2 - A * np.cos(2 * np.pi * xi)) for xi in x])

# Create a grid for visualization
x_vals = np.linspace(-5.12, 5.12, 100)
y_vals = np.linspace(-5.12, 5.12, 100)
X, Y = np.meshgrid(x_vals, y_vals)
Z = rastrigin(np.vstack([X.ravel(), Y.ravel()]))

# Reshape Z to match the shape of X and Y
Z = Z.reshape(X.shape)

# Plot the Rastrigin function
plt.contour(X, Y, Z, levels=50, cmap='viridis', label='Rastrigin Function')

# Initial guess (starting point for the optimization)
initial_guess = (4,3,4,2)

# Define the bounds for each variable in the Rastrigin function
bounds = [(-5.12, 5.12)] * 4  # 4D problem, each variable has bounds (-5.12, 5.12)

# Run the minimize function
result = minimize(rastrigin, initial_guess, bounds=bounds, method='L-BFGS-B')

# Extract the optimal solution
optimal_solution = result.x

# Plot the optimal solution
plt.scatter(optimal_solution[0], optimal_solution[1], color='red', marker='x', label=

# Add labels and legend
```

```
plt.title('Optimization of Rastrigin Function with Minimize')
plt.xlabel('Variable 1')
plt.ylabel('Variable 2')
plt.legend()

# Show the plot
plt.show()

# Print the optimization result
print("Optimal Solution:", optimal_solution)
print("Optimal Objective Value:", result.fun)
```

```
Optimal Solution: [-2.52869119e-08 -2.07795060e-08 -2.52869119e-08 -1.62721002e-08]
Optimal Objective Value: 3.907985046680551e-13
```

3.3.5. Other global optimization algorithms

3.3.6. DIRECT

DIviding RECTangles (DIRECT) is a deterministic global optimization algorithm capable of minimizing a black box function with its variables subject to lower and upper

bound constraints by sampling potential solutions in the search space

3.3.7. SHGO

SHGO stands for "simplicial homology global optimization". It is considered appropriate for solving general purpose NLP and blackbox optimization problems to global optimality (low-dimensional problems).

3.3.8. Basin-hopping

Basin-hopping is a two-phase method that combines a global stepping algorithm with local minimization at each step. Designed to mimic the natural process of energy minimization of clusters of atoms, it works well for similar problems with "funnel-like, but rugged" energy landscapes

3.4. Project: One-Mass Oscillator Optimization

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import minimize
```

3.4.1. Introduction

In this project, you will apply various optimization algorithms to fit a one-mass oscillator model to real-world data. The objective is to minimize the sum of the squared residuals between the model predictions and the observed amplitudes of a one-mass oscillator system across different frequencies.

3.4.2. One-Mass Oscillator Model

The one-mass oscillator is characterized by the following equation, representing the amplitudes of the system:

$$V(\omega) = \frac{F}{\sqrt{(1-\nu^2)^2 + 4D^2\nu^2}}$$

Here, ω represents the angular frequency of the system, ν is the ratio of the excitation frequency to the natural frequency, i.e.,

$$\nu = \frac{\omega_{\text{err}}}{\omega_{\text{eig}}},$$

D is the damping ratio, and F is the force applied to the system.

The goal of the project is to determine the optimal values for the parameters ω_{eig}, D, and F that result in the best fit of the one-mass oscillator model to the observed amplitudes.

3.4.3. The Real-World Data

There are two different measurements. `J` represents the measured frequencies, and `N` represents the measured amplitudes.

```
df1 = pd.read_pickle("./data/Hcf.d/df1.pkl")
df2 = pd.read_pickle("./data/Hcf.d/df2.pkl")
df1.describe()
```

	J	N
count	33.000000	33.000000
mean	8148.750252	10.430887
std	6.870023	2.846469
min	8137.649210	4.698761
25%	8143.799766	8.319253
50%	8146.942295	10.152119
75%	8153.934051	13.407260
max	8162.504002	14.382749

```
df1.head()
```

	J	N
14999	8162.504002	5.527511
15011	8156.384831	7.359789
15016	8159.199238	6.532958
15020	8159.200889	5.895933
15025	8153.934051	9.326749

```
# plot the data, i.e., the measured amplitudes as a function of the measured frequenci
plt.scatter(df1["J"], df1["N"], color="black", label="Spektralpunkte", zorder=5, s=10)
plt.xlabel("Frequency [Hz]")
plt.ylabel("Amplitude")
plt.show()
```

Note: Low amplitudes distort the fit and are negligible therefore we define a lower threshold for N.

```
threshold = 0.4
df1.sort_values("N")
max_N = max(df1["N"])
df1 = df1[df1["N"]>=threshold*max_N]
```

We extract the frequency value for maximum value of the amplitude. This serves as the initial value for one decision variable.

```
df_max=df1[df1["N"]==max(df1["N"])]
initial_Oeig = df_max["J"].values[0]
max_N = df_max["N"].values[0]
```

We also have to define the other two initial guesses for the damping ratio and the force, e.g.,

```
initial_D = 0.006
initial_F = 0.120
initial_values = [initial_Oeig, initial_D, initial_F]
```

Additionally, we define the bounds for the decision variables:

```
min_Oerr = min(df1["J"])
max_Oerr = max(df1["J"])
```

```
bounds = [(min_Oerr, max_Oerr), (0, 0.03), (0, 1)]
```

3.4.4. Objective Function

Then we define the objective function:

```
def one_mass_oscillator(params, Oerr) -> np.ndarray:
    # returns amplitudes of the system
    # Defines the model of a one mass oscilator
    Oeig, D, F = params
    nue = Oerr / Oeig
    V = F / (np.sqrt((1 - nue**2) ** 2 + (4 * D**2 * nue**2)))
    return V
```

```
def objective_function(params, Oerr, amplitudes) -> np.ndarray:
    # objective function to compare calculated and real amplitudes
    return np.sum((amplitudes - one_mass_oscillator(params, Oerr)) ** 2)
```

We define the options for the optimzer and start the optimization process:

```
options = {
    "maxfun": 100000,
    "ftol": 1e-9,
    "xtol": 1e-9,
    "stepmx": 10,
    "eta": 0.25,
    "gtol": 1e-5}
```

```
J = np.array(df1["J"]) # measured frequency
N = np.array(df1["N"]) # measured amplitude
```

```
result = minimize(
    objective_function,
    initial_values,
    args=(J, N),
    method='Nelder-Mead',
    bounds=bounds,
    options=options)
```

3.4.5. Results

We can observe the results:

```
# map optimized values to variables
resonant_frequency = result.x[0]
D = result.x[1]
F = result.x[2]
# predict the resonant amplitude with the fitted one mass oscillator.
X_pred = np.linspace(min_Oerr, max_Oerr, 1000)
ypred_one_mass_oscillator = one_mass_oscillator(result.x, X_pred)
resonant_amplitude = max(ypred_one_mass_oscillator)
print(f"result: {result}")
```

```
result:        message: Optimization terminated successfully.
       success: True
        status: 0
           fun: 53.54144061205875
             x: [ 8.148e+03  7.435e-04  2.153e-02]
           nit: 93
          nfev: 169
 final_simplex: (array([[ 8.148e+03,  7.435e-04,  2.153e-02],
                       [ 8.148e+03,  7.435e-04,  2.153e-02],
                       [ 8.148e+03,  7.435e-04,  2.153e-02],
                       [ 8.148e+03,  7.435e-04,  2.153e-02]]), array([ 5.354e+01,  5.3
```

Finally, we can plot the optimized fit and the real values:

```
plt.scatter(
    df1["J"],
    df1["N"],
    color="black",
    label="Spektralpunkte filtered",
    zorder=5,
```

```
    s=10,
)
# color the max amplitude point red
plt.scatter(
    initial_Oeig,
    max_N,
    color="red",
    label="Max Amplitude",
    zorder=5,
    s=10,
)

plt.plot(
        X_pred,
        ypred_one_mass_oscillator,
        label="Alpha",
        color="blue",
        linewidth=1,
    )
plt.scatter(
    resonant_frequency,
    resonant_amplitude,
    color="blue",
    label="Max Curve Fit",
    zorder=10,
    s=20,
)
```

3.5. Exercises

Exercise 3.1 (Nelder-Mead)**.**

1. What are the steps of the Nelder-Mead algorithm?
2. What are the advantages and disadvantages of the Nelder-Mead algorithm?

Exercise 3.2 (Powell's Method)**.**

1. What are the steps of Powell's method?
2. What are the advantages and disadvantages of Powell's method?
3. What are similarities between the Nelder-Mead and Powell's methods?

Exercise 3.3 (Gradient Descent)**.**

1. What are the steps of the gradient descent algorithm?
2. What is the learning rate in the gradient descent algorithm?

Exercise 3.4 (Newton Method)**.**

1. What is the difference between the gradient descent and Newton method?
2. Which of the two methods converges faster?

Exercise 3.5 (BFGS)**.**

- In which situations is it possible to use algorithms like BFGS, but not the classical Newton method?
- Would you choose Gradient Descent or BFGS for a large-scale optimization problem?

Exercise 3.6 (Dual Annealing)**.**

1. When should you use Simulated Annealing or Dual Annealing over a local optimization algorithm?
2. Describe the Temperature parameter in Simulated Annealing.

Exercise 3.7 (Differential Evolution)**.**

1. What are the key steps in the Differential Evolution algorithm?
2. Explain the crossover operation in Differential Evolution.

3.6. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

4. Sequential Parameter Optimization: Using `scipy` Optimizers

As a default optimizer, `spotpython` uses `differential_evolution` from the `scipy.optimize` package. Alternatively, any other optimizer from the `scipy.optimize` package can be used. This chapter describes how different optimizers from the `scipy optimize` package can be used on the surrogate. The optimization algorithms are available from https://docs.scipy.org/doc/scipy/reference/optimize.html

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from scipy.optimize import shgo
from scipy.optimize import direct
from scipy.optimize import differential_evolution
from scipy.optimize import dual_annealing
from scipy.optimize import basinhopping
from spotpython.utils.init import fun_control_init, design_control_init, optimizer_control_init, sur
```

4.1. The Objective Function Branin

The `spotpython` package provides several classes of objective functions. We will use an analytical objective function, i.e., a function that can be described by a (closed) formula. Here we will use the Branin function. The 2-dim Branin function is

$$y = a(x_2 - bx_1^2 + cx_1 - r)^2 + s(1-t)\cos(x_1) + s,$$

where values of a, b, c, r, s and t are: $a = 1$, $b = 5.1/(4\pi^2)$, $c = 5/\pi$, $r = 6$, $s = 10$ and $t = 1/(8\pi)$.

It has three global minima: $f(x) = 0.397887$ at $(-\pi, 12.275)$, $(\pi, 2.275)$, and $(9.42478, 2.475)$.

Input Domain: This function is usually evaluated on the square $x_1 \in [-5, 10] \times x_2 \in [0, 15]$.

```
from spotpython.fun.objectivefunctions import Analytical
lower = np.array([-5,-0])
upper = np.array([10,15])
fun = Analytical(seed=123).fun_branin
```

4.2. The Optimizer

Differential Evolution (DE) from the `scikit.optimize` package, see https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential_evolution.html#scipy.optimize.differential_evolution is the default optimizer for the search on the surrogate. Other optimiers that are available in `spotpython`, see https://docs.scipy.org/doc/scipy/reference/optimize.html#global-optimization.

- `dual_annealing`
- `direct`
- `shgo`
- `basinhopping`

These optimizers can be selected as follows:

```
from scipy.optimize import differential_evolution
optimizer = differential_evolution
```

As noted above, we will use `differential_evolution`. The optimizer can use `1000` evaluations. This value will be passed to the `differential_evolution` method, which has the argument `maxiter` (int). It defines the maximum number of generations over which the entire differential evolution population is evolved, see https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential_evolution.html#scipy.optimize.differential_evolution

> **TensorBoard**
>
> Similar to the one-dimensional case, which is discussed in Section 7.7, we can use TensorBoard to monitor the progress of the optimization. We will use a similar code, only the prefix is different:

```
fun_control=fun_control_init(
                    lower = lower,
                    upper = upper,
                    fun_evals = 20,
                    PREFIX = "04_DE_"
                    )
surrogate_control=surrogate_control_init(
                    n_theta=len(lower))
```

```
spot_de = Spot(fun=fun,
                   fun_control=fun_control,
                   surrogate_control=surrogate_control)
spot_de.run()
```

```
spotpython tuning: 3.8004662117718677 [######----] 55.00%
spotpython tuning: 3.8004662117718677 [######----] 60.00%
spotpython tuning: 3.159024883515257 [######----] 65.00%
spotpython tuning: 3.133916697143885 [#######---] 70.00%
spotpython tuning: 2.8926749183116236 [########--] 75.00%
spotpython tuning: 0.4190219407803557 [########--] 80.00%
spotpython tuning: 0.401871440801683 [########--] 85.00%
spotpython tuning: 0.39926034519166187 [#########-] 90.00%
spotpython tuning: 0.39926034519166187 [##########] 95.00%
spotpython tuning: 0.39926034519166187 [##########] 100.00% Done...

Experiment saved to 04_DE__res.pkl


<spotpython.spot.spot.Spot at 0x15652bc80>
```

4.2.1. TensorBoard

If the `prefix` argument in `fun_control_init()`is not `None` (as above, where the `prefix` was set to `04_DE_`) , we can start TensorBoard in the background with the following command:

```
tensorboard --logdir="./runs"
```

We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

The TensorBoard plot illustrates how `spotpython` can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important parameter, the learning rate θ of the Kriging surrogate is plotted against the number of optimization steps.

Figure 4.1.: TensorBoard visualization of the spotpython optimization process and the surrogate model.

4.3. Print the Results

```
spot_de.print_results()
```

```
min y: 0.39926034519166187
x0: 3.1509546500431656
x1: 2.298567899278217
```

```
[['x0', np.float64(3.1509546500431656)], ['x1', np.float64(2.298567899278217)]]
```

4.4. Show the Progress

```
spot_de.plot_progress(log_y=True)
```

```
spot_de.surrogate.plot()
```

4.5. Exercises

4.5.1. dual_annealing

- Describe the optimization algorithm, see scipy.optimize.dual_annealing.
- Use the algorithm as an optimizer on the surrogate.

Tip: Selecting the Optimizer for the Surrogate

We can run spotpython with the `dual_annealing` optimizer as follows:

```
spot_da = Spot(fun=fun,
                    fun_control=fun_control,
                    optimizer=dual_annealing,
                    surrogate_control=surrogate_control)
spot_da.run()
spot_da.print_results()
spot_da.plot_progress(log_y=True)
spot_da.surrogate.plot()
```

```
spotpython tuning: 3.800450053998908 [######----] 55.00%
spotpython tuning: 3.800450053998908 [######----] 60.00%
spotpython tuning: 3.158868878483762 [######----] 65.00%
spotpython tuning: 3.1342300163360486 [#######---] 70.00%
spotpython tuning: 2.893119335801825 [########--] 75.00%
spotpython tuning: 0.41578200014582833 [########--] 80.00%
spotpython tuning: 0.40201943256176875 [########--] 85.00%
spotpython tuning: 0.3991976498681318 [#########-] 90.00%
spotpython tuning: 0.3991976498681318 [##########] 95.00%
spotpython tuning: 0.3991976498681318 [##########] 100.00% Done...

Experiment saved to 04_DE__res.pkl
min y: 0.3991976498681318
x0: 3.1505143339793134
x1: 2.2985181488234647
```

4.5.2. `direct`

- Describe the optimization algorithm
- Use the algorithm as an optimizer on the surrogate

> Tip: Selecting the Optimizer for the Surrogate
>
> We can run spotpython with the `direct` optimizer as follows:

```
spot_di = Spot(fun=fun,
                    fun_control=fun_control,
                    optimizer=direct,
                    surrogate_control=surrogate_control)
spot_di.run()
spot_di.print_results()
spot_di.plot_progress(log_y=True)
spot_di.surrogate.plot()
```

```
spotpython tuning: 3.812970247994418 [######----] 55.00%
spotpython tuning: 3.812970247994418 [######----] 60.00%
spotpython tuning: 3.162514679816068 [######----] 65.00%
spotpython tuning: 3.1189615135325983 [#######---] 70.00%
spotpython tuning: 2.6597698275013038 [########--] 75.00%
spotpython tuning: 0.3984917773445744 [########--] 80.00%
spotpython tuning: 0.3984917773445744 [########--] 85.00%
spotpython tuning: 0.3984917773445744 [#########-] 90.00%
spotpython tuning: 0.3984917773445744 [##########] 95.00%
spotpython tuning: 0.3984917773445744 [##########] 100.00% Done...

Experiment saved to 04_DE__res.pkl
min y: 0.3984917773445744
x0: 3.137860082304525
x1: 2.3010973936899863
```

4.5.3. shgo

- Describe the optimization algorithm
- Use the algorithm as an optimizer on the surrogate

Tip: Selecting the Optimizer for the Surrogate

We can run spotpython with the `direct` optimizer as follows:

```
spot_sh = Spot(fun=fun,
                    fun_control=fun_control,
                    optimizer=shgo,
                    surrogate_control=surrogate_control)
spot_sh.run()
spot_sh.print_results()
spot_sh.plot_progress(log_y=True)
spot_sh.surrogate.plot()
```

```
spotpython tuning: 3.800455654373023 [######----] 55.00%
spotpython tuning: 3.800455654373023 [######----] 60.00%
spotpython tuning: 3.1589998607289376 [######----] 65.00%
```

```
spotpython tuning: 3.134192072951082 [#######---] 70.00%
spotpython tuning: 2.896589509267746 [########--] 75.00%
spotpython tuning: 0.433945961604028 [########--] 80.00%
spotpython tuning: 0.40260825513499476 [########--] 85.00%
spotpython tuning: 0.39928786197917887 [#########-] 90.00%
spotpython tuning: 0.39928786197917887 [##########] 95.00%
spotpython tuning: 0.39928786197917887 [##########] 100.00% Done...

Experiment saved to 04_DE__res.pkl
min y: 0.39928786197917887
x0: 3.1517408652058885
x1: 2.2972000572204814
```

4.5.4. basinhopping

- Describe the optimization algorithm
- Use the algorithm as an optimizer on the surrogate

Tip: Selecting the Optimizer for the Surrogate

We can run spotpython with the `direct` optimizer as follows:

```
spot_bh = Spot(fun=fun,
                    fun_control=fun_control,
                    optimizer=basinhopping,
                    surrogate_control=surrogate_control)
spot_bh.run()
spot_bh.print_results()
spot_bh.plot_progress(log_y=True)
spot_bh.surrogate.plot()
```

```
spotpython tuning: 3.8004541616865684 [######----] 55.00%
spotpython tuning: 3.8004541616865684 [######----] 60.00%
spotpython tuning: 3.1590134468691744 [######----] 65.00%
spotpython tuning: 3.1341388578265406 [#######---] 70.00%
spotpython tuning: 2.891819115618228 [########--] 75.00%
spotpython tuning: 0.4232868966271859 [########--] 80.00%
spotpython tuning: 0.4018975355923775 [########--] 85.00%
spotpython tuning: 0.3992908216118565 [#########-] 90.00%
spotpython tuning: 0.3992908216118565 [##########] 95.00%
spotpython tuning: 0.3992908216118565 [##########] 100.00% Done...

Experiment saved to 04_DE__res.pkl
min y: 0.3992908216118565
x0: 3.1513505160646536
x1: 2.2981650300120533
```

4.5.5. Performance Comparison

Compare the performance and run time of the 5 different optimizers:

- `differential_evolution`
- `dual_annealing`
- `direct`
- `shgo`
- `basinhopping`.

The Branin function has three global minima:

- $f(x) = 0.397887$ at
  - $(-\pi, 12.275)$,
  - $(\pi, 2.275)$, and
  - $(9.42478, 2.475)$.

- Which optima are found by the optimizers?
- Does the `seed` argument in `fun = Analytical(seed=123).fun_branin` change this behavior?

4.6. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this chapter is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

Part II.

Numerical Methods

5. Introduction: Numerical Methods

This part deals with numerical implementations of optimization methods. The goal is to understand the implementation of optimization methods and to solve real-world problems numerically and efficiently. We will focus on the implementation of surrogate models, because they are the most efficient way to solve real-world problems.

Starting point is the well-established response surface methodology. It will be extended to the design and analysis of computer experiments (DACE). The DACE methodology is a modern extension of the response surface methodology. It is based on the use of surrogate models, which are used to replace the real-world problem with a simpler problem. The simpler problem is then solved numerically. The solution of the simpler problem is then used to solve the real-world problem.

! Numerical methods: Goals

- Understand implementation of optimization methods
- Solve real-world problems numerically and efficiently

5.1. Response Surface Methods: What is RSM?

Response Surface Methods (RSM) refer to a collection of statistical and mathematical tools that are valuable for developing, improving, and optimizing processes. The overarching theme of RSM involves studying how input variables that control a product or process can potentially influence a response that measures performance or quality characteristics.

The advantages of RSM include a rich literature, well-established methods often used in manufacturing, the importance of careful experimental design combined with a well-understood model, and the potential to add significant value to scientific inquiry, process refinement, optimization, and more. However, there are also drawbacks to RSM, such as the use of simple and crude surrogates, the hands-on nature of the methods, and the limitation of local methods.

RSM is related to various fields, including Design of Experiments (DoE), quality management, reliability, and productivity. Its applications are widespread in industry and manufacturing, focusing on designing, developing, and formulating new products and

improving existing ones, as well as from laboratory research. RSM is commonly applied in domains such as materials science, manufacturing, applied chemistry, climate science, and many others.

An example of RSM involves studying the relationship between a response variable, such as yield (y) in a chemical process, and two process variables: reaction time (ξ_1) and reaction temperature (ξ_2). The provided code illustrates this scenario, following a variation of the so-called "banana function."

In the context of visualization, RSM offers the choice between 3D plots and contour plots. In a 3D plot, the independent variables ξ_1 and ξ_2 are represented, with y as the dependent variable.

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

def fun_rosen(x1, x2):
    b = 10
    return (x1-1)**2 + b*(x2-x1**2)**2

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x = np.arange(-2.0, 2.0, 0.05)
y = np.arange(-1.0, 3.0, 0.05)
X, Y = np.meshgrid(x, y)
zs = np.array(fun_rosen(np.ravel(X), np.ravel(Y)))
Z = zs.reshape(X.shape)

ax.plot_surface(X, Y, Z)

ax.set_xlabel('X1')
ax.set_ylabel('X2')
ax.set_zlabel('Y')

plt.show()
```

- contour plot example:
  - x_1 and x_2 are the independent variables
  - y is the dependent variable

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

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-1.0, 3.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_rosen(X1, X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y , 50)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title("Rosenbrock's Banana Function")
```

```
Text(0.5, 1.0, "Rosenbrock's Banana Function")
```

- Visual inspection: yield is optimized near $(\xi_1.\xi_2)$

5.1.1. Visualization: Problems in Practice

- True response surface is unknown in practice
- When yield evaluation is not as simple as a toy banana function, but a process requiring care to monitor, reconfigure and run, it's far too expensive to observe over a dense grid
- And, measuring yield may be a noisy/inexact process
- That's where stats (RSM) comes in

5.1.2. RSM: Strategies

- RSMs consist of experimental strategies for
- **exploring** the space of the process (i.e., independent/input) variables (above ξ_1 and $\xi 2$)
- empirical statistical **modeling** targeted toward development of an appropriate approximating relationship between the response (yield) and process variables local to a study region of interest

- **optimization** methods for sequential refinement in search of the levels or values of process variables that produce desirable responses (e.g., that maximize yield or explain variation)
- RSM used for fitting an Empirical Model
- True response surface driven by an unknown physical mechanism
- Observations corrupted by noise
- Helpful: fit an empirical model to output collected under different process configurations
- Consider response Y that depends on controllable input variables $\xi_1, \xi_2, \dots, \xi_m$
- RSM: Equations of the Empirical Model
  - $Y = f(\xi_1, \xi_2, \dots, \xi_m) + \epsilon$
  - $\mathbb{E}\{Y\} = \eta = f(\xi 1_1, \xi_2, \dots, \xi_m)$
  - ϵ is treated as zero mean idiosyncratic noise possibly representing
    - inherent variation, or
    - the effect of other systems or
    - variables not under our purview at this time

5.1.3. RSM: Noise in the Empirical Model

- Typical simplifying assumption: $\epsilon \sim N(0, \sigma^2)$
- We seek estimates for f and σ^2 from noisy observations Y at inputs ξ

5.1.4. RSM: Natural and Coded Variables

- Inputs $\xi_1, \xi_2, \dots, \xi_m$ called **natural variables**:
  - expressed in natural units of measurement, e.g., degrees Celsius, pounds per square inch (psi), etc.
- Transformed to **coded variables** $x_1, x_2, \dots, x_m$:
  - to mitigate hassles and confusion that can arise when working with a multitude of scales of measurement
- Typical **Transformations** offering dimensionless inputs $x_1, x_2, \dots, x_m$
  - in the unit cube, or
  - scaled to have a mean of zero and standard deviation of one, are common choices.
- Empirical model becomes $\eta = f(x_1, x_2, \dots, x_m)$

5.1.5. RSM Low-order Polynomials

- Low-order polynomial make the following simplifying Assumptions
  - Learning about f is lots easier if we make some simplifying approximations
  - Appealing to **Taylor's theorem**, a low-order polynomial in a small, localized region of the input (x) space is one way forward
  - Classical RSM:
    * disciplined application of **local analysis** and
    * **sequential refinement** of locality through conservative extrapolation
  - Inherently a **hands-on process**

5.2. First-Order Models (Main Effects Model)

- **First-order model** (sometimes called main effects model) useful in parts of the input space where it's believed that there's little curvature in f:

$$\eta = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

- For example:

$$\eta = 50 + 8x_1 + 3x_2$$

- In practice, such a surface would be obtained by fitting a model to the outcome of a designed experiment
- First-Order Model in python Evaluated on a Grid
- Evaluate model on a grid in a double-unit square centered at the origin
- Coded units are chosen arbitrarily, although one can imagine deploying this approximating function nearby $x^{(0)} = (0, 0)$

```
def fun_1(x1,x2):
    return 50 + 8*x1 + 3*x2
```

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

delta = 0.025
x1 = np.arange(-1.0, 1.0, delta)
x2 = np.arange(-1.0, 1.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_1(X1,X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('First Order Model: $50 + 8x_1 + 3x_2$')
```

```
Text(0.5, 1.0, 'First Order Model: $50 + 8x_1 + 3x_2$')
```

5.2.1. First-Order Model Properties

- First-order model in 2d traces out a **plane** in $y \times (x_1, x_2)$ space
- Only be appropriate for the most trivial of response surfaces, even when applied in a highly localized part of the input space
- Adding **curvature** is key to most applications:
  - First-order model with **interactions** induces limited degree of curvature via different rates of change of y as x_1 is varied for fixed x_2, and vice versa:

$$\eta = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_{12}$$

- For example $\eta = 50 + 8x_1 + 3x_2 - 4x_1 x_2$

5.2.2. First-order Model with Interactions in python

- Code below facilitates evaluations for pairs (x_1, x_2)
- Responses may be observed over a mesh in the same double-unit square

```
def fun_11(x1,x2):
    return 50 + 8 * x1 + 3 * x2 - 4 * x1 * x2
```

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

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-2.0, 2.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_11(X1,X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y, 20)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('First Order Model with Interactions')
```

```
Text(0.5, 1.0, 'First Order Model with Interactions')
```

5.2.3. Observations: First-Order Model with Interactions

- Mean response η is increasing marginally in both x_1 and x_2, or conditional on a fixed value of the other until x_1 is 0.75
- Rate of increase slows as both coordinates grow simultaneously since the coefficient in front of the interaction term x_1x_2 is negative
- Compared to the first-order model (without interactions): surface is far more useful locally
- Least squares regressions often flag up significant interactions when fit to data collected on a design far from local optima

5.3. Second-Order Models

- Second-order model may be appropriate near local optima where f would have substantial curvature:

$$\eta = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{11} x_1^2 + \beta_{22} x^2 + \beta_{12} x_1 x_2$$

- For example

$$\eta = 50 + 8x_1 + 3x_2 - 7x_1^2 - 3x_2^2 - 4x_1x_2$$

- Implementation of the Second-Order Model as `fun_2()`.

```
def fun_2(x1,x2):
    return 50 + 8 * x1 + 3 * x2 - 7 * x1**2 - 3*x2**2 - 4 * x1 * x2
```

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

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-2.0, 2.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_2(X1,X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y, 20)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('Second Order Model with Interactions. Maximum near about $(0.6,0.2)$')
```

```
Text(0.5, 1.0, 'Second Order Model with Interactions. Maximum near about $(0.6,0.2)$')
```

5.3.1. Second-Order Models: Properties

- Not all second-order models would have a single stationary point (in RSM jargon called "a simple maximum")
- In "yield maximizing" setting we're presuming response surface is **concave** down from a global viewpoint
  - even though local dynamics may be more nuanced
- Exact criteria depend upon the eigenvalues of a certain matrix built from those coefficients
- Box and Draper (2007) provide a diagram categorizing all of the kinds of second-order surfaces in RSM analysis, where finding local maxima is the goal

5.3.2. Example: Stationary Ridge

- Example set of coefficients describing what's called a **stationary ridge** is provided by the code below

```
def fun_ridge(x1, x2):
    return 80 + 4*x1 + 8*x2 - 3*x1**2 - 12*x2**2 - 12*x1*x2
```

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

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-2.0, 2.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_ridge(X1,X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y, 20)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('Example of a stationary ridge')
```

```
Text(0.5, 1.0, 'Example of a stationary ridge')
```

5.3.3. Observations: Second-Order Model (Ridge)

- **Ridge**: a whole line of stationary points corresponding to maxima
- Situation means that the practitioner has some flexibility when it comes to optimizing:

- can choose the precise setting of (x_1, x_2) either arbitrarily or (more commonly) by consulting some tertiary criteria

5.3.4. Example: Rising Ridge

- An example of a rising ridge is implemented by the code below.

```
def fun_ridge_rise(x1, x2):
     return 80 - 4*x1 + 12*x2 - 3*x1**2 - 12*x2**2 - 12*x1*x2
```

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

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-2.0, 2.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_ridge_rise(X1,X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y, 20)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('Rising ridge: $\\eta = 80 + 4x_1 + 8x_2 - 3x_1^2 - 12x_2^2 - 12x_1x_2$')
```

```
Text(0.5, 1.0, 'Rising ridge: $\\eta = 80 + 4x_1 + 8x_2 - 3x_1^2 - 12x_2^2 - 12x_1x_2
```

5.3.5. Summary: Rising Ridge

- The stationary point is remote to the study region
- Ccontinuum of (local) stationary points along any line going through the 2d space, excepting one that lies directly on the ridge
- Although estimated response will increase while moving along the axis of symmetry toward its stationary point, this situation indicates
  - either a poor fit by the approximating second-order function, or
  - that the study region is not yet precisely in the vicinity of a local optima—often both.

5.3.6. Falling Ridge

- Inversion of a rising ridge is a falling ridge
- Similarly indicating one is far from local optima, except that the response decreases as you move toward the stationary point
- Finding a falling ridge system can be a back-to-the-drawing-board affair.

5.3.7. Saddle Point

- Finally, we can get what's called a saddle or minimax system.

```
def fun_saddle(x1, x2):
    return 80 + 4*x1 + 8*x2 - 2*x2**2 - 12*x1*x2
```

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

delta = 0.025
x1 = np.arange(-2.0, 2.0, delta)
x2 = np.arange(-2.0, 2.0, delta)
X1, X2 = np.meshgrid(x1, x2)
Y = fun_saddle(X1,X2)
fig, ax = plt.subplots()
CS = ax.contour(X1, X2, Y, 20)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('Saddle Point: $\\eta = 80 + 4x_1 + 8x_2 - 2x_2^2 - 12x_1x_2$')
```

```
Text(0.5, 1.0, 'Saddle Point: $\\eta = 80 + 4x_1 + 8x_2 - 2x_2^2 - 12x_1x_2$')
```

5.3.8. Interpretation: Saddle Points

- Likely further data collection, and/or outside expertise, is needed before determining a course of action in this situation

5.3.9. Summary: Ridge Analysis

- Finding a simple maximum, or stationary ridge, represents ideals in the spectrum of second-order approximating functions
- But getting there can be a bit of a slog
- Using models fitted from data means uncertainty due to noise, and therefore uncertainty in the type of fitted second-order model
- A ridge analysis attempts to offer a principled approach to navigating uncertainties when one is seeking local maxima
- The two-dimensional setting exemplified above is convenient for visualization, but rare in practice
- Complications compound when studying the effect of more than two process variables

5.4. General RSM Models

- General **first-order model** on m process variables $x_1, x_2, \cdots, x_m$ is

$$\eta = \beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m$$

- General **second-order model** on m process variables

$$\eta = \beta_0 + \sum_{j=1}^{m} + \sum_{j=1}^{m} x_j^2 + \sum_{j=2}^{m} \sum_{k=1}^{j} \beta_{kj} x_k x_j.$$

5.4.1. Ordinary Least Squares

- Inference from data is carried out by **ordinary least squares** (OLS)
- For an excellent review including R examples, see Sheather (2009)
- OLS and maximum likelihood estimators (MLEs) are in the typical Gaussian linear modeling setup basically equivalent

5.5. General Linear Regression

We are considering a model, which can be written in the form

$$Y = X\beta + \epsilon,$$

where Y is an $(n \times 1)$ vector of observations (responses), X is an $(n \times p)$ matrix of known form, β is a $(1 \times p)$ vector of unknown parameters, and ϵ is an $(n \times 1)$ vector of errors. Furthermore, $E(\epsilon) = 0$, $Var(\epsilon) = \sigma^2 I$ and the ϵ_i are uncorrelated.

Using the normal equations

$$(X'X)b = X'Y,$$

the solution is given by

$$b = (X'X)^{-1}X'Y.$$

Example 5.1 (Linear Regression)**.**

```
import numpy as np
n = 8
X = np.linspace(0, 2*np.pi, n, endpoint=False).reshape(-1,1)
print(np.round(X, 2))
y = np.sin(X)
print(np.round(y, 2))
# fit an OLS model to the data, predict the response based on the 1ßß x values
m = 100
x = np.linspace(0, 2*np.pi, m, endpoint=False).reshape(-1,1)
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X, y)
y_pred = model.predict(x)
# visualize the data and the fitted model
import matplotlib.pyplot as plt
plt.scatter(X, y, color='black')
plt.plot(x, y_pred, color='blue', linewidth=1)
# add the ground truth (sine function) in orange
plt.plot(x, np.sin(x), color='orange', linewidth=1)
plt.show()
```

```
[[0.  ]
 [0.79]
 [1.57]
 [2.36]
```

```
 [3.14]
 [3.93]
 [4.71]
 [5.5 ]]
[[ 0.  ]
 [ 0.71]
 [ 1.  ]
 [ 0.71]
 [ 0.  ]
 [-0.71]
 [-1.  ]
 [-0.71]]
```

5.6. Designs

- Important: Organize the data collection phase of a response surface study carefully
- **Design**: choice of x's where we plan to observe y's, for the purpose of approximating f
- Analyses and designs need to be carefully matched
- When using a first-order model, some designs are preferred over others
- When using a second-order model to capture curvature, a different sort of design is appropriate

- Design choices often contain features enabling modeling assumptions to be challenged
  - e.g., to check if initial impressions are supported by the data ultimately collected

5.6.1. Different Designs

- **Screening desings**: determine which variables matter so that subsequent experiments may be smaller and/or more focused
- Then there are designs tailored to the form of model (first- or second-order, say) in the screened variables
- And then there are more designs still

5.7. RSM Experimentation

5.7.1. First Step

- RSM-based experimentation begins with a **first-order model**, possibly with interactions
- Presumption: current process operating **far from optimal** conditions
- Collect data and apply **method of steepest ascent** (gradient) on fitted surfaces to move to the optimum

5.7.2. Second Step

- Eventually, if all goes well after several such carefully iterated refinements, **second-order models** are used on appropriate designs in order to zero-in on ideal operating conditions
- Careful analysis of the fitted surface:
  - Ridge analysis with further refinement using gradients of, and
  - standard errors associated with, the fitted surfaces, and so on

5.7.3. Third Step

- Once the practitioner is satisfied with the full arc of
  - design(s),
  - fit(s), and
  - decision(s):

- A small experiment called **confirmation test** may be performed to check if the predicted optimal settings are realizable in practice

5.8. RSM: Review and General Considerations

- First Glimpse, RSM seems sensible, and pretty straightforward as quantitative statistics-based analysis goes
- But: RSM can get complicated, especially when input dimensions are not very low
- Design considerations are particularly nuanced, since the goal is to obtain reliable estimates of main effects, interaction, and curvature while minimizing sampling effort/expense
- RSM Downside: Inefficiency
  - Despite intuitive appeal, several RSM downsides become apparent upon reflection
  - Problems in practice
  - Stepwise nature of sequential decision making is inefficient:
    - Not obvious how to re-use or update analysis from earlier phases, or couple with data from other sources/related experiments
- RSM Downside: Locality
  - In addition to being local in experiment-time (stepwise approach), it's local in experiment-space
  - Balance between
    - exploration (maybe we're barking up the wrong tree) and
    - exploitation (let's make things a little better) is modest at best
- RSM Downside: Expert Knowledge
  - Interjection of expert knowledge is limited to hunches about relevant variables (i.e., the screening phase), where to initialize search, how to design the experiments
  - Yet at the same time classical RSMs rely heavily on constant examination throughout stages of modeling and design and on the instincts of seasoned practitioners
- RSM Downside: Replicability
  - Parallel analyses, conducted according to the same best intentions, rarely lead to the same designs, model fits and so on
  - Sometimes that means they lead to different conclusions, which can be cause for concern

5.8.1. Historical Considerations about RSM

- In spite of those criticisms, however, there was historically little impetus to revise the status quo
- Classical RSM was comfortable in its skin, consistently led to improvements or compelling evidence that none can reasonably be expected
- But then in the late 20th century came an explosive expansion in computational capability, and with it a means of addressing many of those downsides

5.8.2. Status Quo

- Nowadays, field experiments and statistical models, designs and optimizations are coupled with with mathematical models
- Simple equations are not regarded as sufficient to describe real-world systems anymore
- Physicists figured that out fifty years ago; industrial engineers followed, biologists, social scientists, climate scientists and weather forecasters, etc.
- Systems of equations are required, solved over meshes (e.g., finite elements), or stochastically interacting agents
- Goals for those simulation experiments are as diverse as their underlying dynamics
- Optimization of systems is common, e.g., to identify worst-case scenarios

5.8.3. The Role of Statistics

- Solving systems of equations, or interacting agents, requires computing
- Statistics involved at various stages:
  - choosing the mathematical model
  - solving by stochastic simulation (Monte Carlo)
  - designing the computer experiment
  - smoothing over idiosyncrasies or noise
  - finding optimal conditions, or
  - calibrating mathematical/computer models to data from field experiments

5.8.4. New RSM is needed: DACE

- Classical RSMs are not well-suited to any of those tasks, because
  - they lack the fidelity required to model these data
  - their intended application is too local
  - they're also too hands-on.

- Once computers are involved, a natural inclination is to automate—to remove humans from the loop and set the computer running on the analysis in order to maximize computing throughput, or minimize idle time
- **Design and Analysis of Computer Experiments** as a modern extension of RSM
- Experimentation is changing due to advances in machine learning
- **Gaussian process** (GP) regression is the canonical surrogate model
- Origins in geostatistics (gold mining)
- Wide applicability in contexts where prediction is king
- Machine learners exposed GPs as powerful predictors for all sorts of tasks:
- from regression to classification,
- active learning/sequential design,
- reinforcement learning and optimization,
- latent variable modeling, and so on

5.9. Exercises

1. Generate 3d Plots for the Contour Plots in this notebook.
2. Write a `plot_3d` function, that takes the objective function `fun` as an argument.

- It should provide the following interface: `plot_3d(fun)`.

3. Write a `plot_contour` function, that takes the objective function `fun` as an argument:

- It should provide the following interface: `plot_contour(fun)`.

4. Consider further arguments that might be useful for both function, e.g., ranges, size, etc.

5.10. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

6. Kriging (Gaussian Process Regression)

6.1. DACE and RSM

Mathematical models implemented in computer codes are used to circumvent the need for expensive field data collection. These models are particularly useful when dealing with highly nonlinear response surfaces, high signal-to-noise ratios (which often involve deterministic evaluations), and a global scope. As a result, a new approach is required in comparison to Response Surface Methodology (RSM), which was discussed in Section 5.1.

With the improvement in computing power and simulation fidelity, researchers gain higher confidence and a better understanding of the dynamics in physical, biological, and social systems. However, the expansion of configuration spaces and increasing input dimensions necessitates more extensive designs. High-performance computing (HPC) allows for thousands of runs, whereas previously only tens were possible. This shift towards larger models and training data presents new computational challenges.

Research questions for DACE (Design and Analysis of Computer Experiments) include how to design computer experiments that make efficient use of computation and how to meta-model computer codes to save on simulation effort. The choice of surrogate model for computer codes significantly impacts the optimal experiment design, and the preferred model-design pairs can vary depending on the specific goal.

The combination of computer simulation, design, and modeling with field data from similar real-world experiments introduces a new category of computer model tuning problems. The ultimate goal is to automate these processes to the greatest extent possible, allowing for the deployment of HPC with minimal human intervention.

One of the remaining differences between RSM and DACE lies in how they handle noise. DACE employs replication, a technique that would not be used in a deterministic setting, to separate signal from noise. Traditional RSM is best suited for situations where a substantial proportion of the variability in the data is due to noise, and where the acquisition of data values can be severely limited. Consequently, RSM is better suited for a different class of problems, aligning with its intended purposes.

Two very good texts on computer experiments and surrogate modeling are Santner, Williams, and Notz (2003) and Forrester, Sóbester, and Keane (2008). The former

is the canonical reference in the statistics literature and the latter is perhaps more popular in engineering.

Example 6.1 (Example: DACE and RSM)**.** Imagine you are a chemical engineer tasked with optimizing a chemical process to maximize yield. You can control temperature and pressure, but repeated experiments show variability in yield due to inconsistencies in raw materials.

- Using RSM: You would use RSM to design a series of experiments varying temperature and pressure. You would then fit a response surface (a mathematical model) to the data, helping you understand how changes in temperature and pressure affect yield. Using this model, you can identify optimal conditions for maximizing yield despite the noise.
- Using DACE: If instead you use a computational model to simulate the chemical process and want to account for numerical noise or uncertainty in model parameters, you might use DACE. You would run simulations at different conditions, possibly repeating them to assess variability and build a surrogate model that accurately predicts yields, which can be optimized to find the best conditions.

6.1.1. Noise Handling in RSM and DACE

Noise in RSM: In experimental settings, noise often arises due to variability in experimental conditions, measurement errors, or other uncontrollable factors. This noise can significantly affect the response variable, Y. Replication is a standard procedure for handling noise in RSM. In the context of computer experiments, noise might not be present in the traditional sense since simulations can be deterministic. However, variability can arise from uncertainty in input parameters or model inaccuracies. DACE predominantly utilizes advanced interpolation to construct accurate models of deterministic data, sometimes considering statistical noise modeling if needed.

6.2. Background: Expectation, Mean, Standard Deviation

The distribution of a random vector is characterized by some indexes. These are the expectation, the mean, and the standard deviation. The expectation is a measure of the central tendency of a random variable, while the standard deviation quantifies the spread of the distribution. These indexes are essential for understanding the behavior of random variables and making predictions based on them.

Definition 6.1 (Random Variable)**.** A random variable X is a mapping from the sample space of a random experiment to the real numbers. It assigns a numerical value to each outcome of the experiment. Random variables can be either:

- Discrete: If X takes on a countable number of distinct values.
- Continuous: If X takes on an uncountable number of values.

Mathematically, a random variable is a function $X : \Omega \to \mathbb{R}$, where Ω is the sample space.

Definition 6.2 (Probability Distribution)**.** A probability distribution describes how the values of a random variable are distributed. It is characterized for a discrete random variable X by the probability mass function (PMF) $p_X(x)$ and for a continuous random variable X by the probability density function (PDF) $f_X(x)$.

Definition 6.3 (Probability Mass Function (PMF))**.** $p_X(x) = P(X = x)$ gives the probability that X takes the value x.

Definition 6.4 (Probability Density Function (PDF):)**.** $f_X(x)$ is a function such that for any interval $[a, b]$, the probability that X falls within this interval is given by the integral $\int_a^b f_X(x)\mathrm{d}x$.

The distribution function must satisfy:

$$\sum_{x \in D_X} p_X(x) = 1$$

for discrete random variables, where D_X is the domain of X and

$$\int_{-\infty}^{\infty} f_X(x)\mathrm{d}x = 1$$

for continuous random variables.

With these definitions in place, we can now introduce the definition of the expectation, which is a fundamental measure of the central tendency of a random variable.

Definition 6.5 (Expectation)**.** The expectation or expected value of a random variable X, denoted $E[X]$, is defined as follows:

For a discrete random variable X:

$$E[X] = \sum_{x \in D_X} x p_X(x) \quad \text{if } X \text{ is discrete.}$$

For a continuous random variable X:

$$E[X] = \int_{x \in D_X} x f_X(x)\mathrm{d}x \quad \text{if } X \text{ is continuous.}$$

The mean, μ, of a probability distribution is a measure of its central tendency or location. That is, $E(X)$ is defined as the average of all possible values of X, weighted by their probabilities.

Example 6.2 (Expectation)**.** Let X denote the number produced by rolling a fair die. Then

$$E(X) = 1 \times 1/6 + 2 \times 1/6 + 3 \times 1/6 + 4 \times 1/6 + 5 \times 1/6 + 6 \times 1/6 = 3.5.$$

Definition 6.6 (Sample Mean)**.** The sample mean is an important estimate of the population mean. The sample mean of a sample $\{x_i\}$ $(i = 1, 2, \ldots, n)$ is defined as

$$\overline{x} = \frac{1}{n} \sum_i x_i.$$

While both the expectation of a random variable and the sample mean provide measures of central tendency, they differ in their context, calculation, and interpretation.

- The expectation is a theoretical measure that characterizes the average value of a random variable over an infinite number of repetitions of an experiment. The expectation is calculated using a probability distribution and provides a parameter of the entire population or distribution. It reflects the long-term average or central value of the outcomes generated by the random process.
- The sample mean is a statistic. It provides an estimate of the population mean based on a finite sample of data. It is computed directly from the data sample, and its value can vary between different samples from the same population. It serves as an approximation or estimate of the population mean. It is used in statistical inference to make conclusions about the population mean based on sample data.

If we are trying to predict the value of a random variable X by its mean $\mu = E(X)$, the error will be $X - \mu$. In many situations it is useful to have an idea how large this deviation or error is. Since $E(X - \mu) = E(X) - \mu = 0$, it is necessary to use the absolute value or the square of $(X - \mu)$. The squared error is the first choice, because the derivatives are easier to calculate. These considerations motivate the definition of the variance:

Definition 6.7 (Variance)**.** The variance of a random variable X is the mean squared deviation of X from its expected value $\mu = E(X)$.

$$Var(X) = E[(X - \mu)^2]. \tag{6.1}$$

The variance is a measure of the spread of a distribution. It quantifies how much the values of a random variable differ from the mean. A high variance indicates that the values are spread out over a wide range, while a low variance indicates that the values are clustered closely around the mean.

Definition 6.8 (Standard Deviation)**.** Taking the square root of the variance to get back to the same scale of units as X gives the standard deviation. The standard deviation of X is the square root of the variance of X.

$$sd(X) = \sqrt{Var(X)}. \tag{6.2}$$

6.2.1. Calculation of the Standard Deviation with Python

The function `numpy.std` returns the standard deviation, a measure of the spread of a distribution, of the array elements. The argument `ddof` specifies the Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements. By default `ddof` is zero, i.e., `std` uses the formula

$$\sqrt{\frac{1}{N}\sum_i (x_i - \bar{x})^2} \qquad \text{with} \qquad \bar{x} = \sum_{i=1}^{N} x_i/N.$$

Example 6.3 (Standard Deviation with Python)**.** Consider the array $[1, 2, 3]$: Since $\bar{x} = 2$, the following value is computed:

$$\sqrt{1/3 \times ((1-2)^2 + (2-2)^2 + (3-2)^2)} = \sqrt{2/3}.$$

```
import numpy as np
a = np.array([[1, 2, 3]])
np.std(a)
```

```
np.float64(0.816496580927726)
```

The empirical standard deviation (which uses $N-1$), $\sqrt{1/2 \times ((1-2)^2 + (2-2)^2 + (3-2)^2)} = \sqrt{2/2}$, can be calculated in Python as follows:

```
np.std(a, ddof=1)
```

```
np.float64(1.0)
```

6.2.2. The Argument "axis"

When you compute `np.std` with `axis=0`, it calculates the standard deviation along the vertical axis, meaning it computes the standard deviation for each column of the array. On the other hand, when you compute `np.std` with `axis=1`, it calculates the standard deviation along the horizontal axis, meaning it computes the standard deviation for each row of the array. If the axis parameter is not specified, `np.std` computes the standard deviation of the flattened array, i.e., it calculates the standard deviation of all the elements in the array.

Example 6.4 (Axes along which the standard deviation is computed)**.**

```
A = np.array([[1, 2], [3, 4]])
A
```

```
array([[1, 2],
       [3, 4]])
```

First, we calculate the standard deviation of all elements in the array:

```
np.std(A)
```

```
np.float64(1.118033988749895)
```

Setting `axis=0` calculates the standard deviation along the vertical axis (column-wise):

```
np.std(A, axis=0)
```

```
array([1., 1.])
```

Finally, setting `axis=1` calculates the standard deviation along the horizontal axis (row-wise):

```
np.std(A, axis=1)
```

```
array([0.5, 0.5])
```

6.3. Data Types and Precision in Python

The float16 data type in numpy represents a half-precision floating point number. It uses 16 bits of memory, which gives it a precision of about 3 decimal digits.

The float32 data type in numpy represents a single-precision floating point number. It uses 32 bits of memory, which gives it a precision of about 7 decimal digits. On the other hand, float64 represents a double-precision floating point number. It uses 64 bits of memory, which gives it a precision of about 15 decimal digits.

The reason float16 and float32 show fewer digits is because it has less precision due to using less memory. The bits of memory are used to store the sign, exponent, and fraction parts of the floating point number, and with fewer bits, you can represent fewer digits accurately.

Example 6.5 (16 versus 32 versus 64 bit)**.**

```
import numpy as np

# Define a number
num = 0.123456789123456789

num_float16 = np.float16(num)
num_float32 = np.float32(num)
num_float64 = np.float64(num)

print("float16: ", num_float16)
print("float32: ", num_float32)
print("float64: ", num_float64)
```

```
float16:  0.1235
float32:  0.12345679
float64:  0.12345678912345678
```

6.4. Distributions and Random Numbers in Python

Results from computers are deterministic, so it sounds like a contradiction in terms to generate random numbers on a computer. Standard computers generate pseudo-randomnumbers, i.e., numbers that behave as if they were drawn randomly.

> **Deterministic Random Numbers**
>
> - Idea: Generate deterministically numbers that **look** (behave) as if they were drawn randomly.

6.4.1. The Uniform Distribution

Definition 6.9 (The Uniform Distribution)**.** The probability density function of the uniform distribution is defined as:

$$f_X(x) = \frac{1}{b-a} \qquad \text{for } x \in [a, b].$$

Generate 10 random numbers from a uniform distribution between $a = 0$ and $b = 1$:

```
import numpy as np
# Initialize the random number generator
rng = np.random.default_rng(seed=123456789)
n = 10
x = rng.uniform(low=0.0, high=1.0, size=n)
x
```

```
array([0.02771274, 0.90670006, 0.88139355, 0.62489728, 0.79071481,
       0.82590801, 0.84170584, 0.47172795, 0.95722878, 0.94659153])
```

Generate 10,000 random numbers from a uniform distribution between 0 and 10 and plot a histogram of the numbers:

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

# Initialize the random number generator
rng = np.random.default_rng(seed=123456789)

# Generate random numbers from a uniform distribution
x = rng.uniform(low=0, high=10, size=10000)

# Plot a histogram of the numbers
plt.hist(x, bins=50, density=True, edgecolor='black')
plt.title('Uniform Distribution [0,10]')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.show()
```

6.4.2. The Normal Distribution

A normally distributed random variable is a random variable whose associated probability distribution is the normal (or Gaussian) distribution. The normal distribution is a continuous probability distribution characterized by a symmetric bell-shaped curve.

The distribution is defined by two parameters: the mean μ and the standard deviation σ. The mean indicates the center of the distribution, while the standard deviation measures the spread or dispersion of the distribution.

This distribution is widely used in statistics and the natural and social sciences as a simple model for random variables with unknown distributions.

Definition 6.10 (The Normal Distribution)**.** The probability density function of the normal distribution is defined as:

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right), \tag{6.3}$$

where: μ is the mean; σ is the standard deviation.

To generate ten random numbers from a normal distribution, the following command can be used.

```
import numpy as np
rng = np.random.default_rng()
n = 10
mu, sigma = 2, 0.1
x = rng.normal(mu, sigma, n)
x
```

```
array([1.95057148, 1.87913163, 2.17501492, 1.90884462, 1.89728772,
       2.00965324, 2.04690869, 1.87452777, 2.06762231, 1.99860736])
```

Verify the mean:

```
abs(mu - np.mean(x))
```

```
np.float64(0.019183026752047194)
```

Note: To verify the standard deviation, we use `ddof = 1` (empirical standard deviation):

```
abs(sigma - np.std(x, ddof=1))
```

```
np.float64(0.0025472191676452483)
```

```
plot_normal_distribution(mu=0, sigma=1, num_samples=10000)
```

6.4.3. Visualization of the Standard Deviation

The standard deviation of normal distributed can be visualized in terms of the histogram of X:

- about 68% of the values will lie in the interval within one standard deviation of the mean
- 95% lie within two standard deviation of the mean
- and 99.9% lie within 3 standard deviations of the mean.

6.4.4. Standardization of Random Variables

To compare statistical properties of random variables which use different units, it is a common practice to transform these random variables into standardized variables.

Definition 6.11 (Standard Units)**.** If a random variable X has expectation $E(X) = \mu$ and standard deviation $sd(X) = \sigma > 0$, the random variable

$$X^* = (X - \mu)/\sigma$$

is called X in standard units. It has $E(X^*) = 0$ and $sd(X^*) = 1$.

6.4.5. Realizations of a Normal Distribution

Realizations of a normal distribution refers to the actual values that you get when you draw samples from a normal distribution. Each sample drawn from the distribution is a realization of that distribution.

Example 6.6 (Realizations of a Normal Distribution)**.** If you have a normal distribution with a mean of 0 and a standard deviation of 1, each number you draw from that distribution is a realization. Here is a Python example that generates 10 realizations of a normal distribution with a mean of 0 and a standard deviation of 1:

```
import numpy as np
mu = 0
sigma = 1
realizations = np.random.normal(mu, sigma, 10)
print(realizations)
```

```
[ 0.48951662  0.23879586 -0.44811181 -0.610795   -2.02994507  0.60794659
 -0.35410888  0.15258149  0.50127485 -0.78640277]
```

In this code, `np.random.normal` generates ten realizations of a normal distribution with a mean of 0 and a standard deviation of 1. The realizations array contains the actual values drawn from the distribution.

6.4.6. The Multivariate Normal Distribution

The multivariate normal, multinormal, or Gaussian distribution serves as a generalization of the one-dimensional normal distribution to higher dimensions. We will consider k-dimensional random vectors $X = (X_1, X_2, \dots, X_k)$. When drawing samples from this distribution, it results in a set of values represented as $\{x_1, x_2, \dots, x_k\}$. To fully define this distribution, it is necessary to specify its mean μ and covariance matrix Σ. These parameters are analogous to the mean, which represents the central location, and the variance (squared standard deviation) of the one-dimensional normal distribution introduced in Equation 6.3.

Definition 6.12 (The Multivariate Normal Distribution)**.** The probability density function (PDF) of the multivariate normal distribution is defined as:

$$f_X(x) = \frac{1}{\sqrt{(2\pi)^n \det(\Sigma)}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right),$$

where: μ is the $k \times 1$ mean vector; Σ is the $k \times k$ covariance matrix. The covariance matrix Σ is assumed to be positive definite, so that its determinant is strictly positive.

In the context of the multivariate normal distribution, the mean takes the form of a coordinate within an k-dimensional space. This coordinate represents the location where samples are most likely to be generated, akin to the peak of the bell curve in a one-dimensional or univariate normal distribution.

Definition 6.13 (Covariance of two random variables)**.** For two random variables X and Y, the covariance is defined as the expected value (or mean) of the product of their deviations from their individual expected values:

$$\operatorname{cov}(X, Y) = \mathrm{E}\left[(X - \mathrm{E}[X])(Y - \mathrm{E}[Y])\right]$$

For discrete random variables, covariance can be written as:

$$\operatorname{cov}(X, Y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - E(X))(y_i - E(Y)).$$

The covariance within the multivariate normal distribution denotes the extent to which two variables vary together. The elements of the covariance matrix, such as Σ_{ij}, represent the covariances between the variables x_i and x_j. These covariances describe how the different variables in the distribution are related to each other in terms of their variability.

Example 6.7 (The Bivariate Normal Distribution with Positive Covariances)**.** Figure 6.1 shows draws from a bivariate normal distribution with $\mu = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and $\Sigma = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$.

The covariance matrix of a bivariate normal distribution determines the shape, orientation, and spread of the distribution in the two-dimensional space.

The diagonal elements of the covariance matrix (σ_1^2, σ_2^2) are the variances of the individual variables. They determine the spread of the distribution along each axis. A larger variance corresponds to a greater spread along that axis.

The off-diagonal elements of the covariance matrix (σ_{12}, σ_{21}) are the covariances between the variables. They determine the orientation and shape of the distribution. If the covariance is positive, the distribution is stretched along the line $y = x$, indicating that the variables tend to increase together. If the covariance is negative, the distribution is stretched along the line $y = -x$, indicating that one variable tends to decrease as the other increases. If the covariance is zero, the variables are uncorrelated and the distribution is axis-aligned.

In Figure 6.1, the variances are identical and the variables are correlated (covariance is 4), so the distribution is stretched along the line $y = x$.

Figure 6.1.: Bivariate Normal. Mean zero and covariance $\Sigma = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$

Figure 6.2.: Bivariate Normal. Mean zero and covariance $\Sigma = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$.

Example 6.8 (The Bivariate Normal Distribution with Mean Zero and Zero Covariances)**.** The Bivariate Normal Distribution with Mean Zero and Zero Covariances $\sigma_{12} = \sigma_{21} = 0$.

$\Sigma = \begin{pmatrix} 9 & 0 \\ 0 & 9 \end{pmatrix}$

Figure 6.3.: Bivariate Normal. Mean zero and covariance $\Sigma = \begin{pmatrix} 9 & 0 \\ 0 & 9 \end{pmatrix}$

Example 6.9 (The Bivariate Normal Distribution with Mean Zero and Negative Covariances)**.** The Bivariate Normal Distribution with Mean Zero and Negative Covariances $\sigma_{12} = \sigma_{21} = -4$.

$\Sigma = \begin{pmatrix} 9 & -4 \\ -4 & 9 \end{pmatrix}$

6.5. Covariance

In statistics, understanding the relationship between random variables is crucial for making inferences and predictions. Two common measures of such relationships are covariance and correlation. Covariance is a measure of how much two random variables change together. If the variables tend to show similar behavior (i.e., when one increases,

Figure 6.4.: Bivariate Normal. Mean zero and covariance $\Sigma = \begin{pmatrix} 9 & -4 \\ -4 & 9 \end{pmatrix}$

the other tends to increase), the covariance is positive. Conversely, if they tend to move in opposite directions, the covariance is negative.

Definition 6.14 (Covariance)**.** Covariance is calculated as:

$$\mathrm{Cov}(X, Y) = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y]$$

Here, $E[X]$ and $E[Y]$ are the expected values (means) of X and Y, respectively. Covariance has units that are the product of the units of X and Y.

For a vector of random variables $\mathbf{Y} = \left(Y^{(1)}, \dots, Y^{(n)}\right)^T$, the covariance matrix Σ encapsulates the covariances between each pair of variables:

$$\Sigma = \mathrm{Cov}(\mathbf{Y}, \mathbf{Y}) = \begin{pmatrix} \mathrm{Var}(Y^{(1)}) & \mathrm{Cov}(Y^{(1)}, Y^{(2)}) & \dots \\ \mathrm{Cov}(Y^{(2)}, Y^{(1)}) & \mathrm{Var}(Y^{(2)}) & \dots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

The diagonal elements represent the variances, while the off-diagonal elements are the covariances.

6.6. Correlation

6.6.1. Definitions

Definition 6.15 ((Pearson) Correlation Coefficient)**.** The Pearson correlation coefficient, often denoted by ρ for the population or r for a sample, is calculated by dividing the covariance of two variables by the product of their standard deviations.

$$\rho_{XY} = \frac{\mathrm{Cov}(X, Y)}{\sigma_X \sigma_Y}, \tag{6.4}$$

where $\mathrm{Cov}(X, Y)$ is the covariance between variables X and Y, and σ_X and σ_Y are the standard deviations of X and Y, respectively.

Correlation, specifically the correlation coefficient, is a normalized measure of the linear relationship between two variables. It provides a value ranging from -1 to 1, which is scale-free, making it easier to interpret:

- -1: Perfect negative correlation, indicating that as one variable increases, the other decreases.
- 0: No correlation, indicating no linear relationship between the variables.
- 1: Perfect positive correlation, indicating that both variables increase together.

The correlation matrix Ψ provides a way to quantify the linear relationship between multiple variables, extending the notion of the correlation coefficient beyond just pairs of variables. It is derived from the covariance matrix Σ by normalizing each element with respect to the variances of the relevant variables.

Definition 6.16 (The Correlation Matrix Ψ)**.** Given a set of random variables $X_1, X_2, \dots, X_n$, the covariance matrix Σ is:

$$\Sigma = \begin{pmatrix} \sigma_{11} & \sigma_{12} & \cdots & \sigma_{1n} \\ \sigma_{21} & \sigma_{22} & \cdots & \sigma_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{n1} & \sigma_{n2} & \cdots & \sigma_{nn} \end{pmatrix},$$

where $\sigma_{ij} = \mathrm{cov}(X_i, X_j)$ is the covariance between the ith and j^{th} variables. The correlation matrix Ψ is then defined as:

$$\Psi = \left(\rho_{ij}\right) = \left(\tfrac{\sigma_{ij}}{\sqrt{\sigma_{ii}\sigma_{jj}}}\right),$$

where:

- ρ_{ij} is the correlation coefficient between the ith and j^{th} variables.

- σ_{ii} is the variance of the ith variable, i.e., σ_i^2.
- $\sqrt{\sigma_{ii}}$ is the standard deviation of the ith variable, denoted as σ_i.

Thus, Ψ can also be expressed as:

$$\Psi = \begin{pmatrix} 1 & \frac{\sigma_{12}}{\sigma_1\sigma_2} & \cdots & \frac{\sigma_{1n}}{\sigma_1\sigma_n} \\ \frac{\sigma_{21}}{\sigma_2\sigma_1} & 1 & \cdots & \frac{\sigma_{2n}}{\sigma_2\sigma_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\sigma_{n1}}{\sigma_n\sigma_1} & \frac{\sigma_{n2}}{\sigma_n\sigma_2} & \cdots & 1 \end{pmatrix}$$

The correlation matrix Ψ has the following properties:

- The matrix Ψ is symmetric, meaning $\rho_{ij} = \rho_{ji}$.
- The diagonal elements are all 1, as $\rho_{ii} = \frac{\sigma_{ii}}{\sigma_i\sigma_i} = 1$.
- Each off-diagonal element is constrained between -1 and 1, indicating the strength and direction of the linear relationship between pairs of variables.

6.6.2. Computations

Example 6.10 (Computing a Correlation Matrix)**.** Suppose you have a dataset consisting of three variables: X, Y, and Z. You can compute the correlation matrix as follows:

1. Calculate the covariance matrix Σ, which contains covariances between all pairs of variables.
2. Extract the standard deviations for each variable from the diagonal elements of Σ.
3. Use the standard deviations to compute the correlation matrix Ψ.

Suppose we have two sets of data points:

- $X = [1, 2, 3]$
- $Y = [4, 5, 6]$

We want to compute the correlation matrix Ψ for these variables. First, calculate the mean of each variable.

$$\bar{X} = \frac{1+2+3}{3} = 2$$

$$\bar{Y} = \frac{4+5+6}{3} = 5$$

Second, compute the covariance between X and Y. The covariance is calculated as:

$$\text{Cov}(X, Y) = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \bar{X})(Y_i - \bar{Y})$$

For our data:

$$\text{Cov}(X, Y) = \frac{1}{3-1} \left[(1-2)(4-5) + (2-2)(5-5) + (3-2)(6-5)\right]$$

$$= \frac{1}{2} \left[1 + 0 + 1\right] = 1$$

Third, calculate the variances of X and Y. Variance is calculated as:

$$\text{Var}(X) = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \bar{X})^2$$

$$= \frac{1}{2} \left[(1-2)^2 + (2-2)^2 + (3-2)^2\right] = \frac{1}{2}(1 + 0 + 1) = 1$$

Similarly,

$$\text{Var}(Y) = \frac{1}{2} \left[(4-5)^2 + (5-5)^2 + (6-5)^2\right] = \frac{1}{2}(1 + 0 + 1) = 1$$

Then, compute the correlation coefficient. The correlation coefficient ρ_{XY} is:

$$\rho_{XY} = \frac{\text{Cov}(X, Y)}{\sqrt{\text{Var}(X)} \cdot \sqrt{\text{Var}(Y)}}$$

$$= \frac{1}{\sqrt{1} \cdot \sqrt{1}} = 1$$

Finally, construct the correlation matrix. The correlation matrix Ψ is given as:

$$\Psi = \begin{pmatrix} 1 & \rho_{XY} \\ \rho_{XY} & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

Thus, for these two variables, the correlation matrix indicates a perfect positive linear relationship (correlation coefficient of 1) between X and Y.

6.6.3. The Outer-product and the `np.outer` Function

The function `np.outer` from the NumPy library computes the outer product of two vectors. The outer product of two vectors results in a matrix, where each element is the product of an element from the first vector and an element from the second vector.

Definition 6.17 (Outer Product)**.** For two vectors $\mathbf{a}$ and $\mathbf{b}$, the outer product is defined in terms of their elements as:

$$\text{outer}(\mathbf{a}, \mathbf{b}) = \begin{pmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 & \cdots & a_1 \cdot b_n \\ a_2 \cdot b_1 & a_2 \cdot b_2 & \cdots & a_2 \cdot b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_m \cdot b_1 & a_m \cdot b_2 & \cdots & a_m \cdot b_n \end{pmatrix},$$

where $\mathbf{a}$ is a vector of length m and $\mathbf{b}$ is a vector of length n.

Example 6.11 (Computing the Outer Product)**.** We will consider two vectors, $\mathbf{a}$ and $\mathbf{b}$:

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5])
outer_product = np.outer(a, b)
print("Vector a:", a)
print("Vector b:", b)
print("Outer Product:\n", outer_product)
```

```
Vector a: [1 2 3]
Vector b: [4 5]
Outer Product:
 [[ 4  5]
 [ 8 10]
 [12 15]]
```

For the vectors defined:

$$\mathbf{a} = [1, 2, 3], \quad \mathbf{b} = [4, 5]$$

The outer product will be:

$$\begin{pmatrix} 1\cdot 4 & 1\cdot 5 \\ 2\cdot 4 & 2\cdot 5 \\ 3\cdot 4 & 3\cdot 5 \end{pmatrix} = \begin{pmatrix} 4 & 5 \\ 8 & 10 \\ 12 & 15 \end{pmatrix}.$$

Thus, `np.outer` creates a matrix with dimensions $m \times n$, where m is the length of the first vector and n is the length of the second vector. The function is particularly useful in various mathematical and scientific computations where matrix representations of vector relationships are needed.

Example 6.12 (Computing the Covariance and the Correlation Matrix)**.** The following Python code computes the covariance and correlation matrices using the NumPy library.

```
import numpy as np

def calculate_cov_corr_matrices(data, rowvar=False)->(np.array, np.array):
    """
    Calculate the covariance and correlation matrices of the input data.

    Args:
        data (np.array):
            Input data array.
        rowvar (bool):
            Whether the data is row-wise or column-wise.
            Default is False (column-wise).

    Returns:
        np.array: Covariance matrix.
        np.array: Correlation matrix.

    Examples:
        >>> data = np.array([[1, 2, 3],
                             [4, 5, 6],
                             [7, 8, 9]])
        >>> calculate_cov_corr_matrices(data)
    """
    cov_matrix = np.cov(data, rowvar=rowvar)
    std_devs = np.sqrt(np.diag(cov_matrix))
    # check whether the standard deviations are zero
    # and throw an error if they are
    if np.any(std_devs == 0):
        raise ValueError("Correlation matrix cannot be computed,"+
                         " because one or more variables have zero variance.")
```

```
    corr_matrix = cov_matrix / np.outer(std_devs, std_devs)
    return cov_matrix, corr_matrix
```

```
A = np.array([[0,1],
                  [1,0]])
print(f"Input matrix:\n {A}")
Sigma, Psi = calculate_cov_corr_matrices(A)
print(f"Covariance matrix:\n {Sigma}")
print(f"Correlation matrix:\n {Psi}")
```

```
Input matrix:
 [[0 1]
 [1 0]]
Covariance matrix:
 [[ 0.5 -0.5]
 [-0.5  0.5]]
Correlation matrix:
 [[ 1. -1.]
 [-1.  1.]]
```

6.6.4. Correlation and Independence

Definition 6.18 (Statistical Independence (Independence of Random Vectors))**.** Two random vectors are statistically independent if the joint distribution of the vectors is equal to the product of their marginal distributions.

This means that knowing the realization of one vector gives you no information about the realization of the other vector. This independence is a probabilistic concept used in statistics and probability theory to denote that two sets of random variables do not affect each other. Independence implies that all pairwise covariances between the components of the two vectors are zero, but zero covariance does not imply independence unless certain conditions are met (e.g., normality). Statistical independence is a stronger condition than zero covariance. Statistical independence is not related to the linear independence of vectors in linear algebra.

Example 6.13 (Covariance of Independent Variables)**.** Consider a covariance matrix where variables are independent:

```
A = np.array([[1,-1],
[2,0],
[3,1],
[4,0],
```

```
[5,-1]])
print(f"Input matrix:\n {A}")
Sigma, Psi = calculate_cov_corr_matrices(A)
print(f"Covariance matrix:\n {Sigma}")
print(f"Correlation matrix:\n {Psi}")
```

```
Input matrix:
 [[ 1 -1]
 [ 2  0]
 [ 3  1]
 [ 4  0]
 [ 5 -1]]
Covariance matrix:
 [[2.5 0. ]
 [0.  0.7]]
Correlation matrix:
 [[1. 0.]
 [0. 1.]]
```

Here, since the off-diagonal elements are 0, the variables are uncorrelated. X increases linearly, while Y alternates in a simple pattern with no trend that is linearly related to Y.

Example 6.14 (Strong Correlation)**.** For a covariance matrix with strong positive correlation:

```
A = np.array([[10,-1],
[20,0],
[30,1],
[40,2],
[50,3]])
print(f"Input matrix:\n {A}")
Sigma, Psi = calculate_cov_corr_matrices(A)
print(f"Covariance matrix:\n {Sigma}")
print(f"Correlation matrix:\n {Psi}")
```

```
Input matrix:
 [[10 -1]
 [20  0]
 [30  1]
 [40  2]
 [50  3]]
Covariance matrix:
```

```
 [[250.   25. ]
 [ 25.    2.5]]
Correlation matrix:
 [[1. 1.]
 [1. 1.]]
```

A value close to 1 suggests a strong positive relationship between the variables.

Example 6.15 (Strong Negative Correlation)**.**

```
A = np.array([[10,1],
[20,0],
[30,-1],
[40,-2],
[50,-3]])
print(f"Input matrix:\n {A}")
Sigma, Psi = calculate_cov_corr_matrices(A)
print(f"Covariance matrix:\n {Sigma}")
print(f"Correlation matrix:\n {Psi}")
```

```
Input matrix:
 [[10  1]
 [20  0]
 [30 -1]
 [40 -2]
 [50 -3]]
Covariance matrix:
 [[250.  -25. ]
 [-25.    2.5]]
Correlation matrix:
 [[ 1. -1.]
 [-1.  1.]]
```

This matrix indicates a perfect negative correlation where one variable increases as the other decreases.

6.7. R-Squared in Simple Linear Regression

In simple linear regression, the relationship between the independent variable X and the dependent variable Y is modeled using the equation:

$$Y = \beta_0 + \beta_1 X + \epsilon$$

Here, β_0 is the intercept, β_1 is the slope or regression coefficient, and ϵ is the error term.

Definition 6.19 (R-Squared (R^2))**.** R^2 is a measure of how well the regression model explains the variance in the dependent variable. It is calculated as the square of the correlation coefficient (r) between the actual values Y and the predicted values $\widehat{Y}$ from the regression model. It ranges from 0 to 1, where:

- 1 indicates that the regression predictions perfectly fit the data.
- 0 indicates that the model does not explain any of the variability in the target data around its mean.

In simple linear regression, where there is one independent variable X and one dependent variable Y, the R-squared (R^2) is the square of the Pearson correlation coefficient (r) between the observed values of the dependent variable and the values predicted by the regression model. That is, in simple linear regression, we have

$$R^2 = r^2.$$

This equivalence holds specifically for simple linear regression due to the direct relationship between the linear fit and the correlation of two variables. In multiple linear regression, while R^2 still represents the proportion of variance explained by the model, it is not simply the square of a single correlation coefficient as it involves multiple predictors.

6.8. Cholesky Decomposition and Positive Definite Matrices

We consider the definiteness of a matrix, before discussing the Cholesky decomposition.

Definition 6.20 (Positive Definite Matrix)**.** A symmetric matrix A is positive definite if all its eigenvalues are positive.

Example 6.16 (Positive Definite Matrix)**.** Given a symmetric matrix $A = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$, the eigenvalues of A are $\lambda_1 = 13$ and $\lambda_2 = 5$. Since both eigenvalues are positive, the matrix A is positive definite.

Definition 6.21 (Negative Definite, Positive Semidefinite, and Negative Semidefinite Matrices)**.** Similarily, a symmetric matrix A is negative definite if all its eigenvalues are negative. It is positive semidefinite if all its eigenvalues are non-negative, and negative semidefinite if all its eigenvalues are non-positive.

The covariance matrix must be positive definite for a multivariate normal distribution for a couple of reasons:

- Semidefinite vs Definite: A covariance matrix is always symmetric and positive semidefinite. However, for a multivariate normal distribution, it must be positive definite, not just semidefinite. This is because a positive semidefinite matrix can have zero eigenvalues, which would imply that some dimensions in the distribution have zero variance, collapsing the distribution in those dimensions. A positive definite matrix has all positive eigenvalues, ensuring that the distribution has positive variance in all dimensions.
- Invertibility: The multivariate normal distribution's probability density function involves the inverse of the covariance matrix. If the covariance matrix is not positive definite, it may not be invertible, and the density function would be undefined.

In summary, the covariance matrix being positive definite ensures that the multivariate normal distribution is well-defined and has positive variance in all dimensions.

The definiteness of a matrix can be checked by examining the eigenvalues of the matrix. If all eigenvalues are positive, the matrix is positive definite.

```
import numpy as np

def is_positive_definite(matrix):
    return np.all(np.linalg.eigvals(matrix) > 0)

matrix = np.array([[9, 4], [4, 9]])
print(is_positive_definite(matrix))  # Outputs: True
```

```
True
```

However, a more efficient way to check the definiteness of a matrix is through the Cholesky decomposition.

Definition 6.22 (Cholesky Decomposition)**.** For a given symmetric positive-definite matrix $A \in \mathbb{R}^{n \times n}$, there exists a unique lower triangular matrix $L \in \mathbb{R}^{n \times n}$ with positive diagonal elements such that:

$$A = LL^T.$$

Here, L^T denotes the transpose of L.

Example 6.17 (Cholesky decomposition using `numpy`)**.** `linalg.cholesky` computes the Cholesky decomposition of a matrix, i.e., it computes a lower triangular matrix L such that $LL^T = A$. If the matrix is not positive definite, an error (`LinAlgError`) is raised.

```
import numpy as np

# Define a Hermitian, positive-definite matrix
A = np.array([[9, 4], [4, 9]])

# Compute the Cholesky decomposition
L = np.linalg.cholesky(A)

print("L = \n", L)
print("L*LT = \n", np.dot(L, L.T))
```

```
L =
 [[3.         0.        ]
 [1.33333333 2.68741925]]
L*LT =
 [[9. 4.]
 [4. 9.]]
```

Example 6.18 (Cholesky Decomposition)**.** Given a symmetric positive-definite matrix $A = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$, the Cholesky decomposition computes the lower triangular matrix L such that $A = LL^T$. The matrix L is computed as:

$$L = \begin{pmatrix} 3 & 0 \\ 4/3 & 2 \end{pmatrix},$$

so that

$$LL^T = \begin{pmatrix} 3 & 0 \\ 4/3 & \sqrt{65}/3 \end{pmatrix} \begin{pmatrix} 3 & 4/3 \\ 0 & \sqrt{65}/3 \end{pmatrix} = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix} = A.$$

An efficient implementation of the definiteness-check based on Cholesky is already available in the `numpy` library. It provides the `np.linalg.cholesky` function to compute the Cholesky decomposition of a matrix. This more efficient `numpy`-approach can be used as follows:

```
import numpy as np

def is_pd(K):
    try:
```

```
        np.linalg.cholesky(K)
        return True
    except np.linalg.linalg.LinAlgError as err:
        if 'Matrix is not positive definite' in err.message:
            return False
        else:
            raise
matrix = np.array([[9, 4], [4, 9]])
print(is_pd(matrix))  # Outputs: True
```

```
True
```

6.9. Maximum Likelihood Estimation: Multivariate Normal Distribution

6.9.1. The Joint Probability Density Function of the Multivariate Normal Distribution

Consider the first n terms of an identically and independently distributed (i.i..d.) sequence $X^{(j)}$ of k-dimensional multivariate normal random vectors, i.e.,

$$X^{(j)} \sim N(\mu, \Sigma), j = 1, 2, \dots . \tag{6.5}$$

The joint probability density function of the j-th term of the sequence is

$$f_X(x_j) = \frac{1}{\sqrt{(2\pi)^k \det(\Sigma)}} \exp\left(-\frac{1}{2}(x_j - \mu)^T \Sigma^{-1} (x_j - \mu)\right),$$

where: μ is the $k \times 1$ mean vector; Σ is the $k \times k$ covariance matrix. The covariance matrix Σ is assumed to be positive definite, so that its determinant is strictly positive. We use $x_1, \dots x_n$, i.e., the realizations of the first n random vectors in the sequence, to estimate the two unknown parameters μ and Σ.

6.9.2. The Log-Likelihood Function

Definition 6.23 (Likelihood Function)**.** The likelihood function is defined as the joint probability density function of the observed data, viewed as a function of the unknown parameters.

Since the terms in the sequence Equation 6.5 are independent, their joint density is equal to the product of their marginal densities. As a consequence, the likelihood function can be written as the product of the individual densities:

$$\begin{aligned} L(\mu, \Sigma) &= \prod_{j=1}^{n} f_X(x_j) = \prod_{j=1}^{n} \frac{1}{\sqrt{(2\pi)^k \det(\Sigma)}} \exp\left(-\frac{1}{2}(x_j - \mu)^T \Sigma^{-1}(x_j - \mu)\right) \\ &= \frac{1}{(2\pi)^{nk/2} \det(\Sigma)^{n/2}} \exp\left(-\frac{1}{2}\sum_{j=1}^{n}(x_j - \mu)^T \Sigma^{-1}(x_j - \mu)\right). \end{aligned} \tag{6.6}$$

Taking the natural logarithm of the likelihood function, we obtain the log-likelihood function:

Example 6.19 (Log-Likelihood Function of the Multivariate Normal Distribution)**.** The log-likelihood function of the multivariate normal distribution is given by

$$\ell(\mu, \Sigma) = -\frac{nk}{2}\ln(2\pi) - \frac{n}{2}\ln(\det(\Sigma)) - \frac{1}{2}\sum_{j=1}^{n}(x_j - \mu)^T \Sigma^{-1}(x_j - \mu).$$

The likelihood function is well-defined only if $\det(\Sigma) > 0$.

6.10. Constructing a Surrogate

> **Note**
>
> This section is based on chapter 2 in Forrester, Sóbester, and Keane (2008).

Definition 6.24 (Black Box Problem)**.** We are trying to learn a mapping that converts the vector $\mathbf{x}$ into a scalar output y, i.e., we are trying to learn a function

$$y = f(x).$$

If function is hidden ("lives in a black box"), so that the physics of the problem is not known, the problem is called a black box problem.

This black box could take the form of either a physical or computer experiment, for example, a finite element code, which calculates the maximum stress (σ) for given product dimensions ($\mathbf{x}$).

Definition 6.25 (Generic Solution)**.** The generic solution method is to collect the output values $y^{(1)}$, $y^{(2)}$, ..., $y^{(n)}$ that result from a set of inputs $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, ..., $\mathbf{x}^{(n)}$ and find a best guess $\hat{f}(\mathbf{x})$ for the black box mapping f, based on these known observations.

6.10.1. Stage One: Preparing the Data and Choosing a Modelling Approach

The first step is the identification, through a small number of observations, of the inputs that have a significant impact on f; that is the determination of the shortest design variable vector $\mathbf{x} = \{x_1, x_2, \dots, x_k\}^T$ that, by sweeping the ranges of all of its variables, can still elicit most of the behavior the black box is capable of. The ranges of the various design variables also have to be established at this stage.

The second step is to recruit n of these k-vectors into a list

$$\mathbf{X} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(n)}\}^T,$$

where each $\mathbf{x}^{(i)}$ is a k-vector. The corresponding responses are collected in a vector such that this represents the design space as thoroughly as possible.

In the surrogate modeling process, the number of samples n is often limited, as it is constrained by the computational cost (money and/or time) associated with obtaining each observation.

It is advisable to scale $\mathbf{x}$ at this stage into the unit cube $[0, 1]^k$, a step that can simplify the subsequent mathematics and prevent multidimensional scaling issues.

We now focus on the attempt to learn f through data pairs

$$\{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \dots, (\mathbf{x}^{(n)}, y^{(n)})\}.$$

This supervised learning process essentially involves searching across the space of possible functions $\hat{f}$ that would replicate observations of f. This space of functions is infinite. Any number of hypersurfaces could be drawn to pass through or near the known observations, accounting for experimental error. However, most of these would generalize poorly; they would be practically useless at predicting responses at new sites, which is the ultimate goal.

Example 6.20 (The Needle(s) in the Haystack Function)**.** An extreme example is the 'needle(s) in the haystack' function:

$$f(x) = \begin{cases} y^{(1)}, & \text{if } x = \mathbf{x}^{(1)} \\ y^{(2)}, & \text{if } x = \mathbf{x}^{(2)} \\ \vdots & \\ y^{(n)}, & \text{if } x = \mathbf{x}^{(n)} \\ 0, & \text{otherwise.} \end{cases}$$

While this predictor reproduces all training data, it seems counter-intuitive and unsettling to predict 0 everywhere else for most engineering functions. Although there is a small chance that the function genuinely resembles the equation above and we sampled exactly where the needles are, it is highly unlikely.

There are countless other configurations, perhaps less contrived, that still generalize poorly. This suggests a need for systematic means to filter out nonsensical predictors. In our approach, we embed the structure of f into the model selection algorithm and search over its parameters to fine-tune the approximation to observations. For instance, consider one of the simplest models,

$$f(x, \mathbf{w}) = \mathbf{w}^T \mathbf{x} + v. \tag{6.7}$$

Learning f with this model implies that its structure—a hyperplane—is predetermined, and the fitting process involves finding the $k + 1$ parameters (the slope vector $\mathbf{w}$ and the intercept v) that best fit the data. This will be accomplished in Stage Two.

Complicating this further is the noise present in observed responses (we assume design vectors $\mathbf{x}$ are not corrupted). Here, we focus on learning from such data, which sometimes risks overfitting.

Definition 6.26 (Overfitting)**.** Overfitting occurs when the model becomes too flexible and captures not only the underlying trend but also the noise in the data.

In the surrogate modeling process, the second stage as described in Section 6.10.2, addresses this issue of complexity control by estimating the parameters of the fixed structure model. However, foresight is necessary even at the model type selection stage.

Model selection often involves physics-based considerations, where the modeling technique is chosen based on expected underlying responses.

Example 6.21 (Model Selection)**.** Modeling stress in an elastically deformed solid due to small strains may justify using a simple linear approximation. Without insights into the physics, and if one fails to account for the simplicity of the data, a more complex and excessively flexible model may be incorrectly chosen. Although parameter estimation might still adjust the approximation to become linear, an opportunity to develop a simpler and robust model may be lost.

- Simple linear (or polynomial) models, despite their lack of flexibility, have advantages like applicability in further symbolic computations.
- Conversely, if we incorrectly assume a quadratic process when multiple peaks and troughs exist, the parameter estimation stage will not compensate for an unsuitable model choice. A quadratic model is too rigid to fit a multimodal function, regardless of parameter adjustments.

6.10.2. Stage Two: Parameter Estimation and Training

Assuming that Stage One helped identify the k critical design variables, acquire the learning data set, and select a generic model structure $f(\mathbf{x}, \mathbf{w})$, the task now is to

estimate parameters $\mathbf{w}$ to ensure the model fits the data optimally. Among several estimation criteria, we will discuss two methods here.

Definition 6.27 (Maximum Likelihood Estimation)**.** Given a set of parameters $\mathbf{w}$, the model $f(\mathbf{x}, \mathbf{w})$ allows computation of the probability of the data set

$$\{(\mathbf{x}^{(1)}, y^{(1)} \pm \epsilon), (\mathbf{x}^{(2)}, y^{(2)} \pm \epsilon), \dots, (\mathbf{x}^{(n)}, y^{(n)} \pm \epsilon)\}$$

resulting from f (where ϵ is a small error margin around each data point).

Taking Equation 6.6 and assuming errors ϵ are independently and normally distributed with standard deviation σ, the probability of the data set is given by:

$$P = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left[-\frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(y^{(i)} - f(\mathbf{x}^{(i)}, \mathbf{w})\right)^2 \epsilon\right].$$

Intuitively, this is equivalent to the likelihood of the parameters given the data. Accepting this intuitive relationship as a mathematical one aids in model parameter estimation. This is achieved by maximizing the likelihood or, more conveniently, minimizing the negative of its natural logarithm:

$$\min_{\mathbf{w}} \sum_{i=1}^{n} \frac{[y^{(i)} - f(\mathbf{x}^{(i)}, \mathbf{w})]^2}{2\sigma^2} + \frac{n}{2}\ln\epsilon. \tag{6.8}$$

If we assume σ and ϵ are constants, Equation 6.8 simplifies to the well-known least squares criterion:

$$\min_{\mathbf{w}} \sum_{i=1}^{n} [y^{(i)} - f(\mathbf{x}^{(i)}, \mathbf{w})]^2.$$

Cross-validation is another method used to estimate model performance.

Definition 6.28 (Cross-Validation)**.** Cross-validation splits the data randomly into q roughly equal subsets, and then cyclically removing each subset and fitting the model to the remaining $q-1$ subsets. A loss function L is then computed to measure the error between the predictor and the withheld subset for each iteration, with contributions summed over all q iterations. More formally, if a mapping $\theta : \{1, \dots, n\} \to \{1, \dots, q\}$ describes the allocation of the n training points to one of the q subsets and $f^{(-\theta(i))}(\mathbf{x})$ is the predicted value by removing the subset $\theta(i)$ (i.e., the subset where observation i belongs), the cross-validation measure, used as an estimate of prediction error, is:

$$CV = \frac{1}{n}\sum_{i=1}^{n} L(y^{(i)}, f^{(-\theta(i))}(\mathbf{x}^{(i)})). \tag{6.9}$$

Introducing the squared error as the loss function and considering our generic model f still dependent on undetermined parameters, we write Equation 6.9 as:

$$CV = \frac{1}{n}\sum_{i=1}^{n}[y^{(i)} - f^{(-\theta(i))}(\mathbf{x}^{(i)})]^2. \tag{6.10}$$

The extent to which Equation 6.10 is an unbiased estimator of true risk depends on q. It is shown that if $q = n$, the leave-one-out cross-validation (LOOCV) measure is almost unbiased. However, LOOCV can have high variance because subsets are very similar. Hastie, Tibshirani, and Friedman (2017)) suggest using compromise values like $q = 5$ or $q = 10$. Using fewer subsets also reduces the computational cost of the cross-validation process, see also Arlot, Celisse, et al. (2010) and Kohavi (1995).

6.10.3. Stage Three: Model Testing

If there is a sufficient amount of observational data, a random subset should be set aside initially for model testing. Hastie, Tibshirani, and Friedman (2017) recommend setting aside approximately $0.25n$ of $\mathbf{x} \to y$ pairs for testing purposes. These observations must remain untouched during Stages One and Two, as their sole purpose is to evaluate the testing error—the difference between true and approximated function values at the test sites—once the model has been built. Interestingly, if the main goal is to construct an initial surrogate for seeding a global refinement criterion-based strategy (as discussed in Section 3.2 in Forrester, Sóbester, and Keane (2008)), the model testing phase might be skipped.

It is noted that, ideally, parameter estimation (Stage Two) should also rely on a separate subset. However, observational data is rarely abundant enough to afford this luxury (if the function is cheap to evaluate and evaluation sites are selectable, a surrogate model might not be necessary).

When data are available for model testing and the primary objective is a globally accurate model, using either a root mean square error (RMSE) metric or the correlation coefficient (r^2) is recommended. To test the model, a test data set of size n_t is used alongside predictions at the corresponding locations to calculate these metrics.

The RMSE is defined as follows:

Definition 6.29 (Root Mean Square Error (RMSE))**.**

$$\text{RMSE} = \sqrt{\frac{1}{n_t}\sum_{i=1}^{n_t}(y^{(i)} - \hat{y}^{(i)})^2},$$

Ideally, the RMSE should be minimized, acknowledging its limitation by errors in the objective function f calculation. If the error level is known, like a standard deviation, the aim might be to achieve an RMSE within this value. Often, the target is an RMSE within a specific percentage of the observed data's objective value range.

The squared correlation coefficient r, see Equation 6.4, between the observed y and predicted $\hat{y}$ values can be computed as:

$$r^2 = \left(\frac{\mathrm{cov}(y, \hat{y})}{\sqrt{\mathrm{var}(y)\mathrm{var}(\hat{y})}} \right)^2, \tag{6.11}$$

Equation 6.11 and can be expanded as:

$$r^2 = \left(\frac{n_t \sum_{i=1}^{n_t} y^{(i)} \hat{y}^{(i)} - \sum_{i=1}^{n_t} y^{(i)} \sum_{i=1}^{n_t} \hat{y}^{(i)}}{\sqrt{\left(n_t \sum_{i=1}^{n_t} (y^{(i)})^2 - \left(\sum_{i=1}^{n_t} y^{(i)} \right)^2 \right) \left(n_t \sum_{i=1}^{n_t} (\hat{y}^{(i)})^2 - \left(\sum_{i=1}^{n_t} \hat{y}^{(i)} \right)^2 \right)}} \right)^2.$$

The correlation coefficient r^2 does not require scaling the data sets and only compares landscape shapes, not values. An $r^2 > 0.8$ typically indicates a surrogate with good predictive capability.

The methods outlined provide quantitative assessments of model accuracy, yet visual evaluations can also be insightful. In general, the RMSE will not reach zero but will stabilize around a low value. At this point, the surrogate model is saturated with data, and further additions do not enhance the model globally (though local improvements can occur at newly added points if using an interpolating model).

Example 6.22 (The Tea and Sugar Analogy)**.** Forrester, Sóbester, and Keane (2008) illustrates this saturation point using a comparision with a cup of tea and sugar. The tea represents the surrogate model, and sugar represents data. Initially, the tea is unsweetened, and adding sugar increases its sweetness. Eventually, a saturation point is reached where no more sugar dissolves, and the tea cannot get any sweeter. Similarly, a more flexible model, like one with additional parameters or employing interpolation rather than regression, can increase the saturation point—akin to making a hotter cup of tea for dissolving more sugar.

6.11. Sampling Plans

Definition 6.30 (Sampling Plan)**.** In the context of computer experiments, the term "sampling plan" refers to the set of input values at which the computer code is evaluated.

The goal of a sampling plan is to efficiently explore the input space to understand the behavior of the computer code and build a surrogate model that accurately represents the code's behavior. Traditionally, Response Surface Methodology (RSM) has been used to design sampling plans for computer experiments. These sampling plans are based on procedures that generate points by means of a rectangiular grid or a factorial design. However, more recently, Design and Analysis of Computer Experiments (DACE) has emerged as a more flexible and powerful approach for designing sampling plans. `spotpython` uses a class for generating space-filling designs using Latin Hypercube Sampling (LHS) and maximin distance criteria. It is based on `scipy`'s `LatinHypercube` class. The following example demonstrates how to generate a Latin Hypercube Sampling design using `spotpython`. The result is shown in Figure 6.5. As can seen in the figure, a Latin hypercube sample generates n points in $[0, 1)^d$. Each univariate marginal distribution is stratified, placing exactly one point in $[j/n, (j+1)/n)$ for $j = 0, 1, ..., n-1$.

```
import matplotlib.pyplot as plt
import numpy as np
from spotpython.design.spacefilling import SpaceFilling
lhd = SpaceFilling(k=2, seed=123)
X = lhd.scipy_lhd(n=10, repeats=1, lower=np.array([0, 0]), upper=np.array([10, 10]))
plt.scatter(X[:, 0], X[:, 1])
plt.xlabel('x1')
plt.ylabel('x2')
plt.grid()
```

Figure 6.5.: Latin Hypercube Sampling design (sampling plan)

6.12. Kriging

6.12.1. The Kriging Idea in a Nutshell

Kriging can be applied to planned experiments, where the design is based on a sampling plan as shown in Figure 6.5, as well as to computer experiments, where the design is based on the computer code's input space, as shown in Figure 6.6.

Kriging can be explained using the concept of radial basis functions.

Radial basis functions (RBFs) are a class of functions used in various types of interpolation and approximation tasks. An RBF is a real-valued function whose value depends only on the distance from a certain point, called the center, usually in a multidimensional space. This distance is typically measured using the Euclidean distance.

6.12.2. Radial Basis Function (RBF)

Mathematically, a radial basis function ϕ can be expressed as:

Figure 6.6.: Eight measurements of an unknown function. No sampling plan was used.

$$\phi(\mathbf{x}) = \phi(\|\mathbf{x} - \mathbf{c}\|),$$

where $\mathbf{x}$ is the input vector, $\mathbf{c}$ is the center of the function, and $\|\mathbf{x} - \mathbf{c}\|$ denotes the Euclidean distance between $\mathbf{x}$ and $\mathbf{c}$.

Common types of radial basis functions include:

- Linear: $\phi(r) = r$.
- Cubic: $\phi(r) = r^3$,
- Gaussian: $\phi(r) = e^{-(\epsilon r)^2}$,

where r is the distance and ϵ is a parameter that determines the width of the Gaussian function.

Radial basis functions are widely used in interpolation for scattered data points in multiple dimensions, in machine learning models such as radial basis function networks, and in numerical solutions to partial differential equations due to their smooth approximation properties.

In general, we onsider observed data of an unknown function f at n points $x_1, \dots, x_n$. These measurements a considered as realizations of MVN random variables $Y_1, \dots, Y_n$ with mean μ and covariance matrix Σ_n as shown in Figure 6.12, Figure 6.13 or Figure 6.14.

In Kriging, a more general covariance matrix (or equivalently, a correlation matrix Ψ) is used, see Equation 6.12. Using a maximum likelihood approach, we can estimate the unknown parameters μ and Σ_n from the data so that the likelihood function is maximized.

Definition 6.31 (The Kriging Basis Functions)**.** Kriging uses k-dimensional basis functions of the form

$$\psi(\vec{x}^{(i)}, \vec{x}^{(j)}) = \exp\left(-\sum_{l=1}^{k} \theta_l |x_l^{(i)} - x_l^{(j)}|^{p_l}\right), \tag{6.12}$$

where $\vec{x}^{(i)}$ denotes the k-dim vector $\vec{x}^{(i)} = (x_1^{(i)}, \dots, x_k^{(i)})^T$.

6.12.3. The Kriging Model

Consider sample data $\vec{X}$ and $\vec{y}$ from n locations that are available in matrix form: $\vec{X}$ is a $(n \times k)$ matrix, where k denotes the problem dimension and $\vec{y}$ is a $(n \times 1)$ vector. We want to find an expression for a predicted values at a new point $\vec{x}$, denoted as $\hat{y}$.

We start with an abstract, not really intuitive concept: The observed responses $\vec{y}$ are considered as if they are from a stochastic process, which will be denoted as

$$\begin{pmatrix} \vec{Y}(\vec{x}^{(1)}) \\ \vdots \\ \vec{Y}(\vec{x}^{(n)}) \end{pmatrix}. \tag{6.13}$$

The set of random vectors from Equation 6.13 (also referred to as a *random field*) has a mean of $\vec{1}\mu$, which is a $(n \times 1)$ vector. The random vectors are correlated with each other using the basis function expression from Equation 6.12:

$$\operatorname{cor}\left(\vec{Y}(\vec{x}^{(i)}), \vec{Y}(\vec{x}^{(l)})\right) = \exp\left\{-\sum_{j=1}^{k} \theta_j |x_j^{(i)} - x_j^{(l)}|^{p_j}\right\}. \tag{6.14}$$

Using Equation 6.14, we can compute the $(n \times n)$ correlation matrix $\vec{\Psi}$ of the observed sample data as shown in Equation 6.15,

$$\vec{\Psi} = \begin{pmatrix} \operatorname{cor}\left(\vec{Y}(\vec{x}^{(1)}), \vec{Y}(\vec{x}^{(1)})\right) & \dots & \operatorname{cor}\left(\vec{Y}(\vec{x}^{(1)}), \vec{Y}(\vec{x}^{(n)})\right) \\ \vdots & \vdots & \vdots \\ \operatorname{cor}\left(\vec{Y}(\vec{x}^{(n)}), \vec{Y}(\vec{x}^{(1)})\right) & \dots & \operatorname{cor}\left(\vec{Y}(\vec{x}^{(n)}), \vec{Y}(\vec{x}^{(n)})\right) \end{pmatrix}, \tag{6.15}$$

and a covariance matrix as shown in Equation 6.16,

$$\operatorname{Cov}(\mathbf{Y}, \mathbf{Y}) = \sigma^2 . \tag{6.16}$$

This assumed correlation between the sample data reflects our expectation that an engineering function will behave in a certain way and it will be smoothly and continuous.

We now have a set of n random variables ($\mathbf{Y}$) that are correlated with each other as described in the $(n \times n)$ correlation matrix , see Equation 6.15. The correlations depend on the absolute distances in dimension j between the i-th and the l-th sample point $|x_j^{(i)} - x_j^{(l)}|$ and the corresponding parameters p_j and θ_j for dimension j. The correlation is intuitive, because when

- two points move close together, then $|x_j^{(i)} - x_j| \to 0$ and $\exp\left(-|x_j^{(i)} - x_j|^{p_j}\right) \to 1$ (these points show very close correlation and $Y(x_j^{(i)}) = Y(x_j)$).
- two points move far apart, then $|x_j^{(i)} - x_j| \to \infty$ and $\exp\left(-|x_j^{(i)} - x_j|^{p_j}\right) \to 0$ (these points show very low correlation).

Example 6.23 (Correlations for Different p_j)**.** Three different correlations are shown in Figure 6.7: $p_j = 0.1, 1, 2$. The smoothness parameter p_j affects the correlation:

- With $p_j = 0.1$, there is basicaly no immediate correlation between the points and there is a near discontinuity between the points $Y(\vec{x}_j^{(i)})$ and $Y(\vec{x}_j)$.
- With $p_j = 2$, the correlation is more smooth and we have a continuous gradient through $x_j^{(i)} - x_j$.

Reducing p_j increases the rate at which the correlation initially drops with distance. This is shown in Figure 6.7.

Example 6.24 (Correlations for Different θ)**.** Figure 6.8 visualizes the correlation between two points $Y(\vec{x}_j^{(i)})$ and $Y(\vec{x}_j)$ for different values of θ. The parameter θ can be seen as a width parameter:

- low θ_j means that all points will have a high correlation, with $Y(x_j)$ being similar across the sample.
- high θ_j means that there is a significant difference between the $Y(x_j)$'s.
- θ_j is a measure of how active the function we are approximating is.
- High θ_j indicate important parameters, see Figure 6.8.

Considering the activity parameter θ is useful in high-dimensional problems where it is difficult to visualize the design landscape and the effect of the variable is unknown. By examining the elements of the vector $\vec{\theta}$, we can identify the most important variables and focus on them. This is a crucial step in the optimization process, as it allows us to reduce the dimensionality of the problem and focus on the most important variables.

Figure 6.7.: Correlations with varying θ. θ set to 1/10, 1, and 10.

Figure 6.8.: Correlations with varying θ. θ set to 1/10, 1, and 10.

Example 6.25 (Example: The Correlation Matrix (Detailed Computation))**.** Let $n = 4$ and $k = 3$. The sample plan is represented by the following matrix X:

$$X = \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \\ x_{41} & x_{42} & x_{43} \end{pmatrix}$$

To compute the elements of the matrix Ψ, the following k (one for each of the k dimensions) (n, n)-matrices have to be computed:

$$D_1 = \begin{pmatrix} x_{11} - x_{11} & x_{11} - x_{21} & x_{11} - x_{31} & x_{11} - x_{41} \\ x_{21} - x_{11} & x_{21} - x_{21} & x_{21} - x_{31} & x_{21} - x_{41} \\ x_{31} - x_{11} & x_{31} - x_{21} & x_{31} - x_{31} & x_{31} - x_{41} \\ x_{41} - x_{11} & x_{41} - x_{21} & x_{41} - x_{31} & x_{41} - x_{41} \end{pmatrix}$$

$$D_2 = \begin{pmatrix} x_{12} - x_{12} & x_{12} - x_{22} & x_{12} - x_{32} & x_{12} - x_{42} \\ x_{22} - x_{12} & x_{22} - x_{22} & x_{22} - x_{32} & x_{22} - x_{42} \\ x_{32} - x_{12} & x_{32} - x_{22} & x_{32} - x_{32} & x_{32} - x_{42} \\ x_{42} - x_{12} & x_{42} - x_{22} & x_{42} - x_{32} & x_{42} - x_{42} \end{pmatrix}$$

$$D_3 = \begin{pmatrix} x_{13} - x_{13} & x_{13} - x_{23} & x_{13} - x_{33} & x_{13} - x_{43} \\ x_{23} - x_{13} & x_{23} - x_{23} & x_{23} - x_{33} & x_{23} - x_{43} \\ x_{33} - x_{13} & x_{33} - x_{23} & x_{33} - x_{33} & x_{33} - x_{43} \\ x_{43} - x_{13} & x_{43} - x_{23} & x_{43} - x_{33} & x_{43} - x_{43} \end{pmatrix}$$

Since the matrices are symmetric and the main diagonals are zero, it is sufficient to compute the following matrices:

$$D_1 = \begin{pmatrix} 0 & x_{11} - x_{21} & x_{11} - x_{31} & x_{11} - x_{41} \\ 0 & 0 & x_{21} - x_{31} & x_{21} - x_{41} \\ 0 & 0 & 0 & x_{31} - x_{41} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & x_{12} - x_{22} & x_{12} - x_{32} & x_{12} - x_{42} \\ 0 & 0 & x_{22} - x_{32} & x_{22} - x_{42} \\ 0 & 0 & 0 & x_{32} - x_{42} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & x_{13} - x_{23} & x_{13} - x_{33} & x_{13} - x_{43} \\ 0 & 0 & x_{23} - x_{33} & x_{23} - x_{43} \\ 0 & 0 & 0 & x_{33} - x_{43} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

We will consider $p_l = 2$. The differences will be squared and multiplied by θ_i, i.e.:

$$D_1 = \theta_1 \begin{pmatrix} 0 & (x_{11}-x_{21})^2 & (x_{11}-x_{31})^2 & (x_{11}-x_{41})^2 \\ 0 & 0 & (x_{21}-x_{31})^2 & (x_{21}-x_{41})^2 \\ 0 & 0 & 0 & (x_{31}-x_{41})^2 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$D_2 = \theta_2 \begin{pmatrix} 0 & (x_{12}-x_{22})^2 & (x_{12}-x_{32})^2 & (x_{12}-x_{42})^2 \\ 0 & 0 & (x_{22}-x_{32})^2 & (x_{22}-x_{42})^2 \\ 0 & 0 & 0 & (x_{32}-x_{42})^2 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$D_3 = \theta_3 \begin{pmatrix} 0 & (x_{13}-x_{23})^2 & (x_{13}-x_{33})^2 & (x_{13}-x_{43})^2 \\ 0 & 0 & (x_{23}-x_{33})^2 & (x_{23}-x_{43})^2 \\ 0 & 0 & 0 & (x_{33}-x_{43})^2 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

The sum of the three matrices $D = D_1 + D_2 + D_3$ will be calculated next:

$$\begin{pmatrix} 0 & \theta_1(x_{11}-x_{21})^2 + \theta_2(x_{12}-x_{22})^2 + \theta_3(x_{13}-x_{23})^2 & \theta_1(x_{11}-x_{31})^2 + \theta_2(x_{12}-x_{32})^2 + \theta_3(x_{13}-x_{33} \\ 0 & 0 & \theta_1(x_{21}-x_{31})^2 + \theta_2(x_{22}-x_{32})^2 + \theta_3(x_{23}-x_{33} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Finally,

$$\Psi = \exp(-D)$$

is computed.

Next, we will demonstrate how this computation can be implemented in Python. We will consider four points in three dimensions and compute the correlation matrix Ψ using the basis function from Equation 6.12. These points are placed at the origin, at the unit vectors, and at the points $(100, 100, 100)$ and $(101, 100, 100)$. So, they form two clusters: one at the origin and one at $(100, 100, 100)$.

```
from numpy import (array, zeros, power, ones, exp, multiply,
                   eye, linspace, spacing, sqrt, arange,
                   append, ravel)
from numpy.linalg import cholesky, solve
theta = np.array([1,2,3])
X = np.array([ [1,0,0], [0,1,0], [100, 100, 100], [101, 100, 100]])
X
```

```
array([[  1,   0,   0],
       [  0,   1,   0],
       [100, 100, 100],
       [101, 100, 100]])
```

```
def build_Psi(X, theta):
    n = X.shape[0]
    k = X.shape[1]
    D = zeros((k, n, n))
    for l in range(k):
        for i in range(n):
            for j in range(i, n):
                D[l, i, j] = theta[l]*(X[i,l] - X[j,l])**2
    D = sum(D)
    D = D + D.T
    return exp(-D)
```

```
Psi = build_Psi(X, theta)
Psi
```

```
array([[1.        , 0.04978707, 0.        , 0.        ],
       [0.04978707, 1.        , 0.        , 0.        ],
       [0.        , 0.        , 1.        , 0.36787944],
       [0.        , 0.        , 0.36787944, 1.        ]])
```

Example 6.26 (Example: The Correlation Matrix (Using Existing Functions))**.** The same result as computed in the previous example can be obtained with existing python functions, e.g., from the package `scipy`.

```
from scipy.spatial.distance import squareform
from scipy.spatial.distance import pdist

def build_Psi(X, theta, eps=sqrt(spacing(1))):
    return exp(- squareform(pdist(X,
                            metric='sqeuclidean',
                            out=None,
                            w=theta))) +  multiply(eye(X.shape[0]),
                                                   eps)

Psi = build_Psi(X, theta, eps=.0)
Psi
```

```
array([[1.        , 0.04978707, 0.        , 0.        ],
```

```
[0.04978707, 1.        , 0.        , 0.        ],
[0.        , 0.        , 1.        , 0.36787944],
[0.        , 0.        , 0.36787944, 1.        ]])
```

6.12.4. The Condition Number

A small value, `eps`, can be passed to the function `build_Psi` to improve the condition number. For example, `eps=sqrt(spacing(1))` can be used. The numpy function `spacing()` returns the distance between a number and its nearest adjacent number.

The condition number of a matrix is a measure of its sensitivity to small changes in its elements. It is used to estimate how much the output of a function will change if the input is slightly altered.

A matrix with a low condition number is well-conditioned, which means its behavior is relatively stable, while a matrix with a high condition number is ill-conditioned, meaning its behavior is unstable with respect to numerical precision.

```
import numpy as np

# Define a well-conditioned matrix (low condition number)
A = np.array([[1, 0.1], [0.1, 1]])
print("Condition number of A: ", np.linalg.cond(A))

# Define an ill-conditioned matrix (high condition number)
B = np.array([[1, 0.99999999], [0.99999999, 1]])
print("Condition number of B: ", np.linalg.cond(B))
```

```
Condition number of A:  1.2222222222222225
Condition number of B:  200000000.57495335
```

```
np.linalg.cond(Psi)
```

```
np.float64(2.163953413738652)
```

6.12.5. MLE to estimate θ and p

Until now, the observed data $\vec{y}$ was not used. We know what the correlations mean, but how do we estimate the values of θ_j and where does our observed data y come

in? To estimate the values of $\vec{\theta}$ and $\vec{p}$, they are chosen to maximize the likelihood of $\vec{y}$, which can be expressed in terms of the sample data

$$L\left(\vec{Y}(\vec{x}^{(1)}), \dots, \vec{Y}(\vec{x}^{(n)})|\mu, \sigma\right) = \frac{1}{(2\pi\sigma)^{n/2}|\vec{\Psi}|^{1/2}} \exp\left\{\frac{-(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)}{2\sigma^2}\right\},$$

and formulated as the log-likelihood:

$$\ln(L) = -\frac{n}{2}\ln(2\pi\sigma) - \frac{1}{2}\ln|\vec{\Psi}|\frac{-(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)}{2\sigma^2}. \tag{6.17}$$

Optimization of the log-likelihood by taking derivatives with respect to μ and σ results in

$$\hat{\mu} = \frac{\vec{1}^T\vec{\Psi}^{-1}\vec{y}^T}{\vec{1}^T\vec{\Psi}^{-1}\vec{1}^T} \tag{6.18}$$

and

$$\hat{\sigma}^2 = \frac{(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)}{n}. \tag{6.19}$$

Combining the equations, i.e., substituting Equation 6.18 and Equation 6.19 inti Equation 6.17 leads to the concentrated log-likelihood function:

$$\ln(L) = -\frac{n}{2}\ln(\hat{\sigma}) - \frac{1}{2}\ln|\vec{\Psi}|. \tag{6.20}$$

Remark 6.1 (The Concentrated Log-Likelihood).

- The first term in Equation 6.20 requires information about the measured point (observations) y_i.
- To maximize $\ln(L)$, optimal values of $\vec{\theta}$ and $\vec{p}$ are determined numerically, because the function (Equation 6.20) is not differentiable.

The concentrated log-likelihood function is very quick to compute. We do not need a statistical model, because we are only interested in the maximum likelihood estimate (MLE) of θ and p. Optimizers such as Nelder-Mead, Conjugate Gradient, or Simulated Annealing can be used to determine optimal values for θ and p. After the optimization, the correlation matrix Ψ is build with the optimized θ and p values. This is best (most likely) Kriging model for the given data y.

Observing Figure 6.8, there's significant change between $\theta = 0.1$ and $\theta = 1$, just as there is between $\theta = 1$ and $\theta = 10$. Hence, it is sensible to search for θ on a logarithmic scale. Suitable search bounds typically range from 10^{-3} to 10^2, although this is not a stringent requirement. Importantly, the scaling of the observed data does not affect the values of $\hat{\theta}$, but the scaling of the design space does. Therefore, it is advisable to consistently scale variable ranges between zero and one to ensure consistency in the degree of activity $\hat{\theta}_j$ represents across different problems.

Optimizing $\hat{\phi}$ can enhance prediction accuracy across various problems.

6.12.6. Implementing an MLE of the Model Parameters

The matrix algebra necessary for calculating the likelihood is the most computationally intensive aspect of the Kriging process. It's crucial to ensure that the code implementation is as efficient as possible.

Given that Ψ (our correlation matrix) is symmetric, only half of the matrix needs to be computed before adding it to its transpose. When calculating the log-likelihood, several matrix inversions are required. The fastest approach is to conduct one Cholesky factorization and then apply backward and forward substitution for each inverse.

The Cholesky factorization is applicable only to positive-definite matrices, which Ψ generally is. However, if Ψ becomes nearly singular, such as when the $\mathbf{x}^{(i)}$'s are densely packed, the Cholesky factorization might fail. In these cases, one could employ an LU-decomposition, though the result might be unreliable. When Ψ is near singular, the best course of action is to either use regression techniques or, as we do here, assign a poor likelihood value to parameters generating the near singular matrix, thus diverting the MLE search towards better-conditioned Ψ matrices.

Another consideration in calculating the concentrated log-likelihood is that $\det(\Psi) \to 0$ for poorly conditioned matrices, so it is advisable to use twice the sum of the logarithms of the diagonal of the Cholesky factorization when calculating $\ln(|\Psi|)$ in Equation 6.20.

6.12.7. Kriging Prediction

We will use the Kriging correlation Ψ to predict new values based on the observed data. The matrix algebra involved for calculating the likelihood is the most computationally intensive part of the Kriging process. Care must be taken that the computer code is as efficient as possible.

Basic elements of the Kriging based surrogate optimization such as interpolation, expected improvement, and regression are presented. The presentation follows the approach described in Forrester, Sóbester, and Keane (2008) and Bartz et al. (2022).

Main idea for prediction is that the new $Y(\vec{x})$ should be consistent with the old sample data X. For a new prediction $\hat{y}$ at $\vec{x}$, the value of $\hat{y}$ is chosen so that it maximizes the likelihood of the sample data $\vec{X}$ and the prediction, given the (optimized) correlation parameter $\vec{\theta}$ and $\vec{p}$ from above. The observed data $\vec{y}$ is augmented with the new prediction $\hat{y}$ which results in the augmented vector $\vec{\tilde{y}} = (\vec{y}^T, \hat{y})^T$. A vector of correlations between the observed data and the new prediction is defined as

$$\vec{\psi} = \begin{pmatrix} \operatorname{cor}\left(\vec{Y}(\vec{x}^{(1)}), \vec{Y}(\vec{x})\right) \\ \vdots \\ \operatorname{cor}\left(\vec{Y}(\vec{x}^{(n)}), \vec{Y}(\vec{x})\right) \end{pmatrix} = \begin{pmatrix} \vec{\psi}^{(1)} \\ \vdots \\ \vec{\psi}^{(n)} \end{pmatrix}.$$

Definition 6.32 (The Augmented Correlation Matrix)**.** The augmented correlation matrix is constructed as

$$\tilde{\vec{\Psi}} = \begin{pmatrix} \vec{\Psi} & \vec{\psi} \\ \vec{\psi}^T & 1 \end{pmatrix}.$$

The log-likelihood of the augmented data is

$$\ln(L) = -\frac{n}{2}\ln(2\pi) - \frac{n}{2}\ln(\hat{\sigma}^2) - \frac{1}{2}\ln|\tilde{\vec{\Psi}}| - \frac{(\tilde{\vec{y}} - \vec{1}\hat{\mu})^T \tilde{\vec{\Psi}}^{-1} (\tilde{\vec{y}} - \vec{1}\hat{\mu})}{2\hat{\sigma}^2}, \tag{6.21}$$

where $\vec{1}$ is a vector of ones and $\hat{\mu}$ and $\hat{\sigma}^2$ are the MLEs from Equation 6.18 and Equation 6.19. Only the last term in Equation 6.21 depends on $\hat{y}$, so we need only consider this term in the maximization. Details cen be found in Forrester, Sóbester, and Keane (2008). Finally, the MLE for $\hat{y}$ can be calculated as

$$\hat{y}(\vec{x}) = \hat{\mu} + \vec{\psi}^T \tilde{\vec{\Psi}}^{-1} (\vec{y} - \vec{1}\hat{\mu}). \tag{6.22}$$

Equation 6.22 reveals two important properties of the Kriging predictor:

- Basis functions: The basis function impacts the vector $\vec{\psi}$, which contains the n correlations between the new point $\vec{x}$ and the observed locations. Values from the n basis functions are added to a mean base term μ with weightings

$$\vec{w} = \tilde{\vec{\Psi}}^{(-1)} (\vec{y} - \vec{1}\hat{\mu}).$$

- Interpolation: The predictions interpolate the sample data. When calculating the prediction at the ith sample point, $\vec{x}^{(i)}$, the ith column of $\vec{\Psi}^{-1}$ is $\vec{\psi}$, and $\vec{\psi}\vec{\Psi}^{-1}$ is the ith unit vector. Hence,

$$\hat{y}(\vec{x}^{(i)}) = y^{(i)}.$$

6.13. Kriging Example: Sinusoid Function

Toy example in 1d where the response is a simple sinusoid measured at eight equally spaced x-locations in the span of a single period of oscillation.

6.13.1. Calculating the Correlation Matrix Ψ

The correlation matrix Ψ is based on the pairwise squared distances between the input locations. Here we will use $n = 8$ sample locations and θ is set to 1.0.

```
n = 8
X = np.linspace(0, 2*np.pi, n, endpoint=False).reshape(-1,1)
print(np.round(X, 2))
```

```
[[0.  ]
 [0.79]
 [1.57]
 [2.36]
 [3.14]
 [3.93]
 [4.71]
 [5.5 ]]
```

Evaluate at sample points

```
y = np.sin(X)
print(np.round(y, 2))
```

```
[[ 0.  ]
 [ 0.71]
 [ 1.  ]
 [ 0.71]
 [ 0.  ]
 [-0.71]
 [-1.  ]
 [-0.71]]
```

We have the data points shown in Table 6.1.

Table 6.1.: Data points for the sinusoid function

x	y
0.0	0.0
0.79	0.71
1.57	1.0
2.36	0.71
3.14	0.0
3.93	-0.71
4.71	-1.0
5.5	-0.71

The data points are visualized in Figure 6.9.

```
import matplotlib.pyplot as plt
plt.plot(X, y, "bo")
plt.title(f"Sin(x) evaluated at {n} points")
plt.grid()
plt.show()
```

Figure 6.9.: Sin(x) evaluated at 8 points.

6.13.2. Computing the Ψ Matrix

```
# theta should be an array (of one value, for the moment, will be changed later)
theta = np.array([1.0])
Psi = build_Psi(X, theta)
print(np.round(Psi, 2))
```

```
[[1.   0.54 0.08 0.   0.   0.   0.   0.  ]
 [0.54 1.   0.54 0.08 0.   0.   0.   0.  ]
 [0.08 0.54 1.   0.54 0.08 0.   0.   0.  ]
 [0.   0.08 0.54 1.   0.54 0.08 0.   0.  ]
 [0.   0.   0.08 0.54 1.   0.54 0.08 0.  ]
```

```
 [0.   0.   0.   0.08 0.54 1.   0.54 0.08]
 [0.   0.   0.   0.   0.08 0.54 1.   0.54]
 [0.   0.   0.   0.   0.   0.08 0.54 1.  ]]
```

6.13.3. Selecting the New Locations

We would like to predict at $m = 100$ new locations (or testign locations) in the interval $[0, 2\pi]$. The new locations are stored in the variable `x`.

```
m = 100
x = np.linspace(0, 2*np.pi, m, endpoint=False).reshape(-1,1)
```

6.13.4. Computing the ψ Vector

Distances between testing locations x and training data locations X.

```
from scipy.spatial.distance import cdist

def build_psi(X, x, theta, eps=sqrt(spacing(1))):
    n = X.shape[0]
    k = X.shape[1]
    m = x.shape[0]
    psi = zeros((n, m))
    theta = theta * ones(k)
    D = zeros((n, m))
    D = cdist(x.reshape(-1, k),
              X.reshape(-1, k),
              metric='sqeuclidean',
              out=None,
              w=theta)
    psi = exp(-D)
    # return psi transpose to be consistent with the literature
    print(f"Dimensions of psi: {psi.T.shape}")
    return(psi.T)

psi = build_psi(X, x, theta)
```

```
Dimensions of psi: (8, 100)
```

6.13.5. Predicting at New Locations

Computation of the predictive equations.

```
U = cholesky(Psi).T
one = np.ones(n).reshape(-1,1)
mu = (one.T.dot(solve(U, solve(U.T, y)))) / one.T.dot(solve(U, solve(U.T, one)))
f = mu * ones(m).reshape(-1,1) + psi.T.dot(solve(U, solve(U.T, y - one * mu)))
print(f"Dimensions of f: {f.shape}")
```

```
Dimensions of f: (100, 1)
```

To compute f, Equation 6.22 is used.

6.13.6. Visualization

```
import matplotlib.pyplot as plt
plt.plot(x, f, color = "orange", label="Fitted")
plt.plot(x, np.sin(x), color = "grey", label="Original")
plt.plot(X, y, "bo", label="Measurements")
plt.title("Kriging prediction of sin(x) with {} points.\n theta: {}".format(n, theta[0]))
plt.legend(loc='upper right')
plt.show()
```

6.14. Cholesky Decomposition

6.14.1. Example of Cholesky Decomposition

We consider dimension $k = 1$ and $n = 2$ sample points. The sample points are located at $x_1 = 1$ and $x_2 = 5$. The response values are $y_1 = 2$ and $y_2 = 10$. The correlation parameter is $\theta = 1$ and p is set to 1. Using Equation 6.12, we can compute the correlation matrix Ψ:

$$\Psi = \begin{pmatrix} 1 & e^{-1} \\ e^{-1} & 1 \end{pmatrix}.$$

To determine MLE as in Equation 6.22, we need to compute Ψ^{-1}:

$$\Psi^{-1} = \frac{e}{e^2 - 1} \begin{pmatrix} e & -1 \\ -1 & e \end{pmatrix}.$$

Cholesky-decomposition of Ψ is recommended to compute Ψ^{-1}. Cholesky decomposition is a decomposition of a positive definite symmetric matrix into the product of

a lower triangular matrix L, a diagonal matrix D and the transpose of L, which is denoted as L^T. Consider the following example:

$$LDL^T = \begin{pmatrix} 1 & 0 \\ l_{21} & 1 \end{pmatrix} \begin{pmatrix} d_{11} & 0 \\ 0 & d_{22} \end{pmatrix} \begin{pmatrix} 1 & l_{21} \\ 0 & 1 \end{pmatrix} =$$

$$\begin{pmatrix} d_{11} & 0 \\ d_{11}l_{21} & d_{22} \end{pmatrix} \begin{pmatrix} 1 & l_{21} \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} d_{11} & d_{11}l_{21} \\ d_{11}l_{21} & d_{11}l_{21}^2 + d_{22} \end{pmatrix}. \tag{6.23}$$

Using Equation 6.23, we can compute the Cholesky decomposition of Ψ:

1. $d_{11} = 1$,
2. $l_{21}d_{11} = e^{-1} \Rightarrow l_{21} = e^{-1}$, and
3. $d_{11}l_{21}^2 + d_{22} = 1 \Rightarrow d_{22} = 1 - e^{-2}$.

The Cholesky decomposition of Ψ is

$$\Psi = \begin{pmatrix} 1 & 0 \\ e^{-1} & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 - e^{-2} \end{pmatrix} \begin{pmatrix} 1 & e^{-1} \\ 0 & 1 \end{pmatrix} = LDL^T$$

Some programs use U instead of L. The Cholesky decomposition of Ψ is

$$\Psi = LDL^T = U^T DU.$$

Using

$$\sqrt{D} = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1 - e^{-2}} \end{pmatrix},$$

we can write the Cholesky decomposition of Ψ without a diagonal matrix D as

$$\Psi = \begin{pmatrix} 1 & 0 \\ e^{-1} & \sqrt{1 - e^{-2}} \end{pmatrix} \begin{pmatrix} 1 & e^{-1} \\ 0 & \sqrt{1 - e^{-2}} \end{pmatrix} = U^T U.$$

6.14.2. Inverse Matrix Using Cholesky Decomposition

To compute the inverse of a matrix using the Cholesky decomposition, you can follow these steps:

1. Decompose the matrix A into L and L^T, where L is a lower triangular matrix and L^T is the transpose of L.
2. Compute L^{-1}, the inverse of L.
3. The inverse of A is then $(L^{-1})^T L^-1$.

Please note that this method only applies to symmetric, positive-definite matrices.

The inverse of the matrix Ψ from above is:

$$\Psi^{-1} = \frac{e}{e^2 - 1} \begin{pmatrix} e & -1 \\ -1 & e \end{pmatrix}.$$

Here's an example of how to compute the inverse of a matrix using Cholesky decomposition in Python:

```
import numpy as np
from scipy.linalg import cholesky, inv
E = np.exp(1)

# Psi is a symmetric, positive-definite matrix
Psi = np.array([[1, 1/E], [1/E, 1]])
L = cholesky(Psi, lower=True)
L_inv = inv(L)
# The inverse of A is (L^-1)^T * L^-1
Psi_inv = np.dot(L_inv.T, L_inv)

print("Psi:\n", Psi)
print("Psi Inverse:\n", Psi_inv)
```

```
Psi:
 [[1.         0.36787944]
 [0.36787944 1.        ]]
Psi Inverse:
 [[ 1.15651764 -0.42545906]
 [-0.42545906  1.15651764]]
```

6.15. Gaussian Processes—Some Background Information

The concept of GP (Gaussian Process) regression can be understood as a simple extension of linear modeling. It is worth noting that this approach goes by various names and acronyms, including "kriging," a term derived from geostatistics, as introduced by Matheron in 1963. Additionally, it is referred to as Gaussian spatial modeling or a Gaussian stochastic process, and machine learning (ML) researchers often use the term Gaussian process regression (GPR). In all of these instances, the central focus is on regression. This involves training on both inputs and outputs, with the ultimate

objective of making predictions and quantifying uncertainty (referred to as uncertainty quantification or UQ).

However, it's important to emphasize that GPs are not a universal solution for every problem. Specialized tools may outperform GPs in specific, non-generic contexts, and GPs have their own set of limitations that need to be considered.

6.15.1. Gaussian Process Prior

In the context of GP, any finite collection of realizations, which is represented by n observations, is modeled as having a multivariate normal (MVN) distribution. The characteristics of these realizations can be fully described by two key parameters:

1. Their mean, denoted as an n-vector μ.
2. The covariance matrix, denoted as an $n \times n$ matrix Σ. This covariance matrix encapsulates the relationships and variability between the individual realizations within the collection.

6.15.2. Covariance Function

The covariance function is defined by inverse exponentiated squared Euclidean distance:

$$\Sigma(\vec{x}, \vec{x}') = \exp\{-||\vec{x} - \vec{x}'||^2\},$$

where $\vec{x}$ and $\vec{x}'$ are two points in the k-dimensional input space and $\| \cdot \|$ denotes the Euclidean distance, i.e.,

$$||\vec{x} - \vec{x}'||^2 = \sum_{i=1}^{k} (x_i - x_i')^2.$$

An 1-d example is shown in Figure 6.10.

The covariance function is also referred to as the kernel function. The *Gaussian* kernel uses an additional parameter, σ^2, to control the rate of decay. This parameter is referred to as the length scale or the characteristic length scale. The covariance function is then defined as

$$\Sigma(\vec{x}, \vec{x}') = \exp\{-||\vec{x} - \vec{x}'||^2/(2\sigma^2)\}. \tag{6.24}$$

The covariance decays exponentially fast as $\vec{x}$ and $\vec{x}'$ become farther apart. Observe that

$$\Sigma(\vec{x}, \vec{x}) = 1$$

and

Figure 6.10.: One-dim inverse exponentiated squared Euclidean distance

$$\Sigma(\vec{x}, \vec{x}') < 1$$

for $\vec{x} \neq \vec{x}'$. The function $\Sigma(\vec{x}, \vec{x}')$ must be positive definite.

Remark 6.2 (Kriging and Gaussian Basis Functions). The Kriging basis function (Equation 6.12) is related to the 1-dim Gaussian basis function (Equation 6.24), which is defined as

$$\Sigma(\vec{x}^{(i)}, \vec{x}^{(j)}) = \exp\{-||\vec{x}^{(i)} - \vec{x}^{(j)}||^2/(2\sigma^2)\}. \tag{6.25}$$

There are some differences between Gaussian basis functions and Kriging basis functions:

- Where the Gaussian basis function has $1/(2\sigma^2)$, the Kriging basis has a vector $\theta = [\theta_1, \theta_2, \dots, \theta_k]^T$.
- The θ vector allows the width of the basis function to vary from dimension to dimension.
- In the Gaussian basis function, the exponent is fixed at 2, Kriging allows this exponent p_l to vary (typically from 1 to 2).

6.15.2.1. Positive Definiteness

Positive definiteness in the context of the covariance matrix Σ_n is a fundamental requirement. It is determined by evaluating $\Sigma(x_i, x_j)$ at pairs of n $\vec{x}$-values, denoted as $\vec{x}_1, \vec{x}_2, \dots, \vec{x}_n$. The condition for positive definiteness is that for all $\vec{x}$ vectors that are not equal to zero, the expression $\vec{x}^\top \Sigma_n \vec{x}$ must be greater than zero. This property is essential when intending to use Σ_n as a covariance matrix in multivariate normal (MVN) analysis. It is analogous to the requirement in univariate Gaussian distributions where the variance parameter, σ^2, must be positive.

Gaussian Processes (GPs) can be effectively utilized to generate random data that follows a smooth functional relationship. The process involves the following steps:

1. Select a set of $\vec{x}$-values, denoted as $\vec{x}_1, \vec{x}_2, \dots, \vec{x}_n$.
2. Define the covariance matrix Σ_n by evaluating $\Sigma_n^{ij} = \Sigma(\vec{x}_i, \vec{x}_j)$ for $i, j = 1, 2, \dots, n$.
3. Generate an n-variate realization Y that follows a multivariate normal distribution with a mean of zero and a covariance matrix Σ_n, expressed as $Y \sim \mathcal{N}_n(0, \Sigma_n)$.
4. Visualize the result by plotting it in the x-y plane.

6.15.3. Construction of the Covariance Matrix

Here is an one-dimensional example. The process begins by creating an input grid using $\vec{x}$-values. This grid consists of 100 elements, providing the basis for further analysis and visualization.

```
import numpy as np
n = 100
X = np.linspace(0, 10, n, endpoint=False).reshape(-1,1)
```

In the context of this discussion, the construction of the covariance matrix, denoted as Σ_n, relies on the concept of inverse exponentiated squared Euclidean distances. However, it's important to note that a modification is introduced later in the process. Specifically, the diagonal of the covariance matrix is augmented with a small value, represented as "eps" or ϵ.

The reason for this augmentation is that while inverse exponentiated distances theoretically ensure the covariance matrix's positive definiteness, in practical applications, the matrix can sometimes become numerically ill-conditioned. By adding a small value to the diagonal, such as ϵ, this ill-conditioning issue is mitigated. In this context, ϵ is often referred to as "jitter."

```
import numpy as np
from numpy import array, zeros, power, ones, exp, multiply, eye, linspace, spacing, so
from numpy.linalg import cholesky, solve
from numpy.random import multivariate_normal
def build_Sigma(X, sigma2):
    n = X.shape[0]
    k = X.shape[1]
    D = zeros((k, n, n))
    for l in range(k):
        for i in range(n):
            for j in range(i, n):
                D[l, i, j] = 1/(2*sigma2[l])*(X[i,l] - X[j,l])**2
    D = sum(D)
    D = D + D.T
    return exp(-D)
```

```
sigma2 = np.array([1.0])
Sigma = build_Sigma(X, sigma2)
np.round(Sigma[:3,:], 3)
```

```
array([[1.   , 0.995, 0.98 , 0.956, 0.923, 0.882, 0.835, 0.783, 0.726,
        0.667, 0.607, 0.546, 0.487, 0.43 , 0.375, 0.325, 0.278, 0.236,
        0.198, 0.164, 0.135, 0.11 , 0.089, 0.071, 0.056, 0.044, 0.034,
        0.026, 0.02 , 0.015, 0.011, 0.008, 0.006, 0.004, 0.003, 0.002,
        0.002, 0.001, 0.001, 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   ],
       [0.995, 1.   , 0.995, 0.98 , 0.956, 0.923, 0.882, 0.835, 0.783,
        0.726, 0.667, 0.607, 0.546, 0.487, 0.43 , 0.375, 0.325, 0.278,
        0.236, 0.198, 0.164, 0.135, 0.11 , 0.089, 0.071, 0.056, 0.044,
        0.034, 0.026, 0.02 , 0.015, 0.011, 0.008, 0.006, 0.004, 0.003,
        0.002, 0.002, 0.001, 0.001, 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
```

```
 0.   ],
[0.98 , 0.995, 1.   , 0.995, 0.98 , 0.956, 0.923, 0.882, 0.835,
 0.783, 0.726, 0.667, 0.607, 0.546, 0.487, 0.43 , 0.375, 0.325,
 0.278, 0.236, 0.198, 0.164, 0.135, 0.11 , 0.089, 0.071, 0.056,
 0.044, 0.034, 0.026, 0.02 , 0.015, 0.011, 0.008, 0.006, 0.004,
 0.003, 0.002, 0.002, 0.001, 0.001, 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
 0.   ]])
```

```
import matplotlib.pyplot as plt
plt.imshow(Sigma, cmap='hot', interpolation='nearest')
plt.colorbar()
plt.show()
```

6.15.4. Generation of Random Samples and Plotting the Realizations of the Random Function

In the context of the multivariate normal distribution, the next step is to utilize the previously constructed covariance matrix denoted as `Sigma`. It is used as an essential component in generating random samples from the multivariate normal distribution.

The function `multivariate_normal` is employed for this purpose. It serves as a random number generator specifically designed for the multivariate normal distribution. In this case, the mean of the distribution is set equal to `mean`, and the covariance matrix is provided as `Psi`. The argument `size` specifies the number of realizations, which, in this specific scenario, is set to one.

By default, the mean vector is initialized to zero. To match the number of samples, which is equivalent to the number of rows in the `X` and `Sigma` matrices, the argument `zeros(n)` is used, where `n` represents the number of samples (here taken from the size of the matrix, e.g.,: `Sigma.shape[0]`).

```
rng = np.random.default_rng(seed=12345)
Y = rng.multivariate_normal(zeros(Sigma.shape[0]), Sigma, size = 1, check_valid="raise
Y.shape
```

```
(100, 1)
```

Now we can plot the results, i.e., a finite realization of the random function $Y()$ under a GP prior with a particular covariance structure. We will plot those `X` and `Y` pairs as connected points on an x-y plane.

```
import matplotlib.pyplot as plt
plt.plot(X, Y)
plt.title("Realization of Random Functions under a GP prior.\n sigma2: {}".format(sign
plt.show()
```

Figure 6.11.: Realization of one random function under a GP prior. sigma2: 1.0

```
rng = np.random.default_rng(seed=12345)
Y = rng.multivariate_normal(zeros(Sigma.shape[0]), Sigma, size = 3, check_valid="raise")
plt.plot(X, Y.T)
plt.title("Realization of Three Random Functions under a GP prior.\n sigma2: {}".format(sigma2[0]))
plt.show()
```

Figure 6.12.: Realization of three random functions under a GP prior. sigma2: 1.0

6.15.5. Properties of the 1d Example

6.15.5.1. Several Bumps:

In this analysis, we observe several bumps in the x-range of $[0, 10]$. These bumps in the function occur because shorter distances exhibit high correlation, while longer distances tend to be essentially uncorrelated. This leads to variations in the function's behavior:

- When x and x' are one σ unit apart, the correlation is $\exp\left(-\sigma^2/(2\sigma^2)\right) = \exp(-1/2) \approx 0.61$, i.e., a relative high correlation.
- 2σ apart means correlation $\exp(-2^2/2) \approx 0.14$, i.e., only small correlation.
- 4σ apart means correlation $\exp(-4^2/2) \approx 0.0003$, i.e., nearly no correlation—variables are considered independent for almost all practical application.

6.15.5.2. Smoothness:

The function plotted in Figure 6.11 represents only a finite realization, which means that we have data for a limited number of pairs, specifically 100 points. These points

appear smooth in a tactile sense because they are closely spaced, and the plot function connects the dots with lines to create the appearance of smoothness. The complete surface, which can be conceptually extended to an infinite realization over a compact domain, is exceptionally smooth in a calculus sense due to the covariance function's property of being infinitely differentiable.

6.15.5.3. Scale of Two:

Regarding the scale of the Y values, they have a range of approximately $[-2, 2]$, with a 95% probability of falling within this range. In standard statistical terms, 95% of the data points typically fall within two standard deviations of the mean, which is a common measure of the spread or range of data.

```
import numpy as np
from numpy import array, zeros, power, ones, exp, multiply, eye, linspace, spacing, sqrt, arange, ap
from numpy.random import multivariate_normal

def build_Sigma(X, sigma2):
    n = X.shape[0]
    k = X.shape[1]
    D = zeros((k, n, n))
    for l in range(k):
        for i in range(n):
            for j in range(i, n):
                D[l, i, j] = 1/(2*sigma2[l])*(X[i,l] - X[j,l])**2
    D = sum(D)
    D = D + D.T
    return exp(-D)

def plot_mvn( a=0, b=10, sigma2=1.0, size=1, n=100, show=True):
    X = np.linspace(a, b, n, endpoint=False).reshape(-1,1)
    sigma2 = np.array([sigma2])
    Sigma = build_Sigma(X, sigma2)
    rng = np.random.default_rng(seed=12345)
    Y = rng.multivariate_normal(zeros(Sigma.shape[0]), Sigma, size = size, check_valid="raise")
    plt.plot(X, Y.T)
    plt.title("Realization of Random Functions under a GP prior.\n sigma2: {}".format(sigma2[0]))
    if show:
        plt.show()
```

```
plot_mvn(a=0, b=10, sigma2=10.0, size=3, n=250)
```

Figure 6.13.: Realization of Random Functions under a GP prior. sigma2: 10

```
plot_mvn(a=0, b=10, sigma2=0.1, size=3, n=250)
```

Figure 6.14.: Realization of Random Functions under a GP prior. sigma2: 0.1

6.16. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

7. Introduction to spotpython

Surrogate model based optimization methods are common approaches in simulation and optimization. SPOT was developed because there is a great need for sound statistical analysis of simulation and optimization algorithms. SPOT includes methods for tuning based on classical regression and analysis of variance techniques. It presents tree-based models such as classification and regression trees and random forests as well as Bayesian optimization (Gaussian process models, also known as Kriging). Combinations of different meta-modeling approaches are possible. SPOT comes with a sophisticated surrogate model based optimization method, that can handle discrete and continuous inputs. Furthermore, any model implemented in `scikit-learn` can be used out-of-the-box as a surrogate in `spotpython`.

SPOT implements key techniques such as exploratory fitness landscape analysis and sensitivity analysis. It can be used to understand the performance of various algorithms, while simultaneously giving insights into their algorithmic behavior.

The `spot` loop consists of the following steps:

1. Init: Build initial design X
2. Evaluate initial design on real objective f: $y = f(X)$
3. Build surrogate: $S = S(X, y)$
4. Optimize on surrogate: $X_0 = \text{optimize}(S)$
5. Evaluate on real objective: $y_0 = f(X_0)$
6. Impute (Infill) new points: $X = X \cup X_0$, $y = y \cup y_0$.
7. Goto 3.

7.1. Advantages of the spotpython approach

- Neural networks and many ML algorithms are non-deterministic, so results are noisy (i.e., depend on the the initialization of the weights). Enhanced noise handling strategies, OCBA (description from HPT-book).
- Optimal Computational Budget Allocation (OCBA) is a very efficient solution to solve the "general ranking and selection problem" if the objective function is noisy. It allocates function evaluations in an uneven manner to identify the best solutions and to reduce the total optimization costs. [Chen10a, Bart11b] Given a total number of optimization samples N to be allocated to k competing solutions whose performance is depicted by random variables with means $\bar{y}_i$ $(i = 1, 2, \ldots, k)$,

and finite variances σ_i^2, respectively, as $N \to \infty$, the Approximate Probability of Correct Selection (APCS) can be asymptotically maximized when

$$\frac{N_i}{N_j} = \left(\frac{\sigma_i/\delta_{b,i}}{\sigma_j/\delta_{b,j}}\right)^2, i, j \in \{1, 2, \ldots, k\}, \text{ and } i \neq j \neq b, \tag{7.1}$$

$$N_b = \sigma_b \sqrt{\sum_{i=1, i\neq b}^{k} \frac{N_i^2}{\sigma_i^2}}, \tag{7.2}$$

where N_i is the number of replications allocated to solution i, $\delta_{b,i} = \bar{y}_b - \bar{y}_i$, and $\bar{y}_b \leq \min_{i\neq b} \bar{y}_i$ Bartz-Beielstein and Friese (2011).

- Surrogate-based optimization: Better than grid search and random search (Reference to HPT-book)
- Visualization
- Importance based on the Kriging model
- Sensitivity analysis. Exploratory fitness landscape analysis. Provides XAI methods (feature importance, integrated gradients, etc.)
- Uncertainty quantification
- Flexible, modular meta-modeling handling. spotpython come with a Kriging model, which can be replaced by any model implemented in `scikit-learn`.
- Enhanced metric handling, especially for categorical hyperparameters (any sklearn metric can be used). Default is..
- Integration with TensorBoard: Visualization of the hyperparameter tuning process, of the training steps, the model graph. Parallel coordinates plot, scatter plot matrix, and more.
- Reproducibility. Results are stored as pickle files. The results can be loaded and visualized at any time and be transferred between different machines and operating systems.
- Handles scikit-learn models and pytorch models out-of-the-box. The user has to add a simple wrapper for passing the hyperparemeters to use a pytorch model in spotpython.
- Compatible with Lightning.
- User can add own models as plain python code.
- User can add own data sets in various formats.
- Flexible data handling and data preprocessing.
- Many examples online (hyperparameter-tuning-cookbook).

- spotpython uses a robust optimizer that can even deal with hyperparameter-settings that cause crashes of the algorithms to be tuned.
- even if the optimum is not found, HPT with spotpython prevents the user from choosing bad hyperparameters in a systematic way (design of experiments).

7.2. Disadvantages of the spotpython approach

- Time consuming
- Surrogate can be misguiding
- no parallelization implement yet

Central Idea: Evaluation of the surrogate model `S` is much cheaper (or / and much faster) than running the real-world experiment f. We start with a small example.

7.3. Example: `Spot` and the Sphere Function

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.init import fun_control_init, design_control_init
from spotpython.hyperparameters.values import set_control_key_value
from spotpython.spot import Spot
import matplotlib.pyplot as plt
```

7.3.1. The Objective Function: Sphere

The `spotpython` package provides several classes of objective functions. We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x) = x^2$$

```
fun = Analytical().fun_sphere
```

We can apply the function `fun` to input values and plot the result:

```
x = np.linspace(-1,1,100).reshape(-1,1)
y = fun(x)
plt.figure()
plt.plot(x, y, "k")
plt.show()
```

7.3.2. The `Spot` Method as an Optimization Algorithm Using a Surrogate Model

We initialize the `fun_control` dictionary. The `fun_control` dictionary contains the parameters for the objective function. The `fun_control` dictionary is passed to the `Spot` method.

```
fun_control=fun_control_init(lower = np.array([-1]),
                     upper = np.array([1]))
spot_0 = Spot(fun=fun,
                   fun_control=fun_control)
spot_0.run()
```

```
spotpython tuning: 4.960293502265715e-09 [#######---] 73.33%
spotpython tuning: 4.959666330154525e-09 [########--] 80.00%
spotpython tuning: 4.9571338392226926e-09 [#########-] 86.67%
```

```
spotpython tuning: 4.9571338392226926e-09 [#########-] 93.33%
spotpython tuning: 1.866838525968143e-10 [##########] 100.00% Done...

Experiment saved to 000_res.pkl

<spotpython.spot.spot.Spot at 0x107d27d10>
```

The method `print_results()` prints the results, i.e., the best objective function value ("min y") and the corresponding input value ("x0").

```
spot_0.print_results()
```

```
min y: 1.866838525968143e-10
x0: 1.3663229947447064e-05

[['x0', np.float64(1.3663229947447064e-05)]]
```

To plot the search progress, the method `plot_progress()` can be used. The parameter `log_y` is used to plot the objective function values on a logarithmic scale.

```
spot_0.plot_progress(log_y=True)
```

Figure 7.1.: Visualization of the search progress of the `Spot` method. The black elements (points and line) represent the initial design, before the surrogate is build. The red elements represent the search on the surrogate.

If the dimension of the input space is one, the method `plot_model()` can be used to visualize the model and the underlying objective function values.

```
spot_0.plot_model()
```

Figure 7.2.: Visualization of the model and the underlying objective function values.

7.4. Spot Parameters: fun_evals, init_size and show_models

We will modify three parameters:

1. The number of function evaluations (`fun_evals`) will be set to `10` (instead of 15, which is the default value) in the `fun_control` dictionary.
2. The parameter `show_models`, which visualizes the search process for each single iteration for 1-dim functions, in the `fun_control` dictionary.
3. The size of the initial design (`init_size`) in the `design_control` dictionary.

The full list of the `Spot` parameters is shown in code reference on GitHub, see Spot.

```
fun_control=fun_control_init(lower = np.array([-1]),
                             upper = np.array([1]),
```

```
                        fun_evals = 10,
                        show_models = True)
design_control = design_control_init(init_size=9)
spot_1 = Spot(fun=fun,
                     fun_control=fun_control,
                     design_control=design_control)
spot_1.run()
```

```
spotpython tuning: 9.632846333472212e-09 [##########] 100.00% Done...

Experiment saved to 000_res.pkl
```

7.5. Print the Results

```
spot_1.print_results()
```

```
min y: 9.632846333472212e-09
x0: -9.814706482351988e-05

[['x0', np.float64(-9.814706482351988e-05)]]
```

7.6. Show the Progress

```
spot_1.plot_progress()
```

7.7. Visualizing the Optimization and Hyperparameter Tuning Process with TensorBoard

`spotpython` supports the visualization of the hyperparameter tuning process with TensorBoard. The following example shows how to use TensorBoard with `spotpython`.

First, we define an "PREFIX" to identify the hyperparameter tuning process. The PREFIX is used to create a directory for the TensorBoard files.

```
fun_control = fun_control_init(
    PREFIX = "01",
    lower = np.array([-1]),
    upper = np.array([2]),
    fun_evals=100,
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True)
design_control = design_control_init(init_size=5)
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_02_12_22_09_44_0
Created spot_tensorboard_path: runs/spot_logs/01_maans08_2025-02-12_22-09-44 for SummaryWriter()
```

Since the `tensorboard_log` is `True`, `spotpython` will log the optimization process in the TensorBoard files. The argument `TENSORBOARD_CLEAN=True` will move the TensorBoard files from the previous run to a backup folder, so that TensorBoard files from previous runs are not overwritten and a clean start in the `runs` folder is guaranteed.

```
spot_tuner = Spot(fun=fun,
                  fun_control=fun_control,
                  design_control=design_control)
spot_tuner.run()
spot_tuner.print_results()
```

```
spotpython tuning: 2.487613964476317e-05 [#---------] 6.00%
spotpython tuning: 7.85064251023503e-07 [#---------] 7.00%
spotpython tuning: 7.210284620467494e-07 [#---------] 8.00%
spotpython tuning: 4.191441813872346e-07 [#---------] 9.00%
spotpython tuning: 7.4015092835768465e-09 [#---------] 10.00%
spotpython tuning: 7.4015092835768465e-09 [#---------] 11.00%
spotpython tuning: 7.4015092835768465e-09 [#---------] 12.00%
spotpython tuning: 7.4015092835768465e-09 [#---------] 13.00%
spotpython tuning: 7.4015092835768465e-09 [#---------] 14.00%
spotpython tuning: 7.4015092835768465e-09 [##--------] 15.00%
spotpython tuning: 7.4015092835768465e-09 [##--------] 16.00%
spotpython tuning: 7.4015092835768465e-09 [##--------] 17.00%
spotpython tuning: 7.4015092835768465e-09 [##--------] 18.00%
spotpython tuning: 7.4015092835768465e-09 [##--------] 19.00%
spotpython tuning: 7.4015092835768465e-09 [##--------] 20.00%
spotpython tuning: 6.704802934300911e-10 [##--------] 21.00%
spotpython tuning: 6.704802934300911e-10 [##--------] 22.00%
spotpython tuning: 6.668629799831143e-10 [##--------] 23.00%
spotpython tuning: 6.668629799831143e-10 [##--------] 24.00%
spotpython tuning: 6.499816856638441e-10 [##--------] 25.00%
spotpython tuning: 6.499816856638441e-10 [###-------] 26.00%
spotpython tuning: 6.499816856638441e-10 [###-------] 27.00%
spotpython tuning: 6.499816856638441e-10 [###-------] 28.00%
spotpython tuning: 6.499816856638441e-10 [###-------] 29.00%
spotpython tuning: 6.499816856638441e-10 [###-------] 30.00%
spotpython tuning: 6.499816856638441e-10 [###-------] 31.00%
spotpython tuning: 6.499816856638441e-10 [###-------] 32.00%
spotpython tuning: 6.499816856638441e-10 [###-------] 33.00%
spotpython tuning: 6.499816856638441e-10 [###-------] 34.00%
spotpython tuning: 6.499816856638441e-10 [####------] 35.00%
spotpython tuning: 6.499816856638441e-10 [####------] 36.00%
spotpython tuning: 6.499816856638441e-10 [####------] 37.00%
spotpython tuning: 6.499816856638441e-10 [####------] 38.00%
spotpython tuning: 6.499816856638441e-10 [####------] 39.00%
spotpython tuning: 6.499816856638441e-10 [####------] 40.00%
spotpython tuning: 6.499816856638441e-10 [####------] 41.00%
spotpython tuning: 6.499816856638441e-10 [####------] 42.00%
spotpython tuning: 6.499816856638441e-10 [####------] 43.00%
```

```
spotpython tuning: 6.499816856638441e-10 [####------] 44.00%
spotpython tuning: 6.499816856638441e-10 [####------] 45.00%
spotpython tuning: 6.499816856638441e-10 [#####-----] 46.00%
spotpython tuning: 6.499816856638441e-10 [#####-----] 47.00%
spotpython tuning: 6.499816856638441e-10 [#####-----] 48.00%
spotpython tuning: 6.499816856638441e-10 [#####-----] 49.00%
spotpython tuning: 6.499816856638441e-10 [#####-----] 50.00%
spotpython tuning: 6.499816856638441e-10 [#####-----] 51.00%
spotpython tuning: 6.499816856638441e-10 [#####-----] 52.00%
spotpython tuning: 6.499816856638441e-10 [#####-----] 53.00%
spotpython tuning: 6.499816856638441e-10 [#####-----] 54.00%
spotpython tuning: 6.499816856638441e-10 [######----] 55.00%
spotpython tuning: 6.499816856638441e-10 [######----] 56.00%
spotpython tuning: 6.499816856638441e-10 [######----] 57.00%
spotpython tuning: 6.499816856638441e-10 [######----] 58.00%
spotpython tuning: 6.499816856638441e-10 [######----] 59.00%
spotpython tuning: 6.499816856638441e-10 [######----] 60.00%
spotpython tuning: 6.499816856638441e-10 [######----] 61.00%
spotpython tuning: 6.499816856638441e-10 [######----] 62.00%
spotpython tuning: 6.499816856638441e-10 [######----] 63.00%
spotpython tuning: 6.499816856638441e-10 [######----] 64.00%
spotpython tuning: 6.499816856638441e-10 [######----] 65.00%
spotpython tuning: 6.499816856638441e-10 [#######---] 66.00%
spotpython tuning: 6.499816856638441e-10 [#######---] 67.00%
spotpython tuning: 6.499816856638441e-10 [#######---] 68.00%
spotpython tuning: 6.499816856638441e-10 [#######---] 69.00%
spotpython tuning: 6.499816856638441e-10 [#######---] 70.00%
spotpython tuning: 6.499816856638441e-10 [#######---] 71.00%
spotpython tuning: 6.499816856638441e-10 [#######---] 72.00%
spotpython tuning: 6.499816856638441e-10 [#######---] 73.00%
spotpython tuning: 6.499816856638441e-10 [#######---] 74.00%
spotpython tuning: 6.499816856638441e-10 [########--] 75.00%
spotpython tuning: 6.499816856638441e-10 [########--] 76.00%
spotpython tuning: 6.499816856638441e-10 [########--] 77.00%
spotpython tuning: 6.499816856638441e-10 [########--] 78.00%
spotpython tuning: 6.499816856638441e-10 [########--] 79.00%
spotpython tuning: 6.499816856638441e-10 [########--] 80.00%
spotpython tuning: 6.499816856638441e-10 [########--] 81.00%
spotpython tuning: 6.499816856638441e-10 [########--] 82.00%
spotpython tuning: 6.499816856638441e-10 [########--] 83.00%
spotpython tuning: 6.499816856638441e-10 [########--] 84.00%
spotpython tuning: 6.499816856638441e-10 [########--] 85.00%
spotpython tuning: 6.499816856638441e-10 [#########-] 86.00%
spotpython tuning: 6.499816856638441e-10 [#########-] 87.00%
spotpython tuning: 6.499816856638441e-10 [#########-] 88.00%
```

```
spotpython tuning: 6.499816856638441e-10 [#########-] 89.00%
spotpython tuning: 6.499816856638441e-10 [#########-] 90.00%
spotpython tuning: 6.499816856638441e-10 [#########-] 91.00%
spotpython tuning: 6.499816856638441e-10 [#########-] 92.00%
spotpython tuning: 6.499816856638441e-10 [#########-] 93.00%
spotpython tuning: 4.991706867647581e-10 [#########-] 94.00%
spotpython tuning: 4.991706867647581e-10 [##########] 95.00%
spotpython tuning: 4.991706867647581e-10 [##########] 96.00%
spotpython tuning: 4.991706867647581e-10 [##########] 97.00%
spotpython tuning: 4.991706867647581e-10 [##########] 98.00%
spotpython tuning: 4.991706867647581e-10 [##########] 99.00%
spotpython tuning: 4.3763009967401825e-10 [##########] 100.00% Done...

Experiment saved to 01_res.pkl
min y: 4.3763009967401825e-10
x0: 2.0919610409231293e-05
```

```
[['x0', np.float64(2.0919610409231293e-05)]]
```

Now we can start TensorBoard in the background. The TensorBoard process will read the TensorBoard files and visualize the hyperparameter tuning process. From the terminal, we can start TensorBoard with the following command:

```
tensorboard --logdir="./runs"
```

`logdir` is the directory where the TensorBoard files are stored. In our case, the TensorBoard files are stored in the directory `./runs`.

TensorBoard will start a web server on port 6006. We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

The first TensorBoard visualization shows the objective function values plotted against the wall time. The wall time is the time that has passed since the start of the hyperparameter tuning process. The five initial design points are shown in the upper left region of the plot. The line visualizes the optimization process.

The second TensorBoard visualization shows the input values, i.e., x_0, plotted against the wall time.

The third TensorBoard plot illustrates how `spotpython` can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important parameter, the learning rate θ of the Kriging surrogate is plotted against the number of optimization steps.

Figure 7.3.: TensorBoard visualization of the spotpython process.

7.8. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

8. Multi-dimensional Functions

This chapter illustrates how high-dimensional functions can be optimized and analyzed. For reasons of illustration, we will use the three-dimensional Sphere function, which is a simple and well-known function. The problem dimension is $k = 3$, but can be easily adapted to other, higher dimensions.

```
import numpy as np
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.init import fun_control_init, surrogate_control_init, design_control_init
from spotpython.spot import Spot
```

8.1. The Objective Function: 3-dim Sphere

The `spotpython` package provides several classes of objective functions. We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x) = \sum_{i}^{k} x_i^2.$$

The Sphere function is continuous, convex and unimodal. The plot shows its two-dimensional form. The global minimum is

$$f(x) = 0, \text{at } x = (0, 0, \dots, 0).$$

It is available as `fun_sphere` in the `Analytical` class [SOURCE].

```
fun = Analytical().fun_sphere
```

Here we will use problem dimension $k = 3$, which can be specified by the `lower` bound arrays. The size of the `lower` bound array determines the problem dimension. If we select `-1.0 * np.ones(3)`, a three-dimensional function is created.

In contrast to the one-dimensional case (Section 7.7), where only one `theta` value was used, we will use three different `theta` values (one for each dimension), i.e., we set `n_theta=3` in the `surrogate_control`. As default, `spotpython` sets the `n_theta` to

the problem dimension. Therefore, the `n_theta` parameter can be omitted in this case. More specifically, if `n_theta` is larger than 1 or set to the string "anisotropic", then the k theta values are used, where k is the problem dimension. The meaning of "anisotropic" is explained in @#sec-iso-aniso-kriging.

The prefix is set to `"03"` to distinguish the results from the one-dimensional case. Again, TensorBoard can be used to monitor the progress of the optimization.

We can also add interpretable labels to the dimensions, which will be used in the plots. Therefore, we set `var_name=["Pressure", "Temp", "Lambda"]` instead of the default `var_name=None`, which would result in the labels `x_0`, `x_1`, and `x_2`.

```
fun_control = fun_control_init(
              PREFIX="03",
              lower = -1.0*np.ones(3),
              upper = np.ones(3),
              var_name=["Pressure", "Temp", "Lambda"],
              TENSORBOARD_CLEAN=True,
              tensorboard_log=True)
surrogate_control = surrogate_control_init(n_theta=3)
spot_3 = Spot(fun=fun,
                  fun_control=fun_control,
                  surrogate_control=surrogate_control)
spot_3.run()
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_02_12_22_10
Created spot_tensorboard_path: runs/spot_logs/03_maans08_2025-02-12_22-10-10 for Summa
spotpython tuning: 0.03443518849425384 [#######---] 73.33%
spotpython tuning: 0.031343410270600766 [########--] 80.00%
spotpython tuning: 0.0009628776719739201 [#########-] 86.67%
spotpython tuning: 8.551395067781694e-05 [#########-] 93.33%
spotpython tuning: 6.646694576732236e-05 [##########] 100.00% Done...

Experiment saved to 03_res.pkl


<spotpython.spot.spot.Spot at 0x10362bd10>
```

> **Note**
>
> Now we can start TensorBoard in the background with the following command:
>
> ```
> tensorboard --logdir="./runs"
> ```
>
> and can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

8.1.1. Results

8.1.1.1. Best Objective Function Values

The best objective function value and its corresponding input values are printed as follows:

```
_ = spot_3.print_results()
```

```
min y: 6.646694576732236e-05
Pressure: 0.005351119956860987
Temp: 0.001959694434893034
Lambda: 0.005830270893916998
```

The method `plot_progress()` plots current and best found solutions versus the number of iterations as shown in Figure 8.1.

```
spot_3.plot_progress()
```

Figure 8.1.: Progress of the optimization process for the 3-dim Sphere function. The initial design points are shown in black, whereas the points that were found by the search on the surrogate are plotted in red.

8.1.1.2. A Contour Plot

We can select two dimensions, say $i = 0$ and $j = 1$, and generate a contour plot as follows. Note, we have specified identical `min_z` and `max_z` values to generate comparable plots.

```
spot_3.plot_contour(i=0, j=1, min_z=0, max_z=2.25)
```

- In a similar manner, we can plot dimension $i = 0$ and $j = 2$:

```
spot_3.plot_contour(i=0, j=2, min_z=0, max_z=2.25)
```

- The final combination is $i = 1$ and $j = 2$:

```
spot_3.plot_contour(i=1, j=2, min_z=0, max_z=2.25)
```

- The three plots look very similar, because the `fun_sphere` is symmetric.
- This can also be seen from the variable importance:

```
_ = spot_3.print_importance()
```

```
Pressure:  95.21437451356887
Temp:  100.0
Lambda:  87.10302600165961
```

```
spot_3.plot_importance()
```

8.1.2. TensorBoard

Figure 8.2.: TensorBoard visualization of the spotpython process. Objective function values plotted against wall time.

The second TensorBoard visualization shows the input values, i.e., $x_0, \ldots, x_2$, plotted against the wall time.

The third TensorBoard plot illustrates how `spotpython` can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important parameter, the learning rate θ of the Kriging surrogate is plotted against the number of optimization steps.

Figure 8.3.: TensorBoard visualization of the spotpython surrogate model.

8.1.3. Conclusion

Based on this quick analysis, we can conclude that all three dimensions are equally important (as expected, because the Analytical function is known).

8.2. Exercises

Exercise 8.1 (The Three Dimensional `fun_cubed`)**.** The `spotpython` package provides several classes of objective functions.

We will use the `fun_cubed` in the `Analytical` class [SOURCE]. The input dimension is `3`. The search range is $-1 \leq x \leq 1$ for all dimensions.

Tasks: * Generate contour plots * Calculate the variable importance. * Discuss the variable importance: * Are all variables equally important? * If not: * Which is the most important variable? * Which is the least important variable?

Exercise 8.2 (The Ten Dimensional `fun_wing_wt`)**.**

- The input dimension is `10`. The search range is $0 \leq x \leq 1$ for all dimensions.
- Calculate the variable importance.
- Discuss the variable importance:
  - Are all variables equally important?
  - If not:
    - Which is the most important variable?
    - Which is the least important variable?

  - Generate contour plots for the three most important variables. Do they confirm your selection?

Exercise 8.3 (The Three Dimensional `fun_runge`).

- The input dimension is `3`. The search range is $-5 \leq x \leq 5$ for all dimensions.
- Generate contour plots
- Calculate the variable importance.
- Discuss the variable importance:
  - Are all variables equally important?
  - If not:
    * Which is the most important variable?
    * Which is the least important variable?

Exercise 8.4 (The Three Dimensional `fun_linear`).

- The input dimension is `3`. The search range is $-5 \leq x \leq 5$ for all dimensions.
- Generate contour plots
- Calculate the variable importance.
- Discuss the variable importance:
  - Are all variables equally important?
  - If not:
    * Which is the most important variable?
    * Which is the least important variable?

Exercise 8.5 (The Two Dimensional Rosenbrock Function `fun_rosen`).

- The input dimension is `2`. The search range is $-5 \leq x \leq 10$ for all dimensions.
- See Rosenbrock function and Rosenbrock Function for details.
- Generate contour plots
- Calculate the variable importance.
- Discuss the variable importance:
  - Are all variables equally important?
  - If not:
    * Which is the most important variable?
    * Which is the least important variable?

8.3. Selected Solutions

Solution 8.1 (Solution to Exercise 8.1: The Three-dimensional Cubed Function `fun_cubed`). We instanciate the `fun_cubed` function from the `Analytical` class.

```
from spotpython.fun.objectivefunctions import Analytical
fun_cubed = Analytical().fun_cubed
```

- Here we will use problem dimension $k = 3$, which can be specified by the `lower` bound arrays. The size of the `lower` bound array determines the problem dimension. If we select `-1.0 * np.ones(3)`, a three-dimensional function is created.
- In contrast to the one-dimensional case, where only one `theta` value was used, we will use three different `theta` values (one for each dimension), i.e., we can set `n_theta=3` in the `surrogate_control`. However, this is not necessary, because by default, `n_theta` is set to the number of dimensions.
- The prefix is set to `"03"` to distinguish the results from the one-dimensional case.
- We will set the `fun_evals=20` to limit the number of function evaluations to 20 for this example.
- The size of the initial design is set to `10` by default. It can be changed by setting `init_size=10` via `design_control_init` in the `design_control` dictionary.
- Again, TensorBoard can be used to monitor the progress of the optimization.
- We can also add interpretable labels to the dimensions, which will be used in the plots. Therefore, we set `var_name=["Pressure", "Temp", "Lambda"]` instead of the default `var_name=None`, which would result in the labels `x_0`, `x_1`, and `x_2`.

Here is the link to the documentation of the fun_control_init function: [DOC]. The documentation of the `design_control_init` function can be found here: [DOC].

The setup can be done as follows:

```
fun_control = fun_control_init(
              PREFIX="cubed",
              fun_evals=20,
              lower = -1.0*np.ones(3),
              upper = np.ones(3),
              var_name=["Pressure", "Temp", "Lambda"],
              TENSORBOARD_CLEAN=True,
              tensorboard_log=True
              )

surrogate_control = surrogate_control_init(n_theta=3)
design_control = design_control_init(init_size=10)
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_02_12_22_10_12_0
Created spot_tensorboard_path: runs/spot_logs/cubed_maans08_2025-02-12_22-10-12 for SummaryWriter()
```

- After the setup, we can pass the dictionaries to the `Spot` class and run the optimization process.

```
spot_cubed = Spot(fun=fun_cubed,
                  fun_control=fun_control,
                  surrogate_control=surrogate_control)
spot_cubed.run()
```

```
spotpython tuning: -1.4616833740603914 [######----] 55.00%
spotpython tuning: -1.4616833740603914 [######----] 60.00%
spotpython tuning: -2.0535965272858117 [######----] 65.00%
spotpython tuning: -2.0535965272858117 [#######---] 70.00%
spotpython tuning: -2.0535965272858117 [########--] 75.00%
spotpython tuning: -2.0535965272858117 [########--] 80.00%
spotpython tuning: -2.0535965272858117 [########--] 85.00%
spotpython tuning: -2.0906487259161515 [#########-] 90.00%
spotpython tuning: -2.0906487259161515 [##########] 95.00%
spotpython tuning: -3.0 [##########] 100.00% Done...

Experiment saved to cubed_res.pkl

<spotpython.spot.spot.Spot at 0x15c17aa50>
```

- Results

```
_ = spot_cubed.print_results()
```

```
min y: -3.0
Pressure: -1.0
Temp: -1.0
Lambda: -1.0
```

```
spot_cubed.plot_progress()
```

- Contour Plots

We can select two dimensions, say $i = 0$ and $j = 1$, and generate a contour plot as follows.

We can specify identical `min_z` and `max_z` values to generate comparable plots. The default values are `min_z=None` and `max_z=None`, which will be replaced by the minimum and maximum values of the objective function.

```
min_z = -3
max_z = 1
spot_cubed.plot_contour(i=0, j=1, min_z=min_z, max_z=max_z)
```

- In a similar manner, we can plot dimension $i = 0$ and $j = 2$:

```
spot_cubed.plot_contour(i=0, j=2, min_z=min_z, max_z=max_z)
```

- The final combination is $i = 1$ and $j = 2$:

```
spot_cubed.plot_contour(i=1, j=2, min_z=min_z, max_z=max_z)
```

- The variable importance can be printed and visualized as follows:

```
_ = spot_cubed.print_importance()
```

```
Pressure:  100.0
Temp:  83.62271900424608
Lambda:  97.98572230390421
```

```
spot_cubed.plot_importance()
```

Solution 8.2 (Solution to Exercise 8.5: The Two-dimensional Rosenbrock Function `fun_rosen`).

```
import numpy as np
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.init import fun_control_init, surrogate_control_init
from spotpython.spot import Spot
```

- The Objective Function: 2-dim `fun_rosen`

The `spotpython` package provides several classes of objective functions. We will use the `fun_rosen` in the `Analytical` class [SOURCE].

```
fun_rosen = Analytical().fun_rosen
```

- Here we will use problem dimension $k = 2$, which can be specified by the `lower` bound arrays.
- The size of the `lower` bound array determines the problem dimension. If we select `-5.0 * np.ones(2)`, a two-dimensional function is created.
- In contrast to the one-dimensional case, where only one `theta` value is used, we will use k different `theta` values (one for each dimension), i.e., we set `n_theta=3` in the `surrogate_control`.
- The prefix is set to `"ROSEN"`.
- Again, TensorBoard can be used to monitor the progress of the optimization.

```
fun_control = fun_control_init(
              PREFIX="ROSEN",
              lower = -5.0*np.ones(2),
              upper = 10*np.ones(2),
              fun_evals=25)
surrogate_control = surrogate_control_init(n_theta=2)
spot_rosen = Spot(fun=fun_rosen,
                  fun_control=fun_control,
                  surrogate_control=surrogate_control)
spot_rosen.run()
```

```
spotpython tuning: 90.78737194937058 [####------] 44.00%
spotpython tuning: 1.0172240416576994 [#####-----] 48.00%
spotpython tuning: 1.0172240416576994 [#####-----] 52.00%
spotpython tuning: 1.0172240416576994 [######----] 56.00%
spotpython tuning: 1.0172240416576994 [######----] 60.00%
spotpython tuning: 1.0172240416576994 [######----] 64.00%
spotpython tuning: 1.0172240416576994 [#######---] 68.00%
spotpython tuning: 1.0172240416576994 [#######---] 72.00%
spotpython tuning: 1.0172240416576994 [########--] 76.00%
spotpython tuning: 1.0172240416576994 [########--] 80.00%
spotpython tuning: 0.9204178748278081 [########--] 84.00%
```

```
spotpython tuning: 0.9204178748278081 [#########-] 88.00%
spotpython tuning: 0.9204178748278081 [#########-] 92.00%
spotpython tuning: 0.9204178748278081 [##########] 96.00%
spotpython tuning: 0.7259403018002056 [##########] 100.00% Done...

Experiment saved to ROSEN_res.pkl


<spotpython.spot.spot.Spot at 0x15c1ca7e0>
```

Now we can start TensorBoard in the background with the following command:

```
tensorboard --logdir="./runs"
```

and can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

- Results

```
_ = spot_rosen.print_results()
```

```
min y: 0.7259403018002056
x0: 0.16580025802317414
x1: 0.08230860213240618
```

```
spot_rosen.plot_progress(log_y=True)
```

- A Contour Plot: We can select two dimensions, say $i = 0$ and $j = 1$, and generate a contour plot as follows.

- Note: For higher dimensions, it might be useful to have identical `min_z` and `max_z` values to generate comparable plots. The default values are `min_z=None` and `max_z=None`, which will be replaced by the minimum and maximum values of the objective function.

```
min_z = None
max_z = None
spot_rosen.plot_contour(i=0, j=1, min_z=min_z, max_z=max_z)
```

- The variable importance can be calculated as follows:

```
_ = spot_rosen.print_importance()
```

```
x0:  99.99999999999999
x1:  1.2430550048669098
```

```
spot_rosen.plot_importance()
```

8.4. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

9. Isotropic and Anisotropic Kriging

This chapter illustrates the difference between isotropic and anisotropic Kriging models. The difference is illustrated with the help of the `spotpython` package. Isotropic Kriging models use the same `theta` value for every dimension. Anisotropic Kriging models use different `theta` values for each dimension.

9.1. Example: Isotropic `Spot` Surrogate and the 2-dim Sphere Function

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.utils.init import fun_control_init, surrogate_control_init
PREFIX="003"
```

9.1.1. The Objective Function: 2-dim Sphere

The `spotpython` package provides several classes of objective functions. We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x, y) = x^2 + y^2$$

The size of the `lower` bound vector determines the problem dimension. Here we will use `np.array([-1, -1])`, i.e., a two-dimensional function.

```
fun = Analytical().fun_sphere
fun_control = fun_control_init(PREFIX=PREFIX,
                               lower = np.array([-1, -1]),
                               upper = np.array([1, 1]))
```

Although the default `spot` surrogate model is an isotropic Kriging model, we will explicitly set the `n_theta` parameter to a value of `1`, so that the same theta value is used for both dimensions. This is done to illustrate the difference between isotropic and anisotropic Kriging models.

```
surrogate_control=surrogate_control_init(n_theta=1)
```

```
spot_2 = Spot(fun=fun,
                   fun_control=fun_control,
                   surrogate_control=surrogate_control)

spot_2.run()
```

```
spotpython tuning: 1.9577055446455904e-05 [#######---] 73.33%
spotpython tuning: 1.9577055446455904e-05 [########--] 80.00%
spotpython tuning: 1.9577055446455904e-05 [#########-] 86.67%
spotpython tuning: 1.9577055446455904e-05 [#########-] 93.33%
spotpython tuning: 1.9577055446455904e-05 [##########] 100.00% Done...

Experiment saved to 003_res.pkl


<spotpython.spot.spot.Spot at 0x154ec3c80>
```

9.1.2. Results

```
spot_2.print_results()
```

```
min y: 1.9577055446455904e-05
x0: 0.00171841680372826
x1: 0.0040772661349389805


[['x0', np.float64(0.00171841680372826)],
 ['x1', np.float64(0.0040772661349389805)]]
```

```
spot_2.plot_progress(log_y=True)
```

9.2. Example With Anisotropic Kriging

As described in Section 9.1, the default parameter setting of `spotpython`'s Kriging surrogate uses the same `theta` value for every dimension. This is referred to as "using an isotropic kernel". If different `theta` values are used for each dimension, then an anisotropic kernel is used. To enable anisotropic models in `spotpython`, the number of `theta` values should be larger than one. We can use `surrogate_control=surrogate_control_init(n_theta=2)` to enable this behavior (2 is the problem dimension).

```
surrogate_control = surrogate_control_init(n_theta=2)
spot_2_anisotropic = Spot(fun=fun,
                    fun_control=fun_control,
                    surrogate_control=surrogate_control)
spot_2_anisotropic.run()
```

```
spotpython tuning: 1.5904060546935205e-05 [#######---] 73.33%
spotpython tuning: 1.5904060546935205e-05 [########--] 80.00%
spotpython tuning: 1.5904060546935205e-05 [#########-] 86.67%
spotpython tuning: 1.5904060546935205e-05 [#########-] 93.33%
spotpython tuning: 1.2084513018724136e-05 [##########] 100.00% Done...

Experiment saved to 003_res.pkl


<spotpython.spot.spot.Spot at 0x156455e20>
```

The search progress of the optimization with the anisotropic model can be visualized:

```
spot_2_anisotropic.plot_progress(log_y=True)
```

```
spot_2_anisotropic.print_results()
```

```
min y: 1.2084513018724136e-05
x0: -0.003294464459178357
x1: -0.0011095120305498227
```

```
[['x0', np.float64(-0.003294464459178357)],
 ['x1', np.float64(-0.0011095120305498227)]]
```

```
spot_2_anisotropic.surrogate.plot()
```

9.2.1. Taking a Look at the `theta` Values

9.2.1.1. `theta` Values from the `spot` Model

We can check, whether one or several `theta` values were used. The `theta` values from the surrogate can be printed as follows:

```
spot_2_anisotropic.surrogate.theta
```

```
[np.float64(-0.3285891031238254), np.float64(-0.13977864690749586)]
```

- Since the surrogate from the isotropic setting was stored as `spot_2`, we can also take a look at the `theta` value from this model:

```
spot_2.surrogate.theta
```

```
[np.float64(-0.1652222808345842)]
```

9.2.1.2. TensorBoard

Now we can start TensorBoard in the background with the following command:

```
tensorboard --logdir="./runs"
```

We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

The TensorBoard plot illustrates how `spotpython` can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important parameter, the learning rate θ of the Kriging surrogate is plotted against the number of optimization steps.

Figure 9.1.: TensorBoard visualization of the spotpython surrogate model.

9.3. Exercises

9.3.1. 1. The Branin Function `fun_branin`

- Describe the function.
  - The input dimension is 2. The search range is $-5 \leq x_1 \leq 10$ and $0 \leq x_2 \leq 15$.
- Compare the results from `spotpython` run a) with isotropic and b) anisotropic surrogate models.

- Modify the termination criterion: instead of the number of evaluations (which is specified via `fun_evals`), the time should be used as the termination criterion. This can be done as follows (`max_time=1` specifies a run time of one minute):

```
from math import inf
fun_control = fun_control_init(
              fun_evals=inf,
              max_time=1)
```

9.3.2. 2. The Two-dimensional Sin-Cos Function `fun_sin_cos`

- Describe the function.
  - The input dimension is 2. The search range is $-2\pi \leq x_1 \leq 2\pi$ and $-2\pi \leq x_2 \leq 2\pi$.
- Compare the results from `spotpython` run a) with isotropic and b) anisotropic surrogate models.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

9.3.3. 3. The Two-dimensional Runge Function `fun_runge`

- Describe the function.
  - The input dimension is 2. The search range is $-5 \leq x_1 \leq 5$ and $-5 \leq x_2 \leq 5$.
- Compare the results from `spotpython` run a) with isotropic and b) anisotropic surrogate models.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

9.3.4. 4. The Ten-dimensional Wing-Weight Function `fun_wingwt`

- Describe the function.
  - The input dimension is 10. The search ranges are between 0 and 1 (values are mapped internally to their natural bounds).
- Compare the results from `spotpython` run a) with isotropic and b) anisotropic surrogate models.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

9.3.5. 5. The Two-dimensional Rosenbrock Function `fun_rosen`

- Describe the function.
  - The input dimension is 2. The search ranges are between -5 and 10.
- Compare the results from `spotpython` run a) with isotropic and b) anisotropic surrogate models.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

9.4. Selected Solutions

9.4.1. Solution to Exercise Section 9.3.5: The Two-dimensional Rosenbrock Function `fun_rosen`

9.4.1.1. The Two Dimensional `fun_rosen`: The Isotropic Case

```
import numpy as np
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.init import fun_control_init, surrogate_control_init
from spotpython.spot import Spot
```

The `spotpython` package provides several classes of objective functions. We will use the `fun_rosen` in the `analytical` class [SOURCE].

```
fun_rosen = Analytical().fun_rosen
```

Here we will use problem dimension $k = 2$, which can be specified by the `lower` bound arrays. The size of the `lower` bound array determines the problem dimension.

The prefix is set to `"ROSEN"` to distinguish the results from the one-dimensional case. Again, TensorBoard can be used to monitor the progress of the optimization.

```
fun_control = fun_control_init(
              PREFIX="ROSEN",
              lower = np.array([-5, -5]),
              upper = np.array([10, 10]),
              show_progress=True)
surrogate_control = surrogate_control_init(n_theta=1)
spot_rosen = Spot(fun=fun_rosen,
                  fun_control=fun_control,
```

```
                  surrogate_control=surrogate_control)
spot_rosen.run()
```

```
spotpython tuning: 52.87634888275474 [#######---] 73.33%
spotpython tuning: 52.36206921045742 [########--] 80.00%
spotpython tuning: 52.36206921045742 [#########-] 86.67%
spotpython tuning: 43.44263277019559 [#########-] 93.33%
spotpython tuning: 12.275794686387082 [##########] 100.00% Done...

Experiment saved to ROSEN_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x163e59fd0>
```

> **Note**
>
> Now we can start TensorBoard in the background with the following command:
>
> ```
> tensorboard --logdir="./runs"
> ```
>
> and can access the TensorBoard web server with the following URL:
>
> ```
> http://localhost:6006/
> ```

9.4.1.1.1. Results

```
_ = spot_rosen.print_results()
```

```
min y: 12.275794686387082
x0: -2.3708433337788763
x1: 5.923091744213865
```

```
spot_rosen.plot_progress()
```

9.4.1.1.2. A Contour Plot

We can select two dimensions, say $i = 0$ and $j = 1$, and generate a contour plot as follows.

```
min_z = None
max_z = None
spot_rosen.plot_contour(i=0, j=1, min_z=min_z, max_z=max_z)
```

- The variable importance cannot be calculated, because only one `theta` value was used.

9.4.1.1.3. TensorBoard

TBD

9.4.1.2. The Two Dimensional `fun_rosen`: The Anisotropic Case

```
import numpy as np
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.init import fun_control_init, surrogate_control_init
from spotpython.spot import Spot
```

The `spotpython` package provides several classes of objective functions. We will use the `fun_rosen` in the `analytical` class [SOURCE].

```
fun_rosen = Analytical().fun_rosen
```

Here we will use problem dimension $k = 2$, which can be specified by the `lower` bound arrays. The size of the `lower` bound array determines the problem dimension.

We can also add interpreable labels to the dimensions, which will be used in the plots.

```
fun_control = fun_control_init(
              PREFIX="ROSEN",
              lower = np.array([-5, -5]),
              upper = np.array([10, 10]),
              show_progress=True)
surrogate_control = surrogate_control_init(n_theta=2)
spot_rosen = Spot(fun=fun_rosen,
                  fun_control=fun_control,
                  surrogate_control=surrogate_control)
spot_rosen.run()
```

```
spotpython tuning: 90.78737194937058 [#######---] 73.33%
spotpython tuning: 1.0172240416576994 [########--] 80.00%
spotpython tuning: 1.0172240416576994 [#########-] 86.67%
spotpython tuning: 1.0172240416576994 [#########-] 93.33%
spotpython tuning: 1.0172240416576994 [##########] 100.00% Done...

Experiment saved to ROSEN_res.pkl

<spotpython.spot.spot.Spot at 0x163f05370>
```

> **Note**
>
> Now we can start TensorBoard in the background with the following command:

```
tensorboard --logdir="./runs"
```

and can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

9.4.1.2.1. Results

```
_ = spot_rosen.print_results()
```

```
min y: 1.0172240416576994
x0: 0.00278369658649557
x1: -0.04772451042254187
```

```
spot_rosen.plot_progress()
```

9.4.1.2.2. A Contour Plot

We can select two dimensions, say $i = 0$ and $j = 1$, and generate a contour plot as follows.

```
min_z = None
max_z = None
spot_rosen.plot_contour(i=0, j=1, min_z=min_z, max_z=max_z)
```

- The variable importance can be calculated as follows:

```
_ = spot_rosen.print_importance()
```

```
x0:  100.00000000000001
x1:  2.227560396746198
```

```
spot_rosen.plot_importance()
```

9.4.1.2.3. TensorBoard

TBD

9.5. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

10. Using sklearn Surrogates in spotpython

Besides the internal kriging surrogate, which is used as a default by spotpython, any surrogate model from scikit-learn can be used as a surrogate in spotpython. This chapter explains how to use scikit-learn surrogates in spotpython.

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
```

10.1. Example: Branin Function with spotpython's Internal Kriging Surrogate

10.1.1. The Objective Function Branin

- The spotpython package provides several classes of objective functions.
- We will use an analytical objective function, i.e., a function that can be described by a (closed) formula.
- Here we will use the Branin function:

  ```
  y = a * (x2 - b * x1**2 + c * x1 - r) ** 2 + s * (1 - t) * np.cos(x1) + s,
  where values of a, b, c, r, s and t are: a = 1, b = 5.1 / (4*pi**2),
  c = 5 / pi, r = 6, s = 10 and t = 1 / (8*pi).
  ```

- It has three global minima:

  ```
  f(x) = 0.397887 at (-pi, 12.275), (pi, 2.275), and (9.42478, 2.475).
  ```

```
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical().fun_branin
```

TensorBoard

Similar to the one-dimensional case, which was introduced in Section Section 7.7, we can use TensorBoard to monitor the progress of the optimization. We will use the same code, only the prefix is different:

```
from spotpython.utils.init import fun_control_init, design_control_init
PREFIX = "04"
fun_control = fun_control_init(
    PREFIX=PREFIX,
    lower = np.array([-5,-0]),
    upper = np.array([10,15]),
    fun_evals=20,
    max_time=inf)

design_control = design_control_init(
    init_size=10)
```

10.1.2. Running the surrogate model based optimizer Spot:

```
spot_2 = Spot(fun=fun,
                   fun_control=fun_control,
                   design_control=design_control)
```

```
spot_2.run()
```

```
spotpython tuning: 3.8004662117718677 [######----] 55.00%
spotpython tuning: 3.8004662117718677 [######----] 60.00%
spotpython tuning: 3.159024883515257 [######----] 65.00%
spotpython tuning: 3.133916697143885 [#######---] 70.00%
spotpython tuning: 2.8926749183116236 [########--] 75.00%
spotpython tuning: 0.4190219407803557 [########--] 80.00%
spotpython tuning: 0.401871440801683 [########--] 85.00%
spotpython tuning: 0.39926034519166187 [#########-] 90.00%
spotpython tuning: 0.39926034519166187 [##########] 95.00%
spotpython tuning: 0.39926034519166187 [##########] 100.00% Done...

Experiment saved to 04_res.pkl


<spotpython.spot.spot.Spot at 0x152ed6e10>
```

10.1.3. TensorBoard

Now we can start TensorBoard in the background with the following command:

```
tensorboard --logdir="./runs"
```

We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

The TensorBoard plot illustrates how spotpython can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important parameter, the learning rate θ of the Kriging surrogate is plotted against the number of optimization steps.

Figure 10.1.: TensorBoard visualization of the spotpython optimization process and the surrogate model.

10.1.4. Print the Results

```
spot_2.print_results()
```

```
min y: 0.39926034519166187
x0: 3.1509546500431656
x1: 2.298567899278217
```

```
[['x0', np.float64(3.1509546500431656)], ['x1', np.float64(2.298567899278217)]]
```

10.1.5. Show the Progress and the Surrogate

```
spot_2.plot_progress(log_y=True)
```

```
spot_2.surrogate.plot()
```

10.2. Example: Using Surrogates From scikit-learn

- Default is the `spotpython` (i.e., the internal) `kriging` surrogate.
- It can be called explicitely and passed to `Spot`.

```
from spotpython.build.kriging import Kriging
S_0 = Kriging(name='kriging', seed=123)
```

- Alternatively, models from `scikit-learn` can be selected, e.g., Gaussian Process, RBFs, Regression Trees, etc.

```
# Needed for the sklearn surrogates:
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn import linear_model
from sklearn import tree
import pandas as pd
```

- Here are some additional models that might be useful later:

```
S_Tree = DecisionTreeRegressor(random_state=0)
S_LM = linear_model.LinearRegression()
S_Ridge = linear_model.Ridge()
S_RF = RandomForestRegressor(max_depth=2, random_state=0)
```

10.2.1. GaussianProcessRegressor as a Surrogate

- To use a Gaussian Process model from `sklearn`, that is similar to `spotpython`'s `Kriging`, we can proceed as follows:

```
kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
S_GP = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
```

- The scikit-learn GP model `S_GP` is selected for `Spot` as follows:

  `surrogate = S_GP`

- We can check the kind of surogate model with the command `isinstance`:

```
isinstance(S_GP, GaussianProcessRegressor)
```

```
True
```

```
isinstance(S_0, Kriging)
```

```
True
```

- Similar to the `Spot` run with the internal `Kriging` model, we can call the run with the `scikit-learn` surrogate:

```
fun = Analytical(seed=123).fun_branin
spot_2_GP = Spot(fun=fun,
                     fun_control=fun_control,
                     design_control=design_control,
                     surrogate = S_GP)
spot_2_GP.run()
```

```
spotpython tuning: 18.865129821249617 [######----] 55.00%
spotpython tuning: 4.066961682805861 [######----] 60.00%
spotpython tuning: 3.4619112320780285 [######----] 65.00%
spotpython tuning: 3.4619112320780285 [#######---] 70.00%
spotpython tuning: 1.3283123221495199 [########--] 75.00%
```

```
spotpython tuning: 0.9548698218896146 [########--] 80.00%
spotpython tuning: 0.9356616728510581 [########--] 85.00%
spotpython tuning: 0.39968125707661706 [#########-] 90.00%
spotpython tuning: 0.3983050744842078 [##########] 95.00%

spotpython tuning: 0.39821610604643354 [##########] 100.00% Done...

Experiment saved to 04_res.pkl

<spotpython.spot.spot.Spot at 0x156ba9d00>
```

```
spot_2_GP.plot_progress()
```

```
spot_2_GP.print_results()
```

```
min y: 0.39821610604643354
x0: 3.1496411777654334
x1: 2.272943969041002

[['x0', np.float64(3.1496411777654334)], ['x1', np.float64(2.272943969041002)]]
```

10.3. Example: One-dimensional Sphere Function With spotpython's Kriging

- In this example, we will use an one-dimensional function, which allows us to visualize the optimization process.
  - `show_models= True` is added to the argument list.

```
from spotpython.fun.objectivefunctions import Analytical
fun_control = fun_control_init(
    lower = np.array([-1]),
    upper = np.array([1]),
    fun_evals=10,
    max_time=inf,
    show_models= True,
    tolerance_x = np.sqrt(np.spacing(1)))
fun = Analytical(seed=123).fun_sphere
design_control = design_control_init(
    init_size=3)
```

```
spot_1 = Spot(fun=fun,
                    fun_control=fun_control,
                    design_control=design_control)
spot_1.run()
```

```
spotpython tuning: 0.03475493366922229 [####------] 40.00%
```

```
spotpython tuning: 0.03475493366922229 [#####-----] 50.00%
```

```
spotpython tuning: 0.014602288560505551 [######----] 60.00%
```

```
spotpython tuning: 0.00020552455663660785 [#######---] 70.00%
```

```
spotpython tuning: 5.673799497313666e-08 [########--] 80.00%
```

```
spotpython tuning: 5.673799497313666e-08 [#########-] 90.00%
```

```
spotpython tuning: 5.673799497313666e-08 [##########] 100.00% Done...

Experiment saved to 000_res.pkl
```

10.3.1. Results

```
spot_1.print_results()
```

```
min y: 5.673799497313666e-08
x0: -0.00023819738657914922
```

```
[['x0', np.float64(-0.00023819738657914922)]]
```

```
spot_1.plot_progress(log_y=True)
```

- The method `plot_model` plots the final surrogate:

```
spot_1.plot_model()
```

10.4. Example: Sklearn Model GaussianProcess

- This example visualizes the search process on the `GaussianProcessRegression` surrogate from `sklearn`.

- Therefore `surrogate = S_GP` is added to the argument list.

```
fun = Analytical(seed=123).fun_sphere
spot_1_GP = Spot(fun=fun,
                       fun_control=fun_control,
                       design_control=design_control,
                       surrogate = S_GP)
spot_1_GP.run()
```

```
spotpython tuning: 0.004925671418704527 [####------] 40.00%
```

```
spotpython tuning: 0.002612062398164981 [#####-----] 50.00%
```

```
spotpython tuning: 5.609944300870913e-07 [######----] 60.00%
```

```
spotpython tuning: 3.399776625316493e-08 [#######---] 70.00%
```

```
spotpython tuning: 2.8303204876737398e-08 [########--] 80.00%
```

```
spotpython tuning: 2.8303204876737398e-08 [#########-] 90.00%
```

```
spotpython tuning: 2.2894458385368016e-08 [##########] 100.00% Done...

Experiment saved to 000_res.pkl
```

```
spot_1_GP.print_results()
```

```
min y: 2.2894458385368016e-08
x0: 0.0001513091483862361
```

```
[['x0', np.float64(0.0001513091483862361)]]
```

```
spot_1_GP.plot_progress(log_y=True)
```

```
spot_1_GP.plot_model()
```

10.5. Exercises

10.5.1. 1. A decision tree regressor: `DecisionTreeRegressor`

- Describe the surrogate model. Use the information from the scikit-learn documentation.

- Use the surrogate as the model for optimization.

10.5.2. 2. A random forest regressor: RandomForestRegressor

- Describe the surrogate model. Use the information from the scikit-learn documentation.
- Use the surrogate as the model for optimization.

10.5.3. 3. Ordinary least squares Linear Regression: LinearRegression

- Describe the surrogate model. Use the information from the scikit-learn documentation.
- Use the surrogate as the model for optimization.

10.5.4. 4. Linear least squares with l2 regularization: Ridge

- Describe the surrogate model. Use the information from the scikit-learn documentation.
- Use the surrogate as the model for optimization.

10.5.5. 5. Gradient Boosting: HistGradientBoostingRegressor

- Describe the surrogate model. Use the information from the scikit-learn documentation.
- Use the surrogate as the model for optimization.

10.5.6. 6. Comparison of Surrogates

- Use the following two objective functions

  1. the 1-dim sphere function fun_sphere and
  2. the two-dim Branin function fun_branin:

  for a comparison of the performance of the five different surrogates:

  - spotpython's internal Kriging
  - DecisionTreeRegressor
  - RandomForestRegressor
  - linear_model.LinearRegression
  - linear_model.Ridge.

- Generate a table with the results (number of function evaluations, best function value, and best parameter vector) for each surrogate and each function as shown in Table 10.1.

Table 10.1.: Result table

surrogate	fun	fun_evals	max_time	x_0	min_y	Comments
Kriging	fun_sphere	10	inf			
Kriging	fun_branin	10	inf			
DecisionTreeRegressor	fun_sphere	10	inf			
…	…	…	…			
Ridge	fun_branin	10	inf			

- Discuss the results. Which surrogate is the best for which function? Why?

10.6. Selected Solutions

10.6.1. Solution to Exercise Section 10.5.5: Gradient Boosting

10.6.1.1. Branin: Using SPOT

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.init import fun_control_init, design_control_init
from spotpython.spot import Spot
```

- The Objective Function Branin

```
fun = Analytical().fun_branin
PREFIX = "BRANIN"
fun_control = fun_control_init(
    PREFIX=PREFIX,
    lower = np.array([-5,-0]),
    upper = np.array([10,15]),
    fun_evals=20,
    max_time=inf)

design_control = design_control_init(
    init_size=10)
```

- Running the surrogate model based optimizer Spot:

```
spot_2 = Spot(fun=fun,
                   fun_control=fun_control,
                   design_control=design_control)
spot_2.run()
```

```
spotpython tuning: 3.1468376213815015 [######----] 55.00%
spotpython tuning: 3.1468376213815015 [######----] 60.00%
spotpython tuning: 3.1468376213815015 [######----] 65.00%
spotpython tuning: 3.1468376213815015 [#######---] 70.00%
spotpython tuning: 1.1460879999819689 [########--] 75.00%
spotpython tuning: 1.0254127943018325 [########--] 80.00%
spotpython tuning: 0.42994831006071443 [########--] 85.00%
spotpython tuning: 0.4020917650024458 [#########-] 90.00%
spotpython tuning: 0.3992153710593467 [##########] 95.00%
spotpython tuning: 0.3992153710593467 [##########] 100.00% Done...

Experiment saved to BRANIN_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x1633d0da0>
```

- Print the results

```
spot_2.print_results()
```

```
min y: 0.3992153710593467
x0: 3.1555383337491234
x1: 2.2840066834425232
```

```
[['x0', np.float64(3.1555383337491234)],
 ['x1', np.float64(2.2840066834425232)]]
```

- Show the optimization progress:

```
spot_2.plot_progress(log_y=True)
```

- Generate a surrogate model plot:

```
spot_2.surrogate.plot()
```

10.6.1.2. Branin: Using Surrogates From scikit-learn

- The `HistGradientBoostingRegressor` model from `scikit-learn` is selected:

```
# Needed for the sklearn surrogates:
from sklearn.ensemble import HistGradientBoostingRegressor
import pandas as pd
S_XGB = HistGradientBoostingRegressor()
```

- The scikit-learn XGB model `S_XGB` is selected for `Spot` as follows: `surrogate = S_XGB`.
- Similar to the `Spot` run with the internal `Kriging` model, we can call the run with the `scikit-learn` surrogate:

```
fun = Analytical(seed=123).fun_branin
spot_2_XGB = Spot(fun=fun,
                     fun_control=fun_control,
                     design_control=design_control,
                     surrogate = S_XGB)
spot_2_XGB.run()
```

```
spotpython tuning: 30.69410528614059 [######----] 55.00%
spotpython tuning: 30.69410528614059 [######----] 60.00%
spotpython tuning: 30.69410528614059 [######----] 65.00%
spotpython tuning: 30.69410528614059 [#######---] 70.00%
spotpython tuning: 1.3263745845108854 [########--] 75.00%
spotpython tuning: 1.3263745845108854 [########--] 80.00%
spotpython tuning: 1.3263745845108854 [########--] 85.00%
spotpython tuning: 1.3263745845108854 [#########-] 90.00%
spotpython tuning: 1.3263745845108854 [##########] 95.00%
spotpython tuning: 1.3263745845108854 [##########] 100.00% Done...

Experiment saved to BRANIN_res.pkl


<spotpython.spot.spot.Spot at 0x1635f7020>
```

- Print the Results

```
spot_2_XGB.print_results()
```

```
min y: 1.3263745845108854
x0: -2.872730773493426
x1: 10.874313833535739


[['x0', np.float64(-2.872730773493426)],
 ['x1', np.float64(10.874313833535739)]]
```

- Show the Progress

```
spot_2_XGB.plot_progress(log_y=True)
```

- Since the `sklearn` model does not provide a `plot` method, we cannot generate a surrogate model plot.

10.6.1.3. One-dimensional Sphere Function With `spotpython`'s Kriging

- In this example, we will use an one-dimensional function, which allows us to visualize the optimization process.
  - `show_models= True` is added to the argument list.

```
from spotpython.fun.objectivefunctions import Analytical
fun_control = fun_control_init(
    lower = np.array([-1]),
    upper = np.array([1]),
    fun_evals=10,
    max_time=inf,
    show_models= True,
    tolerance_x = np.sqrt(np.spacing(1)))
fun = Analytical(seed=123).fun_sphere
design_control = design_control_init(
    init_size=3)
```

```
spot_1 = Spot(fun=fun,
                    fun_control=fun_control,
                    design_control=design_control)
spot_1.run()
```

fun_evals: 3. min_y: 0.034755
Model
True function
Samples
Last Sample
Best Sample
1.0
0.8
0.6
0.4
0.2
y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x

fun_evals: 4. min_y: 0.034755
Model
True function
Samples
Last Sample
Best Sample
1.0
0.8
0.6
0.4
0.2
y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x

```
spotpython tuning: 0.03475493366922229 [####------] 40.00%
```

```
spotpython tuning: 0.03475493366922229 [#####-----] 50.00%
```

```
spotpython tuning: 0.014602288560505551 [######----] 60.00%
```

```
spotpython tuning: 0.00020552455663660785 [#######---] 70.00%
```

```
spotpython tuning: 5.673799497313666e-08 [########--] 80.00%
```

```
spotpython tuning: 5.673799497313666e-08 [#########-] 90.00%
```

```
spotpython tuning: 5.673799497313666e-08 [##########] 100.00% Done...

Experiment saved to 000_res.pkl
```

- Print the Results

```
spot_1.print_results()
```

```
min y: 5.673799497313666e-08
x0: -0.00023819738657914922
```

```
[['x0', np.float64(-0.00023819738657914922)]]
```

- Show the Progress

```
spot_1.plot_progress(log_y=True)
```

- The method `plot_model` plots the final surrogate:

```
spot_1.plot_model()
```

10.6.1.4. One-dimensional Sphere Function With `Sklearn` Model HistGradientBoostingRegressor

- This example visualizes the search process on the `HistGradientBoostingRegressor` surrogate from `sklearn`.

- Therefore `surrogate = S_XGB` is added to the argument list.

```
fun_control = fun_control_init(
    lower = np.array([-1]),
    upper = np.array([1]),
    fun_evals=10,
    max_time=inf,
    show_models= True,
    tolerance_x = np.sqrt(np.spacing(1)))
fun = Analytical(seed=123).fun_sphere
design_control = design_control_init(
    init_size=3)
spot_1_XGB = Spot(fun=fun,
                      fun_control=fun_control,
                      design_control=design_control,
                      surrogate = S_XGB)
spot_1_XGB.run()
```

```
spotpython tuning: 0.03475493366922229 [####------] 40.00%
```

```
spotpython tuning: 0.03475493366922229 [#####-----] 50.00%
```

```
spotpython tuning: 0.03475493366922229 [######----] 60.00%
```

```
spotpython tuning: 0.03475493366922229 [#######---] 70.00%
```

```
spotpython tuning: 0.008730885505764131 [########--] 80.00%
```

```
spotpython tuning: 0.008730885505764131 [#########-] 90.00%
```

```
spotpython tuning: 0.008730885505764131 [##########] 100.00% Done...

Experiment saved to 000_res.pkl
```

```
spot_1_XGB.print_results()
```

```
min y: 0.008730885505764131
x0: 0.09343920754032609
```

```
[['x0', np.float64(0.09343920754032609)]]
```

```
spot_1_XGB.plot_progress(log_y=True)
```

```
spot_1_XGB.plot_model()
```

10.7. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

11. Sequential Parameter Optimization: Gaussian Process Models

This chapter analyzes differences between the `Kriging` implementation in `spotpython` and the `GaussianProcessRegressor` in `scikit-learn`.

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.design.spacefilling import SpaceFilling
from spotpython.spot import Spot
from spotpython.build.kriging import Kriging
from scipy.optimize import shgo
from scipy.optimize import direct
from scipy.optimize import differential_evolution
import matplotlib.pyplot as plt
import math as m
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
```

11.1. Gaussian Processes Regression: Basic Introductory `scikit-learn` Example

- This is the example from scikit-learn: https://scikit-learn.org/stable/auto_examples/gaussian_process/plot_gpr_
- After fitting our model, we see that the hyperparameters of the kernel have been optimized.
- Now, we will use our kernel to compute the mean prediction of the full dataset and plot the 95% confidence interval.

11.1.1. Train and Test Data

```
X = np.linspace(start=0, stop=10, num=1_000).reshape(-1, 1)
y = np.squeeze(X * np.sin(X))
rng = np.random.RandomState(1)
training_indices = rng.choice(np.arange(y.size), size=6, replace=False)
X_train, y_train = X[training_indices], y[training_indices]
```

11.1.2. Building the Surrogate With `Sklearn`

- The model building with `sklearn` consisits of three steps:
  1. Instantiating the model, then
  2. fitting the model (using `fit`), and
  3. making predictions (using `predict`)

```
kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
gaussian_process = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
gaussian_process.fit(X_train, y_train)
mean_prediction, std_prediction = gaussian_process.predict(X, return_std=True)
```

11.1.3. Plotting the `Sklearn`Model

```
plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
plt.fill_between(
    X.ravel(),
    mean_prediction - 1.96 * std_prediction,
    mean_prediction + 1.96 * std_prediction,
    alpha=0.5,
    label=r"95% confidence interval",
)
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("sk-learn Version: Gaussian process regression on noise-free dataset")
```

11.1.4. The spotpython Version

- The spotpython version is very similar:
  1. Instantiating the model, then
  2. fitting the model and
  3. making predictions (using predict).

```
S = Kriging(name='kriging',  seed=123, log_level=50, cod_type="norm")
S.fit(X_train, y_train)
S_mean_prediction, S_std_prediction, S_ei = S.predict(X, return_val="all")
```

```
plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, S_mean_prediction, label="Mean prediction")
plt.fill_between(
    X.ravel(),
    S_mean_prediction - 1.96 * S_std_prediction,
    S_mean_prediction + 1.96 * S_std_prediction,
    alpha=0.5,
    label=r"95% confidence interval",
)
plt.legend()
plt.xlabel("$x$")
```

```
plt.ylabel("$f(x)$")
_ = plt.title("spotpython Version: Gaussian process regression on noise-free dataset")
```

11.1.5. Visualizing the Differences Between the `spotpython` and the `sklearn` Model Fits

```
plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, S_mean_prediction, label="spotpython Mean prediction")
plt.plot(X, mean_prediction, label="Sklearn Mean Prediction")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Comparing Mean Predictions")
```

11.2. Exercises

11.2.1. Schonlau Example Function

- The Schonlau Example Function is based on sample points only (there is no analytical function description available):

```
X = np.linspace(start=0, stop=13, num=1_000).reshape(-1, 1)
X_train = np.array([1., 2., 3., 4., 12.]).reshape(-1,1)
y_train = np.array([0., -1.75, -2, -0.5, 5.])
```

- Describe the function.
- Compare the two models that were build using the `spotpython` and the `sklearn` surrogate.
- Note: Since there is no analytical function available, you might be interested in adding some points and describe the effects.

11.2.2. Forrester Example Function

- The Forrester Example Function is defined as follows:

```
f(x) = (6x- 2)^2 sin(12x-4) for x in [0,1].
```

- Data points are generated as follows:

```
from spotpython.utils.init import fun_control_init
X = np.linspace(start=-0.5, stop=1.5, num=1_000).reshape(-1, 1)
X_train = np.array([0.0, 0.175, 0.225, 0.3, 0.35, 0.375, 0.5,1]).reshape(-1,1)
fun = Analytical().fun_forrester
fun_control = fun_control_init(sigma = 0.1)
y = fun(X, fun_control=fun_control)
y_train = fun(X_train, fun_control=fun_control)
```

- Describe the function.
- Compare the two models that were build using the `spotpython` and the `sklearn` surrogate.
- Note: Modify the noise level (`"sigma"`), e.g., use a value of `0.2`, and compare the two models.

```
fun_control = fun_control_init(sigma = 0.2)
```

11.2.3. `fun_runge` Function (1-dim)

- The Runge function is defined as follows:

```
f(x) = 1/ (1 + sum(x_i))^2
```

- Data points are generated as follows:

```
gen = SpaceFilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = Analytical().fun_runge
fun_control = fun_control_init(sigma = 0.025)
X_train = gen.scipy_lhd(10, lower=lower, upper = upper).reshape(-1,1)
y_train = fun(X, fun_control=fun_control)
X = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
y = fun(X, fun_control=fun_control)
```

- Describe the function.
- Compare the two models that were build using the `spotpython` and the `sklearn` surrogate.
- Note: Modify the noise level (`"sigma"`), e.g., use a value of `0.05`, and compare the two models.

```
fun_control = fun_control_init(sigma = 0.5)
```

11.2.4. fun_cubed (1-dim)

- The Cubed function is defined as follows:

  `np.sum(X[i]** 3)`

- Data points are generated as follows:

```
gen = SpaceFilling(1)
rng = np.random.RandomState(1)
fun_control = fun_control_init(sigma = 0.025,
                lower = np.array([-10]),
                upper = np.array([10]))
fun = Analytical().fun_cubed
X_train = gen.scipy_lhd(10, lower=lower, upper = upper).reshape(-1,1)
y_train = fun(X, fun_control=fun_control)
X = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
y = fun(X, fun_control=fun_control)
```

- Describe the function.
- Compare the two models that were build using the `spotpython` and the `sklearn` surrogate.
- Note: Modify the noise level (`"sigma"`), e.g., use a value of `0.05`, and compare the two models.

```
fun_control = fun_control_init(sigma = 0.025)
```

11.2.5. The Effect of Noise

How does the behavior of the `spotpython` fit changes when the argument `noise` is set to `True`, i.e.,

`S = Kriging(name='kriging',  seed=123, n_theta=1, noise=True)`

is used?

12. Expected Improvement

This chapter describes, analyzes, and compares different infill criterion. An infill criterion defines how the next point x_{n+1} is selected from the surrogate model S. Expected improvement is a popular infill criterion in Bayesian optimization.

12.1. Example: `Spot` and the 1-dim Sphere Function

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.utils.init import fun_control_init, surrogate_control_init, design_control_init
import matplotlib.pyplot as plt
```

12.1.1. The Objective Function: 1-dim Sphere

- The `spotpython` package provides several classes of objective functions.
- We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x) = x^2$$

```
fun = Analytical().fun_sphere
```

- The size of the `lower` bound vector determines the problem dimension.
- Here we will use `np.array([-1])`, i.e., a one-dim function.

> **TensorBoard**
>
> Similar to the one-dimensional case, which was introduced in Section Section 7.7, we can use TensorBoard to monitor the progress of the optimization. We will use the same code, only the prefix is different:

```
from spotpython.utils.init import fun_control_init
PREFIX = "07_Y"
fun_control = fun_control_init(
    PREFIX=PREFIX,
    fun_evals = 25,
    lower = np.array([-1]),
    upper = np.array([1]),
    tolerance_x = np.sqrt(np.spacing(1)),)
design_control = design_control_init(init_size=10)
```

```
spot_1 = Spot(
            fun=fun,
            fun_control=fun_control,
            design_control=design_control)
spot_1.run()
```

```
spotpython tuning: 4.960293502265715e-09 [####------] 44.00%
spotpython tuning: 4.960293502265715e-09 [#####-----] 48.00%
spotpython tuning: 3.91132389612039e-09 [#####-----] 52.00%
spotpython tuning: 3.91132389612039e-09 [######----] 56.00%
spotpython tuning: 2.8792582932753584e-09 [######----] 60.00%
spotpython tuning: 2.8792582932753584e-09 [######----] 64.00%
spotpython tuning: 2.8792582932753584e-09 [#######---] 68.00%
spotpython tuning: 2.8792582932753584e-09 [#######---] 72.00%
spotpython tuning: 2.8792582932753584e-09 [########--] 76.00%
spotpython tuning: 2.8792582932753584e-09 [########--] 80.00%
spotpython tuning: 5.121302337041985e-10 [########--] 84.00%
spotpython tuning: 5.121302337041985e-10 [#########-] 88.00%
spotpython tuning: 9.915881842777748e-12 [#########-] 92.00%
spotpython tuning: 9.915881842777748e-12 [##########] 96.00%
spotpython tuning: 9.915881842777748e-12 [##########] 100.00% Done...

Experiment saved to 07_Y_res.pkl


<spotpython.spot.spot.Spot at 0x143456600>
```

12.1.2. Results

```
spot_1.print_results()
```

```
min y: 9.915881842777748e-12
x0: -3.1489493236280808e-06

[['x0', np.float64(-3.1489493236280808e-06)]]
```

```
spot_1.plot_progress(log_y=True)
```

Figure 12.1.: TensorBoard visualization of the spotpython optimization process and the surrogate model.

12.2. Same, but with EI as infill_criterion

```
PREFIX = "07_EI_ISO"
fun_control = fun_control_init(
    PREFIX=PREFIX,
    lower = np.array([-1]),
    upper = np.array([1]),
    fun_evals = 25,
    tolerance_x = np.sqrt(np.spacing(1)),
    infill_criterion = "ei")
```

```
spot_1_ei = Spot(fun=fun,
                     fun_control=fun_control)
spot_1_ei.run()
```

```
spotpython tuning: 1.0205727057090308e-08 [####------] 44.00%
spotpython tuning: 1.0205727057090308e-08 [#####-----] 48.00%
spotpython tuning: 1.0205727057090308e-08 [#####-----] 52.00%
spotpython tuning: 1.0205727057090308e-08 [######----] 56.00%
spotpython tuning: 2.2781224716456335e-14 [######----] 60.00%
spotpython tuning: 2.2781224716456335e-14 [######----] 64.00%
spotpython tuning: 2.2781224716456335e-14 [#######---] 68.00%
spotpython tuning: 2.2781224716456335e-14 [#######---] 72.00%
spotpython tuning: 2.2781224716456335e-14 [########--] 76.00%
spotpython tuning: 2.2781224716456335e-14 [########--] 80.00%
spotpython tuning: 2.2781224716456335e-14 [########--] 84.00%
spotpython tuning: 2.2781224716456335e-14 [#########-] 88.00%
spotpython tuning: 2.2781224716456335e-14 [#########-] 92.00%
spotpython tuning: 2.2781224716456335e-14 [##########] 96.00%
spotpython tuning: 2.2781224716456335e-14 [##########] 100.00% Done...

Experiment saved to 07_EI_ISO_res.pkl

<spotpython.spot.spot.Spot at 0x1437d6060>
```

```
spot_1_ei.plot_progress(log_y=True)
```

```
spot_1_ei.print_results()
```

```
min y: 2.2781224716456335e-14
x0: 1.5093450472458687e-07
```

```
[['x0', np.float64(1.5093450472458687e-07)]]
```

Figure 12.2.: TensorBoard visualization of the spotpython optimization process and the surrogate model. Expected improvement, isotropic Kriging.

12.3. Non-isotropic Kriging

```
PREFIX = "07_EI_NONISO"
fun_control = fun_control_init(
    PREFIX=PREFIX,
    lower = np.array([-1, -1]),
    upper = np.array([1, 1]),
    fun_evals = 25,
    tolerance_x = np.sqrt(np.spacing(1)),
    infill_criterion = "ei")
surrogate_control = surrogate_control_init(
    n_theta=2,
    noise=False,
    )
```

```
spot_2_ei_noniso = Spot(fun=fun,
                    fun_control=fun_control,
                    surrogate_control=surrogate_control)
spot_2_ei_noniso.run()
```

```
spotpython tuning: 1.885355036033269e-05 [####------] 44.00%
spotpython tuning: 1.885355036033269e-05 [#####-----] 48.00%
spotpython tuning: 1.885355036033269e-05 [#####-----] 52.00%
spotpython tuning: 1.885355036033269e-05 [######----] 56.00%
spotpython tuning: 1.885355036033269e-05 [######----] 60.00%
spotpython tuning: 1.885355036033269e-05 [######----] 64.00%
spotpython tuning: 1.885355036033269e-05 [#######---] 68.00%
```

```
spotpython tuning: 1.885355036033269e-05 [#######---] 72.00%
spotpython tuning: 9.720933590577543e-06 [########--] 76.00%
spotpython tuning: 8.679050871673044e-06 [########--] 80.00%
spotpython tuning: 8.679050871673044e-06 [########--] 84.00%
spotpython tuning: 8.679050871673044e-06 [#########-] 88.00%
spotpython tuning: 8.679050871673044e-06 [#########-] 92.00%
spotpython tuning: 8.679050871673044e-06 [##########] 96.00%
spotpython tuning: 8.679050871673044e-06 [##########] 100.00% Done...

Experiment saved to 07_EI_NONISO_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x1457dc680>
```

```
spot_2_ei_noniso.plot_progress(log_y=True)
```

```
spot_2_ei_noniso.print_results()
```

```
min y: 8.679050871673044e-06
x0: -0.0029115430445807067
x1: -0.000449408468129777
```

```
[['x0', np.float64(-0.0029115430445807067)],
 ['x1', np.float64(-0.000449408468129777)]]
```

```
spot_2_ei_noniso.surrogate.plot()
```

Figure 12.3.: TensorBoard visualization of the spotpython optimization process and the surrogate model. Expected improvement, isotropic Kriging.

12.4. Using sklearn Surrogates

12.4.1. The spot Loop

The spot loop consists of the following steps:

1. Init: Build initial design X
2. Evaluate initial design on real objective f: $y = f(X)$
3. Build surrogate: $S = S(X, y)$
4. Optimize on surrogate: $X_0 = \text{optimize}(S)$
5. Evaluate on real objective: $y_0 = f(X_0)$
6. Impute (Infill) new points: $X = X \cup X_0$, $y = y \cup y_0$.

7. Got 3.

The `spot` loop is implemented in `R` as follows:

Figure 12.4.: Visual representation of the model based search with SPOT. Taken from: Bartz-Beielstein, T., and Zaefferer, M. Hyperparameter tuning approaches. In Hyperparameter Tuning for Machine and Deep Learning with R - A Practical Guide, E. Bartz, T. Bartz-Beielstein, M. Zaefferer, and O. Mersmann, Eds. Springer, 2022, ch. 4, pp. 67–114.

12.4.2. spot: The Initial Model

12.4.2.1. Example: Modifying the initial design size

This is the "Example: Modifying the initial design size" from Chapter 4.5.1 in [bart21i].

```
spot_ei = Spot(fun=fun,
                fun_control=fun_control_init(
                lower = np.array([-1,-1]),
                upper= np.array([1,1])),
                design_control = design_control_init(init_size=5))
spot_ei.run()
```

```
spotpython tuning: 0.13771720107579405 [####------] 40.00%
spotpython tuning: 0.008747581912500914 [#####-----] 46.67%
spotpython tuning: 0.002833855020194859 [#####-----] 53.33%
```

```
spotpython tuning: 0.0008113730206162874 [######----] 60.00%
spotpython tuning: 0.0003658334488102912 [#######---] 66.67%
spotpython tuning: 0.000357376362957228 [#######---] 73.33%
spotpython tuning: 0.000357376362957228 [########--] 80.00%
spotpython tuning: 0.00032569308461158667 [#########-] 86.67%
spotpython tuning: 0.000272252838237925 [#########-] 93.33%
spotpython tuning: 0.0001495137205828153 [##########] 100.00% Done...

Experiment saved to 000_res.pkl

<spotpython.spot.spot.Spot at 0x145708380>
```

```
spot_ei.plot_progress()
```

```
np.min(spot_1.y), np.min(spot_ei.y)
```

```
(np.float64(9.915881842777748e-12), np.float64(0.0001495137205828153))
```

12.4.3. Init: Build Initial Design

```
from spotpython.design.spacefilling import SpaceFilling
from spotpython.build.kriging import Kriging
from spotpython.fun.objectivefunctions import Analytical
gen = SpaceFilling(2)
rng = np.random.RandomState(1)
lower = np.array([-5,-0])
upper = np.array([10,15])
fun = Analytical().fun_branin
```

```
X = gen.scipy_lhd(10, lower=lower, upper = upper)
print(X)
y = fun(X, fun_control=fun_control)
print(y)
```

```
[[ 8.97647221 13.41926847]
 [ 0.66946019  1.22344228]
 [ 5.23614115 13.78185824]
 [ 5.6149825  11.5851384 ]
 [-1.72963184  1.66516096]
 [-4.26945568  7.1325531 ]
 [ 1.26363761 10.17935555]
 [ 2.88779942  8.05508969]
 [-3.39111089  4.15213772]
 [ 7.30131231  5.22275244]]
[128.95676449  31.73474356 172.89678121 126.71295908  64.34349975
  70.16178611  48.71407916  31.77322887  76.91788181  30.69410529]
```

```
S = Kriging(name='kriging',  seed=123)
S.fit(X, y)
S.plot()
```

```
gen = SpaceFilling(2, seed=123)
X0 = gen.scipy_lhd(3)
gen = SpaceFilling(2, seed=345)
X1 = gen.scipy_lhd(3)
X2 = gen.scipy_lhd(3)
gen = SpaceFilling(2, seed=123)
X3 = gen.scipy_lhd(3)
X0, X1, X2, X3
```

```
(array([[0.77254938, 0.31539299],
        [0.59321338, 0.93854273],
        [0.27469803, 0.3959685 ]]),
 array([[0.78373509, 0.86811887],
        [0.06692621, 0.6058029 ],
        [0.41374778, 0.00525456]]),
 array([[0.121357  , 0.69043832],
        [0.41906219, 0.32838498],
        [0.86742658, 0.52910374]]),
 array([[0.77254938, 0.31539299],
        [0.59321338, 0.93854273],
        [0.27469803, 0.3959685 ]]))
```

12.4.4. Evaluate

12.4.5. Build Surrogate

12.4.6. A Simple Predictor

The code below shows how to use a simple model for prediction.

- Assume that only two (very costly) measurements are available:
  1. f(0) = 0.5
  2. f(2) = 2.5
- We are interested in the value at $x_0 = 1$, i.e., $f(x_0 = 1)$, but cannot run an additional, third experiment.

```
from sklearn import linear_model
X = np.array([[0], [2]])
y = np.array([0.5, 2.5])
S_lm = linear_model.LinearRegression()
S_lm = S_lm.fit(X, y)
```

```
X0 = np.array([[1]])
y0 = S_lm.predict(X0)
print(y0)
```

```
[1.5]
```

- Central Idea:
  - Evaluation of the surrogate model `S_lm` is much cheaper (or / and much faster) than running the real-world experiment f.

12.5. Gaussian Processes regression: basic introductory example

This example was taken from scikit-learn. After fitting our model, we see that the hyperparameters of the kernel have been optimized. Now, we will use our kernel to compute the mean prediction of the full dataset and plot the 95% confidence interval.

```
import numpy as np
import matplotlib.pyplot as plt
import math as m
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF

X = np.linspace(start=0, stop=10, num=1_000).reshape(-1, 1)
y = np.squeeze(X * np.sin(X))
rng = np.random.RandomState(1)
training_indices = rng.choice(np.arange(y.size), size=6, replace=False)
X_train, y_train = X[training_indices], y[training_indices]

kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
gaussian_process = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
gaussian_process.fit(X_train, y_train)
gaussian_process.kernel_

mean_prediction, std_prediction = gaussian_process.predict(X, return_std=True)

plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
plt.fill_between(
    X.ravel(),
```

```
    mean_prediction - 1.96 * std_prediction,
    mean_prediction + 1.96 * std_prediction,
    alpha=0.5,
    label=r"95% confidence interval",
)
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("sk-learn Version: Gaussian process regression on noise-free dataset")
```

```
from spotpython.build.kriging import Kriging
import numpy as np
import matplotlib.pyplot as plt
rng = np.random.RandomState(1)
X = np.linspace(start=0, stop=10, num=1_000).reshape(-1, 1)
y = np.squeeze(X * np.sin(X))
training_indices = rng.choice(np.arange(y.size), size=6, replace=False)
X_train, y_train = X[training_indices], y[training_indices]


S = Kriging(name='kriging',  seed=123, log_level=50, cod_type="norm")
S.fit(X_train, y_train)

mean_prediction, std_prediction, ei = S.predict(X, return_val="all")
```

```
std_prediction

plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
plt.fill_between(
    X.ravel(),
    mean_prediction - 1.96 * std_prediction,
    mean_prediction + 1.96 * std_prediction,
    alpha=0.5,
    label=r"95% confidence interval",
)
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("spotpython Version: Gaussian process regression on noise-free dataset")
```

12.6. The Surrogate: Using scikit-learn models

Default is the internal `kriging` surrogate.

```
S_0 = Kriging(name='kriging', seed=123)
```

Models from `scikit-learn` can be selected, e.g., Gaussian Process:

```
# Needed for the sklearn surrogates:
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn import linear_model
from sklearn import tree
import pandas as pd
```

```
kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
S_GP = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
```

- and many more:

```
S_Tree = DecisionTreeRegressor(random_state=0)
S_LM = linear_model.LinearRegression()
S_Ridge = linear_model.Ridge()
S_RF = RandomForestRegressor(max_depth=2, random_state=0)
```

- The scikit-learn GP model `S_GP` is selected.

```
S = S_GP
```

```
isinstance(S, GaussianProcessRegressor)
```

```
True
```

```
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical().fun_branin
fun_control = fun_control_init(
    lower = np.array([-5,-0]),
    upper = np.array([10,15]),
    fun_evals = 15)
design_control = design_control_init(init_size=5)
spot_GP = Spot(fun=fun,
                    fun_control=fun_control,
                    surrogate=S,
                    design_control=design_control)
spot_GP.run()
```

```
spotpython tuning: 24.51465459019188 [####------] 40.00%
spotpython tuning: 11.003092545432404 [#####-----] 46.67%
spotpython tuning: 11.003092545432404 [#####-----] 53.33%
spotpython tuning: 7.281405479109784 [######----] 60.00%
spotpython tuning: 7.281405479109784 [#######---] 66.67%
spotpython tuning: 7.281405479109784 [#######---] 73.33%
spotpython tuning: 2.9520033012954237 [########--] 80.00%
spotpython tuning: 2.9520033012954237 [#########-] 86.67%
spotpython tuning: 2.1049818033904044 [#########-] 93.33%
spotpython tuning: 1.9431597967021723 [##########] 100.00% Done...

Experiment saved to 000_res.pkl

<spotpython.spot.spot.Spot at 0x15f703d40>
```

```
spot_GP.y
```

```
array([ 69.32459936, 152.38491454, 107.92560483,  24.51465459,
        76.73500031,  86.30426863,  11.00309255,  16.11758333,
         7.28140548,  21.82343562,  10.96088904,   2.9520033 ,
         3.02912616,   2.1049818 ,   1.9431598 ])
```

```
spot_GP.plot_progress()
```

```
spot_GP.print_results()
```

```
min y: 1.9431597967021723
x0: 10.0
x1: 2.99858238342458

[['x0', np.float64(10.0)], ['x1', np.float64(2.99858238342458)]]
```

12.7. Additional Examples

```
# Needed for the sklearn surrogates:
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn import linear_model
from sklearn import tree
import pandas as pd
```

```
kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
S_GP = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
```

```
from spotpython.build.kriging import Kriging
import numpy as np
import spotpython
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot

S_K = Kriging(name='kriging',
              seed=123,
              log_level=50,
              infill_criterion = "y",
              n_theta=1,
              noise=False,
              cod_type="norm")
fun = Analytical().fun_sphere

fun_control = fun_control_init(
    lower = np.array([-1,-1]),
    upper = np.array([1,1]),
    fun_evals = 25)

spot_S_K = Spot(fun=fun,
                     fun_control=fun_control,
                     surrogate=S_K,
                     design_control=design_control,
                     surrogate_control=surrogate_control)
spot_S_K.run()
```

```
spotpython tuning: 0.13771716894083716 [##--------] 24.00%
```

```
spotpython tuning: 0.008764900158613986 [###-------] 28.00%
spotpython tuning: 0.002831737294424899 [###-------] 32.00%
spotpython tuning: 0.0008144336474759649 [####------] 36.00%
spotpython tuning: 0.000363982666025624 [####------] 40.00%
spotpython tuning: 0.0003615841465166041 [####------] 44.00%
spotpython tuning: 0.0003590011672749327 [#####-----] 48.00%
spotpython tuning: 0.00032913641643097483 [#####-----] 52.00%
spotpython tuning: 0.0002791331313588125 [######----] 56.00%
spotpython tuning: 0.00016536102611694684 [######----] 60.00%
spotpython tuning: 1.979364042364845e-05 [######----] 64.00%
spotpython tuning: 2.328711577671373e-06 [#######---] 68.00%
spotpython tuning: 5.408003176528451e-07 [#######---] 72.00%
spotpython tuning: 4.501997119079259e-07 [########--] 76.00%
spotpython tuning: 3.902062597093855e-07 [########--] 80.00%
spotpython tuning: 1.9521044693395355e-07 [########--] 84.00%
spotpython tuning: 1.684568701593145e-07 [#########-] 88.00%
spotpython tuning: 1.684568701593145e-07 [#########-] 92.00%
spotpython tuning: 1.684568701593145e-07 [##########] 96.00%
spotpython tuning: 1.684568701593145e-07 [##########] 100.00% Done...

Experiment saved to 000_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x15f72a720>
```

```
spot_S_K.plot_progress(log_y=True)
```

```
spot_S_K.surrogate.plot()
```

```
spot_S_K.print_results()
```

```
min y: 1.684568701593145e-07
x0: 0.0003249898265724749
x1: 0.0002506760514762174

[['x0', np.float64(0.0003249898265724749)],
 ['x1', np.float64(0.0002506760514762174)]]
```

12.7.1. Optimize on Surrogate

12.7.2. Evaluate on Real Objective

12.7.3. Impute / Infill new Points

12.8. Tests

```python
import numpy as np
from spotpython.spot import Spot
from spotpython.fun.objectivefunctions import Analytical

fun_sphere = Analytical().fun_sphere

fun_control = fun_control_init(
                    lower=np.array([-1, -1]),
                    upper=np.array([1, 1]),
                    n_points = 2)
spot_1 = Spot(
    fun=fun_sphere,
    fun_control=fun_control,
)

# (S-2) Initial Design:
spot_1.X = spot_1.design.scipy_lhd(
    spot_1.design_control["init_size"], lower=spot_1.lower, upper=spot_1.upper
)
print(spot_1.X)

# (S-3): Eval initial design:
spot_1.y = spot_1.fun(spot_1.X)
print(spot_1.y)

spot_1.fit_surrogate()
X0 = spot_1.suggest_new_X()
print(X0)
assert X0.size == spot_1.n_points * spot_1.k
```

```
[[ 0.86352963  0.7892358 ]
 [-0.24407197 -0.83687436]
 [ 0.36481882  0.8375811 ]
 [ 0.415331    0.54468512]
 [-0.56395091 -0.77797854]
 [-0.90259409 -0.04899292]
 [-0.16484832  0.35724741]
 [ 0.05170659  0.07401196]
 [-0.78548145 -0.44638164]
 [ 0.64017497 -0.30363301]]
[1.36857656 0.75992983 0.83463487 0.46918172 0.92329124 0.8170764
 0.15480068 0.00815134 0.81623768 0.502017  ]
[[0.00160545 0.00421075]
 [0.00171842 0.00407727]]
```

12.9. EI: The Famous Schonlau Example

```
X_train0 = np.array([1, 2, 3, 4, 12]).reshape(-1,1)
X_train = np.linspace(start=0, stop=10, num=5).reshape(-1, 1)
```

```
from spotpython.build.kriging import Kriging
import numpy as np
import matplotlib.pyplot as plt

X_train = np.array([1., 2., 3., 4., 12.]).reshape(-1,1)
y_train = np.array([0., -1.75, -2, -0.5, 5.])

S = Kriging(name='kriging',  seed=123, log_level=50, n_theta=1, noise=False, cod_type="norm")
S.fit(X_train, y_train)

X = np.linspace(start=0, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
if True:
    plt.fill_between(
        X.ravel(),
        mean_prediction - 2 * std_prediction,
        mean_prediction + 2 * std_prediction,
        alpha=0.5,
        label=r"95% confidence interval",
    )
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noise-free dataset")
```

```
#plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
# plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, -ei, label="Expected Improvement")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noise-free dataset")
```

```
S.log
```

```
{'negLnLike': array([1.20788205]),
 'theta': array([-0.99002508]),
 'p': [],
 'Lambda': []}
```

12.10. EI: The Forrester Example

```
from spotpython.build.kriging import Kriging
import numpy as np
import matplotlib.pyplot as plt
import spotpython
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot

# exact x locations are unknown:
X_train = np.array([0.0, 0.175, 0.225, 0.3, 0.35, 0.375, 0.5,1]).reshape(-1,1)
```

```
fun = Analytical().fun_forrester
fun_control = fun_control_init(
    PREFIX="07_EI_FORRESTER",
    sigma=1.0,
    seed=123,)
y_train = fun(X_train, fun_control=fun_control)

S = Kriging(name='kriging',  seed=123, log_level=50, n_theta=1, noise=False, cod_type=
S.fit(X_train, y_train)

X = np.linspace(start=0, stop=1, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, mean_prediction, label="Mean prediction")
if True:
    plt.fill_between(
        X.ravel(),
        mean_prediction - 2 * std_prediction,
        mean_prediction + 2 * std_prediction,
        alpha=0.5,
        label=r"95% confidence interval",
    )
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noise-free dataset")
```

```
#plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
# plt.scatter(X_train, y_train, label="Observations")
plt.plot(X, -ei, label="Expected Improvement")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noise-free dataset")
```

12.11. Noise

```
import numpy as np
import spotpython
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.design.spacefilling import SpaceFilling
from spotpython.build.kriging import Kriging
import matplotlib.pyplot as plt

gen = SpaceFilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = Analytical().fun_sphere
fun_control = fun_control_init(
    PREFIX="07_Y",
    sigma=2.0,
    seed=123,)
X = gen.scipy_lhd(10, lower=lower, upper = upper)
```

```
print(X)
y = fun(X, fun_control=fun_control)
print(y)
y.shape
X_train = X.reshape(-1,1)
y_train = y

S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=False)
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

#plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression on noisy dataset")
```

```
[[ 0.63529627]
 [-4.10764204]
 [-0.44071975]
 [ 9.63125638]
 [-8.3518118 ]
 [-3.62418901]
 [ 4.15331   ]
 [ 3.4468512 ]
 [ 6.36049088]
 [-7.77978539]]
[-1.57464135 16.13714981  2.77008442 93.14904827 71.59322218 14.28895359
 15.9770567  12.96468767 39.82265329 59.88028242]
```

```
S.log
```

```
{'negLnLike': array([26.18505386]),
 'theta': array([-1.10547472]),
 'p': [],
 'Lambda': []}
```

```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=True)
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

#plt.plot(X, y, label=r"$f(x) = x \sin(x)$", linestyle="dotted")
plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
```

```
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression with nugget on noisy dataset")
```

```
S.log
```

```
{'negLnLike': array([21.82276721]),
 'theta': array([-2.94197609]),
 'p': [],
 'Lambda': array([4.89634062e-05])}
```

12.12. Cubic Function

```
import numpy as np
import spotpython
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.design.spacefilling import SpaceFilling
from spotpython.build.kriging import Kriging
```

```
import matplotlib.pyplot as plt

gen = SpaceFilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = Analytical().fun_cubed
fun_control = fun_control_init(
    PREFIX="07_Y",
    sigma=10.0,
    seed=123,)

X = gen.scipy_lhd(10, lower=lower, upper = upper)
print(X)
y = fun(X, fun_control=fun_control)
print(y)
y.shape
X_train = X.reshape(-1,1)
y_train = y

S = Kriging(name='kriging',  seed=123, log_level=50, n_theta=1, noise=False)
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Cubed: Gaussian process regression on noisy dataset")
```

```
[[ 0.63529627]
 [-4.10764204]
 [-0.44071975]
 [ 9.63125638]
 [-8.3518118 ]
 [-3.62418901]
 [ 4.15331   ]
 [ 3.4468512 ]
 [ 6.36049088]
 [-7.77978539]]
```

```
[  -9.63480707  -72.98497325   12.7936499   895.34567477 -573.35961837
  -41.83176425   65.27989461   46.37081417  254.1530734  -474.09587355]
```

```
S = Kriging(name='kriging',  seed=123, log_level=0, n_theta=1, noise=True)
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Cubed: Gaussian process with nugget regression on noisy dataset")
```

```
import numpy as np
import spotpython
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.design.spacefilling import SpaceFilling
from spotpython.build.kriging import Kriging
import matplotlib.pyplot as plt

gen = SpaceFilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = Analytical().fun_runge
fun_control = fun_control_init(
    PREFIX="07_Y",
    sigma=0.25,
    seed=123,)

X = gen.scipy_lhd(10, lower=lower, upper = upper)
print(X)
y = fun(X, fun_control=fun_control)
print(y)
y.shape
X_train = X.reshape(-1,1)
y_train = y
```

```
S = Kriging(name='kriging',  seed=123, log_level=50, n_theta=1, noise=False)
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression on noisy dataset")
```

```
[[ 0.63529627]
 [-4.10764204]
 [-0.44071975]
 [ 9.63125638]
 [-8.3518118 ]
 [-3.62418901]
 [ 4.15331   ]
 [ 3.4468512 ]
 [ 6.36049088]
 [-7.77978539]]
[ 0.46517267 -0.03599548  1.15933822  0.05915901  0.24419145  0.21502359
 -0.10432134  0.21312309 -0.05502681 -0.06434374]
```

```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=True)
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression with nugget on noisy dataset")
```

12.13. Modifying Lambda Search Space

```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=True,
            min_Lambda=0.1,
            max_Lambda=10)
S.fit(X_train, y_train)

print(f"Lambda: {S.Lambda}")
```

```
Lambda: 0.8567558302695025
```

```
X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
```

```
#plt.plot(X, ei, label="Expected Improvement")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Gaussian process regression with nugget on noisy dataset. Modified Lamb
```

13. Handling Noise

This chapter demonstrates how noisy functions can be handled by `Spot` and how noise can be simulated, i.e., added to the objective function.

13.1. Example: `Spot` and the Noisy Sphere Function

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
import matplotlib.pyplot as plt
from spotpython.utils.init import fun_control_init, get_spot_tensorboard_path
from spotpython.utils.init import fun_control_init, design_control_init, surrogate_control_init

PREFIX = "08"
```

13.1.1. The Objective Function: Noisy Sphere

The `spotpython` package provides several classes of objective functions, which return a one-dimensional output $y = f(x)$ for a given input x (independent variable). Several objective functions allow one- or multidimensional input, some also combinations of real-valued and categorial input values.

An objective function is considered as "analytical" if it can be described by a closed mathematical formula, e.g.,

$$f(x, y) = x^2 + y^2.$$

To simulate measurement errors, adding artificial noise to the function value y is a common practice, e.g.,:

$$f(x, y) = x^2 + y^2 + \epsilon.$$

Usually, noise is assumed to be normally distributed with mean $\mu = 0$ and standard deviation σ. spotpython uses numpy's `scale` parameter, which specifies the

standard deviation (spread or "width") of the distribution is used. This must be a non-negative value, see https://numpy.org/doc/stable/reference/random/generated/numpy.random.normal.html.

Example: The sphere function without noise

The default setting does not use any noise.

```
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical().fun_sphere
x = np.linspace(-1,1,100).reshape(-1,1)
y = fun(x)
plt.figure()
plt.plot(x,y, "k")
plt.show()
```

Example: The sphere function with noise

Noise can be added to the sphere function as follows:

```
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical(seed=123, sigma=0.02).fun_sphere
x = np.linspace(-1,1,100).reshape(-1,1)
y = fun(x)
plt.figure()
plt.plot(x,y, "k")
plt.show()
```

13.1.2. Reproducibility: Noise Generation and Seed Handling

spotpython provides two mechanisms for generating random noise:

1. The seed is initialized once, i.e., when the objective function is instantiated. This can be done using the following call: `fun = Analytical(sigma=0.02, seed=123).fun_sphere`.
2. The seed is set every time the objective function is called. This can be done using the following call: `y = fun(x, sigma=0.02, seed=123)`.

These two different ways lead to different results as explained in the following tables:

Example: Noise added to the sphere function

Since `sigma` is set to `0.02`, noise is added to the function:

```
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical(sigma=0.02, seed=123).fun_sphere
x = np.array([1]).reshape(-1,1)
for i in range(3):
    print(f"{i}: {fun(x)}")
```

```
0: [0.98021757]
1: [0.99264427]
2: [1.02575851]
```

The seed is set once. Every call to `fun()` results in a different value. The whole experiment can be repeated, the initial seed is used to generate the same sequence as shown below:

Example: Noise added to the sphere function

Since `sigma` is set to `0.02`, noise is added to the function:

```
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical(sigma=0.02, seed=123).fun_sphere
x = np.array([1]).reshape(-1,1)
for i in range(3):
    print(f"{i}: {fun(x)}")
```

```
0: [0.98021757]
1: [0.99264427]
2: [1.02575851]
```

If `spotpython` is used as a hyperparameter tuner, it is important that only one realization of the noise function is optimized. This behaviour can be accomplished by passing the same seed via the dictionary `fun_control` to every call of the objective function `fun` as shown below:

Example: The same noise added to the sphere function

Since `sigma` is set to `0.02`, noise is added to the function:

```
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical().fun_sphere
fun_control = fun_control_init(
    PREFIX=PREFIX,
    sigma=0.02)
y = fun(x, fun_control=fun_control)
x = np.array([1]).reshape(-1,1)
for i in range(3):
    print(f"{i}: {fun(x)}")
```

```
0: [0.98021757]
1: [0.98021757]
2: [0.98021757]
```

13.2. spotpython's Noise Handling Approaches

The following setting will be used for the next steps:

```
fun = Analytical().fun_sphere
fun_control = fun_control_init(
    PREFIX=PREFIX,
    sigma=0.02,
)
```

`spotpython` is adopted as follows to cope with noisy functions:

1. `fun_repeats` is set to a value larger than 1 (here: 2)
2. `noise` is set to `true`. Therefore, a nugget (`Lambda`) term is added to the correlation matrix
3. `init size` (of the `design_control` dictionary) is set to a value larger than 1 (here: 3)

```
spot_1_noisy = Spot(fun=fun,
                    fun_control=fun_control_init(
                                    lower = np.array([-1]),
                                    upper = np.array([1]),
                                    fun_evals = 20,
                                    fun_repeats = 2,
                                    noise = True,
                                    show_models=True),
                    design_control=design_control_init(init_size=3, repeats=2),
                    surrogate_control=surrogate_control_init(noise=True))
```

```
spot_1_noisy.run()
```

```
spotpython tuning: 0.03475287368052604 [####------] 40.00%
```

```
spotpython tuning: 0.03232164139525234 [#####-----] 50.00%
```

```
spotpython tuning: 0.015588763085618356 [######----] 60.00%
```

```
spotpython tuning: 0.0009558222960697881 [#######---] 70.00%
```

```
spotpython tuning: 5.565145515980805e-05 [########--] 80.00%
```

```
spotpython tuning: 7.349927945791069e-07 [#########-] 90.00%
```

```
spotpython tuning: 4.3889499806904336e-07 [##########] 100.00% Done...

Experiment saved to 000_res.pkl
```

13.3. Print the Results

```
spot_1_noisy.print_results()
```

```
min y: 4.3889499806904336e-07
min mean y: 4.3889499806904336e-07
x0: 0.0006624915079222098

[['x0', np.float64(0.0006624915079222098)]]
```

```
spot_1_noisy.plot_progress(log_y=False,
    filename="./figures/" + PREFIX + "_progress.png")
```

Figure 13.1.: Progress plot. *Black* dots denote results from the initial design. *Red* dots illustrate the improvement found by the surrogate model based optimization.

13.4. Noise and Surrogates: The Nugget Effect

13.4.1. The Noisy Sphere

13.4.1.1. The Data

- We prepare some data first:

```
import numpy as np
import spotpython
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.design.spacefilling import SpaceFilling
from spotpython.build.kriging import Kriging
import matplotlib.pyplot as plt

gen = SpaceFilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = Analytical().fun_sphere
fun_control = fun_control_init(
    PREFIX=PREFIX,
    sigma=4)
X = gen.scipy_lhd(10, lower=lower, upper = upper)
y = fun(X, fun_control=fun_control)
```

```
X_train = X.reshape(-1,1)
y_train = y
```

- A surrogate without nugget is fitted to these data:

```
S = Kriging(name='kriging',
            n_theta=1,
            noise=False)
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression on noisy dataset")
```

- In comparison to the surrogate without nugget, we fit a surrogate with nugget to the data:

```
S_nug = Kriging(name='kriging',
            n_theta=1,
            noise=True)
S_nug.fit(X_train, y_train)
X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S_nug.predict(X_axis, return_val="all")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression with nugget on noisy dataset")
```

- The value of the nugget term can be extracted from the model as follows:

```
S.Lambda
```

```
S_nug.Lambda
```

```
np.float64(0.0005592170940621742)
```

- We see:
  - the first model `S` has no nugget,
  - whereas the second model has a nugget value (`Lambda`) larger than zero.

13.5. Exercises

13.5.1. Noisy `fun_cubed`

- Analyse the effect of noise on the `fun_cubed` function with the following settings:

```
fun = Analytical().fun_cubed
fun_control = fun_control_init(
    sigma=10)
lower = np.array([-10])
upper = np.array([10])
```

13.5.2. `fun_runge`

- Analyse the effect of noise on the `fun_runge` function with the following settings:

```
lower = np.array([-10])
upper = np.array([10])
fun = Analytical().fun_runge
fun_control = fun_control_init(
    sigma=0.25)
```

13.5.3. `fun_forrester`

- Analyse the effect of noise on the `fun_forrester` function with the following settings:

```
lower = np.array([0])
upper = np.array([1])
fun = Analytical().fun_forrester
fun_control = fun_control_init(
    sigma=5)
```

13.5.4. `fun_xsin`

- Analyse the effect of noise on the `fun_xsin` function with the following settings:

```
lower = np.array([-1.])
upper = np.array([1.])
fun = Analytical().fun_xsin
fun_control = fun_control_init(
    sigma=0.5)
```

14. Optimal Computational Budget Allocation in Spot

This chapter demonstrates how noisy functions can be handled with Optimal Computational Budget Allocation (OCBA) by Spot.

14.1. Example: Spot, OCBA, and the Noisy Sphere Function

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
import matplotlib.pyplot as plt
from spotpython.utils.init import fun_control_init, get_spot_tensorboard_path
from spotpython.utils.init import fun_control_init, design_control_init, surrogate_control_init

PREFIX = "09"
```

14.1.1. The Objective Function: Noisy Sphere

The spotpython package provides several classes of objective functions. We will use an analytical objective function with noise, i.e., a function that can be described by a (closed) formula:

$$f(x) = x^2 + \epsilon$$

Since sigma is set to 0.1, noise is added to the function:

```
fun = Analytical().fun_sphere
fun_control = fun_control_init(
    PREFIX=PREFIX,
    sigma=0.1)
```

A plot illustrates the noise:

```
x = np.linspace(-1,1,100).reshape(-1,1)
y = fun(x, fun_control=fun_control)
plt.figure()
plt.plot(x,y, "k")
plt.show()
```

Spot is adopted as follows to cope with noisy functions:

1. fun_repeats is set to a value larger than 1 (here: 2)
2. noise is set to true. Therefore, a nugget (Lambda) term is added to the correlation matrix
3. init size (of the design_control dictionary) is set to a value larger than 1 (here: 2)

```
spot_1_noisy = Spot(fun=fun,
                    fun_control=fun_control_init(
                    lower = np.array([-1]),
                    upper = np.array([1]),
                    fun_evals = 20,
                    fun_repeats = 2,
                    infill_criterion="ei",
                    noise = True,
                    tolerance_x=0.0,
```

```
                    ocba_delta = 1,
                    show_models=True),
                    design_control=design_control_init(init_size=3, repeats=2),
                    surrogate_control=surrogate_control_init(noise=True))
```

```
spot_1_noisy.run()
```

```
spotpython tuning: 0.008919341332677182 [####------] 40.00%
```

```
spotpython tuning: 0.008919341332677182 [#####-----] 50.00%
```

```
spotpython tuning: 6.82082870070088e-05 [######----] 60.00%
```

```
spotpython tuning: 3.692698488681066e-06 [#######---] 70.00%
```

```
spotpython tuning: 1.3740935234320857e-06 [########--] 80.00%
```

```
spotpython tuning: 1.3740935234320857e-06 [#########-] 90.00%
```

```
spotpython tuning: 1.3740935234320857e-06 [##########] 100.00% Done...

Experiment saved to 000_res.pkl
```

14.2. Print the Results

```
spot_1_noisy.print_results()
```

```
min y: 1.3740935234320857e-06
min mean y: 1.3740935234320857e-06
x0: -0.001172217353323216
```

```
[['x0', np.float64(-0.001172217353323216)]]
```

```
spot_1_noisy.plot_progress(log_y=False)
```

14.3. Noise and Surrogates: The Nugget Effect

14.3.1. The Noisy Sphere

14.3.1.1. The Data

We prepare some data first:

```
import numpy as np
import spotpython
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.design.spacefilling import SpaceFilling
from spotpython.build.kriging import Kriging
import matplotlib.pyplot as plt

gen = SpaceFilling(1)
rng = np.random.RandomState(1)
lower = np.array([-10])
upper = np.array([10])
fun = Analytical().fun_sphere
fun_control = fun_control_init(
    sigma=2,
    seed=125)
X = gen.scipy_lhd(10, lower=lower, upper = upper)
y = fun(X, fun_control=fun_control)
X_train = X.reshape(-1,1)
y_train = y
```

A surrogate without nugget is fitted to these data:

```
S = Kriging(name='kriging',
            seed=123,
            log_level=50,
            n_theta=1,
            noise=False)
S.fit(X_train, y_train)

X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S.predict(X_axis, return_val="all")

plt.scatter(X_train, y_train, label="Observations")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression on noisy dataset")
```

In comparison to the surrogate without nugget, we fit a surrogate with nugget to the data:

```
S_nug = Kriging(name='kriging',
            seed=123,
```

```
            log_level=50,
            n_theta=1,
            noise=True)
S_nug.fit(X_train, y_train)
X_axis = np.linspace(start=-13, stop=13, num=1000).reshape(-1, 1)
mean_prediction, std_prediction, ei = S_nug.predict(X_axis, return_val="all")
plt.scatter(X_train, y_train, label="Observations")
plt.plot(X_axis, mean_prediction, label="mue")
plt.legend()
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
_ = plt.title("Sphere: Gaussian process regression with nugget on noisy dataset")
```

The value of the nugget term can be extracted from the model as follows:

```
S.Lambda
```

```
S_nug.Lambda
```

```
np.float64(9.867759985778659e-05)
```

We see:

- the first model `S` has no nugget,
- whereas the second model has a nugget value (`Lambda`) larger than zero.

14.4. Exercises

14.4.1. Noisy `fun_cubed`

Analyse the effect of noise on the `fun_cubed` function with the following settings:

```
fun = Analytical().fun_cubed
fun_control = fun_control_init(
    sigma=10,
    seed=123)
lower = np.array([-10])
upper = np.array([10])
```

14.4.2. `fun_runge`

Analyse the effect of noise on the `fun_runge` function with the following settings:

```
lower = np.array([-10])
upper = np.array([10])
fun = Analytical().fun_runge
fun_control = fun_control_init(
    sigma=0.25,
    seed=123)
```

14.4.3. `fun_forrester`

Analyse the effect of noise on the `fun_forrester` function with the following settings:

```
lower = np.array([0])
upper = np.array([1])
fun = Analytical().fun_forrester
fun_control = {"sigma": 5,
               "seed": 123}
```

14.4.4. `fun_xsin`

Analyse the effect of noise on the `fun_xsin` function with the following settings:

```
lower = np.array([-1.])
upper = np.array([1.])
fun = Analytical().fun_xsin
fun_control = fun_control_init(
    sigma=0.5,
    seed=123)
```

15. Kriging with Varying Correlation-p

This chapter illustrates the difference between Kriging models with varying p. The difference is illustrated with the help of the `spotpython` package.

15.1. Example: `Spot` Surrogate and the 2-dim Sphere Function

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from spotpython.utils.init import fun_control_init, surrogate_control_init
PREFIX="015"
```

15.1.1. The Objective Function: 2-dim Sphere

- The `spotpython` package provides several classes of objective functions.
- We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x, y) = x^2 + y^2$$

- The size of the `lower` bound vector determines the problem dimension.
- Here we will use `np.array([-1, -1])`, i.e., a two-dim function.

```
fun = Analytical().fun_sphere
fun_control = fun_control_init(PREFIX=PREFIX,
                               lower = np.array([-1, -1]),
                               upper = np.array([1, 1]))
```

- Although the default `spot` surrogate model is an isotropic Kriging model, we will explicitly set the `theta` parameter to a value of `1` for both dimensions. This is done to illustrate the difference between isotropic and anisotropic Kriging models.

```
surrogate_control=surrogate_control_init(n_p=1,
                                         p_val=2.0,)
```

```
spot_2 = Spot(fun=fun,
                   fun_control=fun_control,
                   surrogate_control=surrogate_control)

spot_2.run()
```

```
spotpython tuning: 1.5904060546935205e-05 [#######---] 73.33%
spotpython tuning: 1.5904060546935205e-05 [########--] 80.00%
spotpython tuning: 1.5904060546935205e-05 [#########-] 86.67%
spotpython tuning: 1.5904060546935205e-05 [#########-] 93.33%
spotpython tuning: 1.2084513018724136e-05 [##########] 100.00% Done...

Experiment saved to 015_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x12ff62b40>
```

15.1.2. Results

```
spot_2.print_results()
```

```
min y: 1.2084513018724136e-05
x0: -0.003294464459178357
x1: -0.0011095120305498227
```

```
[['x0', np.float64(-0.003294464459178357)],
 ['x1', np.float64(-0.0011095120305498227)]]
```

```
spot_2.plot_progress(log_y=True)
```

```
spot_2.surrogate.plot()
```

15.2. Example With Modified p

- We can use set `p` to a value other than `2` to obtain a different Kriging model.

```
surrogate_control = surrogate_control_init(n_p=1,
                                           p_val=1.0)
spot_2_p1= Spot(fun=fun,
```

```
                    fun_control=fun_control,
                    surrogate_control=surrogate_control)
spot_2_p1.run()
```

```
spotpython tuning: 1.5904060546935205e-05 [#######---] 73.33%
spotpython tuning: 1.5904060546935205e-05 [########--] 80.00%
spotpython tuning: 1.5904060546935205e-05 [#########-] 86.67%
spotpython tuning: 1.5904060546935205e-05 [#########-] 93.33%
spotpython tuning: 1.2084513018724136e-05 [##########] 100.00% Done...

Experiment saved to 015_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x14f5d2180>
```

- The search progress of the optimization with the anisotropic model can be visualized:

```
spot_2_p1.plot_progress(log_y=True)
```

```
spot_2_p1.print_results()
```

```
min y: 1.2084513018724136e-05
x0: -0.003294464459178357
x1: -0.0011095120305498227

[['x0', np.float64(-0.003294464459178357)],
 ['x1', np.float64(-0.0011095120305498227)]]
```

```
spot_2_p1.surrogate.plot()
```

15.2.1. Taking a Look at the p Values

15.2.1.1. p Values from the spot Model

- We can check, which p values the spot model has used:
- The p values from the surrogate can be printed as follows:

```
spot_2_p1.surrogate.p
```

```
array([1.])
```

- Since the surrogate from the isotropic setting was stored as spot_2, we can also take a look at the theta value from this model:

```
spot_2.surrogate.p
```

```
array([2.])
```

15.3. Optimization of the p Values

```
surrogate_control = surrogate_control_init(n_p=1,
                                           optim_p=True)
spot_2_pm= Spot(fun=fun,
                    fun_control=fun_control,
                    surrogate_control=surrogate_control)
spot_2_pm.run()
```

```
spotpython tuning: 1.7257202516906525e-05 [#######---] 73.33%
spotpython tuning: 1.7257202516906525e-05 [########--] 80.00%
spotpython tuning: 1.7257202516906525e-05 [#########-] 86.67%
spotpython tuning: 1.7257202516906525e-05 [#########-] 93.33%
spotpython tuning: 1.5016013698596226e-05 [##########] 100.00% Done...

Experiment saved to 015_res.pkl

<spotpython.spot.spot.Spot at 0x14f56ac60>
```

```
spot_2_pm.plot_progress(log_y=True)
```

```
spot_2_pm.print_results()
```

```
min y: 1.5016013698596226e-05
x0: -0.0038508851250580807
x1: 0.00043208500576001916

[['x0', np.float64(-0.0038508851250580807)],
 ['x1', np.float64(0.00043208500576001916)]]
```

```
spot_2_pm.surrogate.plot()
```

```
spot_2_pm.surrogate.p
```

```
[np.float64(1.4233686495123274)]
```

15.4. Optimization of Multiple p Values

```
surrogate_control = surrogate_control_init(n_p=2,
                                           optim_p=True)
spot_2_pmo= Spot(fun=fun,
                 fun_control=fun_control,
                 surrogate_control=surrogate_control)
spot_2_pmo.run()
```

```
spotpython tuning: 1.9920765737363724e-05 [#######---] 73.33%
spotpython tuning: 1.9920765737363724e-05 [########--] 80.00%
spotpython tuning: 1.9920765737363724e-05 [#########-] 86.67%
spotpython tuning: 1.9920765737363724e-05 [#########-] 93.33%
```

```
spotpython tuning: 9.190384771923126e-06 [##########] 100.00% Done...

Experiment saved to 015_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x14fd30d10>
```

```
spot_2_pmo.plot_progress(log_y=True)
```

```
spot_2_pmo.print_results()
```

```
min y: 9.190384771923126e-06
x0: -0.0028761470296043675
x1: -0.0009582082425136512
```

```
[['x0', np.float64(-0.0028761470296043675)],
 ['x1', np.float64(-0.0009582082425136512)]]
```

```
spot_2_pmo.surrogate.plot()
```

```
spot_2_pmo.surrogate.p
```

```
[np.float64(1.1410902802804275), np.float64(1.5986000604554)]
```

15.5. Exercises

15.5.1. fun_branin

- Describe the function.
  - The input dimension is 2. The search range is $-5 \leq x_1 \leq 10$ and $0 \leq x_2 \leq 15$.
- Compare the results from `spotpython` runs with different options for `p`.
- Modify the termination criterion: instead of the number of evaluations (which is specified via `fun_evals`), the time should be used as the termination criterion. This can be done as follows (`max_time=1` specifies a run time of one minute):

```
fun_evals=inf,
max_time=1,
```

15.5.2. `fun_sin_cos`

- Describe the function.
  - The input dimension is `2`. The search range is $-2\pi \leq x_1 \leq 2\pi$ and $-2\pi \leq x_2 \leq 2\pi$.
- Compare the results from `spotpython` run a) with isotropic and b) anisotropic surrogate models.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

15.5.3. `fun_runge`

- Describe the function.
  - The input dimension is `2`. The search range is $-5 \leq x_1 \leq 5$ and $-5 \leq x_2 \leq 5$.
- Compare the results from `spotpython` runs with different options for `p`.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

15.5.4. `fun_wingwt`

- Describe the function.
  - The input dimension is `10`. The search ranges are between 0 and 1 (values are mapped internally to their natural bounds).
- Compare the results from `spotpython` runs with different options for `p`.
- Modify the termination criterion (`max_time` instead of `fun_evals`) as described for `fun_branin`.

15.6. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

16. Factorial Variables

Until now, we have considered continuous variables. However, in many applications, the variables are not continuous, but rather discrete or categorical. For example, the number of layers in a neural network, the number of trees in a random forest, or the type of kernel in a support vector machine are all discrete variables. In the following, we will consider a simple example with two numerical variables and one categorical variable.

```
from spotpython.design.spacefilling import SpaceFilling
from spotpython.build.kriging import Kriging
from spotpython.fun.objectivefunctions import Analytical
import numpy as np
```

First, we generate the test data set for fitting the Kriging model. We use the `SpaceFilling` class to generate the first two diemnsion of $n = 30$ design points. The third dimension is a categorical variable, which can take the values 0, 1, or 2.

```
gen = SpaceFilling(2)
n = 30
rng = np.random.RandomState(1)
lower = np.array([-5,-0])
upper = np.array([10,15])
fun_orig = Analytical().fun_branin
fun = Analytical().fun_branin_factor

X0 = gen.scipy_lhd(n, lower=lower, upper = upper)
X1 = np.random.randint(low=0, high=3, size=(n,))
X = np.c_[X0, X1]
print(X[:5,:])
```

```
[[-2.84117593  5.97308949  1.        ]
 [-3.61017994  6.90781409  1.        ]
 [ 9.91204705  5.09395275  2.        ]
 [-4.4616725   1.3617128   2.        ]
 [-2.40987728  8.05505365  0.        ]]
```

The objective function is the `fun_branin_factor` in the `analytical` class [SOURCE]. It calculates the Branin function of (x_1, x_2) with an additional factor based on the value of x_3. If $x_3 = 1$, the value of the Branin function is increased by 10. If $x_3 = 2$, the value of the Branin function is decreased by 10. Otherwise, the value of the Branin function is not changed.

```
y = fun(X)
y_orig = fun_orig(X0)
data = np.c_[X, y_orig, y]
print(data[:5,:])
```

```
[[ -2.84117593   5.97308949   1.          32.09388125  42.09388125]
 [ -3.61017994   6.90781409   1.          43.965223    53.965223  ]
 [  9.91204705   5.09395275   2.           6.25588575  -3.74411425]
 [ -4.4616725    1.3617128    2.         212.41884106 202.41884106]
 [ -2.40987728   8.05505365   0.           9.25981051   9.25981051]]
```

We fit two Kriging models, one with three numerical variables and one with two numerical variables and one categorical variable. We then compare the predictions of the two models.

```
S = Kriging(name='kriging',  seed=123, log_level=50, n_theta=3, noise=False, var_type=
S.fit(X, y)
Sf = Kriging(name='kriging',  seed=123, log_level=50, n_theta=3, noise=False, var_type
Sf.fit(X, y)
```

We can now compare the predictions of the two models. We generate a new test data set and calculate the sum of the absolute differences between the predictions of the two models and the true values of the objective function. If the categorical variable is important, the sum of the absolute differences should be smaller than if the categorical variable is not important.

```
n = 100
k = 100
y_true = np.zeros(n*k)
y_pred= np.zeros(n*k)
y_factor_pred= np.zeros(n*k)
for i in range(k):
  X0 = gen.scipy_lhd(n, lower=lower, upper = upper)
  X1 = np.random.randint(low=0, high=3, size=(n,))
  X = np.c_[X0, X1]
  a = i*n
  b = (i+1)*n
```

```
  y_true[a:b] = fun(X)
  y_pred[a:b] = S.predict(X)
  y_factor_pred[a:b] = Sf.predict(X)
```

```
import pandas as pd
df = pd.DataFrame({"y":y_true, "Prediction":y_pred, "Prediction_factor":y_factor_pred})
df.head()
```

	y	Prediction	Prediction_factor
0	16.684749	7.660267	16.050572
1	105.865258	99.777210	107.161260
2	39.811774	37.643524	39.791177
3	18.177150	19.459025	19.391307
4	10.968377	1.963544	4.948437

```
df.tail()
```

	y	Prediction	Prediction_factor
9995	73.620503	74.147679	73.697531
9996	96.187178	90.787843	98.600971
9997	39.494401	41.717765	40.101268
9998	15.390268	51.425758	20.522226
9999	26.261264	28.013805	26.048805

```
s=np.sum(np.abs(y_pred - y_true))
sf=np.sum(np.abs(y_factor_pred - y_true))
res = (sf - s)
print(res)
```

```
-46409.07666539212
```

```
from spotpython.plot.validation import plot_actual_vs_predicted
plot_actual_vs_predicted(y_test=df["y"], y_pred=df["Prediction"], title="Default")
plot_actual_vs_predicted(y_test=df["y"], y_pred=df["Prediction_factor"], title="Factor")
```

16.1. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

17. User-Specified Functions: Extending the Analytical Class

This chapter illustrates how user-specified functions can be optimized and analyzed with the `spotpython` package by extending the `Analytical` class.

```
import numpy as np
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.init import fun_control_init, surrogate_control_init
from spotpython.spot import Spot
```

17.1. The Objective Function: User Specified

We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x) = \sum_{i}^{k} x_i^4.$$

This function is continuous, convex and unimodal. The global minimum is

$$f(x) = 0, \text{at } x = (0, 0, \dots, 0).$$

- The `Analytical` class can be extended as follows:

```
from typing import Optional, Dict

class UserAnalytical(Analytical):
    def fun_user_function(self, X: np.ndarray, fun_control: Optional[Dict] = None) -> np.ndarray:
        """
        Custom new function: f(x) = x^4

        Args:
            X (np.ndarray): Input data as a 2D array.
            fun_control (Optional[Dict]): Control parameters for the function.
```

```
        Returns:
            np.ndarray: Computed values with optional noise.

        Examples:
            >>> import numpy as np
            >>> X = np.array([[1, 2, 3], [4, 5, 6]])
            >>> fun = UserAnalytical()
            >>> fun.fun_user_function(X)
        """
        X = self._prepare_input_data(X, fun_control)

        offset = np.ones(X.shape[1]) * self.offset
        y = np.sum((X - offset) **4, axis=1)

        # Add noise if specified in fun_control
        return self._add_noise(y)
```

```
user_fun = UserAnalytical()
X = np.array([[0, 0, 0], [1, 1, 1]])
results = user_fun.fun_user_function(X)
print(results)
```

```
[0. 3.]
```

```
user_fun = UserAnalytical(offset=1.0)
X = np.array([[0, 0, 0], [1, 1, 1]])
results = user_fun.fun_user_function(X)
print(results)
```

```
[3. 0.]
```

```
user_fun = UserAnalytical(sigma=1.0)
X = np.array([[0, 0, 0], [1, 1, 1]])
results = user_fun.fun_user_function(X)
print(results)
```

```
[0.06691138 3.11495313]
```

```
user_fun = UserAnalytical().fun_user_function
fun_control = fun_control_init(
              PREFIX="USER",
```

```
              lower = -1.0*np.ones(2),
              upper = np.ones(2),
              var_name=["User Pressure", "User Temp"],
              TENSORBOARD_CLEAN=True,
              tensorboard_log=True)
spot_user = Spot(fun=user_fun,
                 fun_control=fun_control)
spot_user.run()
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_02_12_22_14_53_0
Created spot_tensorboard_path: runs/spot_logs/USER_maans08_2025-02-12_22-14-53 for SummaryWriter()
spotpython tuning: 3.715394917589437e-05 [#######---] 73.33%
spotpython tuning: 3.715394917589437e-05 [########--] 80.00%
spotpython tuning: 3.715394917589437e-05 [#########-] 86.67%
spotpython tuning: 3.715394917589437e-05 [#########-] 93.33%
spotpython tuning: 1.3840535984457729e-05 [##########] 100.00% Done...

Experiment saved to USER_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x11652e960>
```

17.2. Results

```
_ = spot_user.print_results()
```

```
min y: 1.3840535984457729e-05
User Pressure: 0.060781734267030006
User Temp: 0.020927320560756916
```

```
spot_user.plot_progress()
```

17.3. A Contour Plot

We can select two dimensions, say $i = 0$ and $j = 1$, and generate a contour plot as follows.

> **Note:**
>
> We have specified identical `min_z` and `max_z` values to generate comparable plots.

```
spot_user.plot_contour(i=0, j=1, min_z=0, max_z=2.25)
```

- The variable importance:

```
_ = spot_user.print_importance()
```

```
User Pressure:  77.95830652608618
User Temp:  100.0
```

```
spot_user.plot_importance()
```

17.4. Jupyter Notebook

> **Note**
>
> - The Jupyter-Notebook of this lecture is available on GitHub in the Hyperparameter-Tuning-Cookbook Repository

Part III.

Data-Driven Modeling and Optimization

18. Data-Driven Modeling and Optimization

18.1. StatQuest Videos

18.1.1. June, 11th 2024

18.1.1.1. Histograms

- Video: Histograms, Clearly Explained

Exercise 18.1 (Histograms)**.** Problems with histograms?

18.1.1.2. Probability Distributions

- Video: The Main Ideas behind Probability Distributions

Exercise 18.2 (Smaller Bins)**.** What happens when we use smaller bins in a histogram?

Exercise 18.3 (Density Curve)**.** Why plot a curve to approximate a histogram?

18.1.1.3. Normal Distribution

- Video: The Normal Distribution, Clearly Explained!!!

Exercise 18.4 (TwoSDQuestion)**.** How many samples are plus/minus two SD around the mean?

Exercise 18.5 (OneSDQuestion)**.** How many samples are plus/minus one SD around the mean?

Exercise 18.6 (ThreeSDQuestion)**.** How many samples are plus/minus three SD around the mean?

Exercise 18.7 (DataRangeQuestion)**.** You have a mean at 100 and a SD of 10. Where are 95% of the data?

Exercise 18.8 (PeakHeightQuestion)**.** If the peak is very high, is the SD low or high?

18.1.1.4. The mean, the media, and the mode

18.1.1.5. The exponential distribution

18.1.1.6. Population and Estimated Parameters

- Video: Population and Estimated Parameters, Clearly Explained

Exercise 18.9 (ProbabilityQuestion)**.** If we have a certain curve and want to calculate the probability of values equal to 20 if the mean is 20.

18.1.1.7. Mean, Variance, and Standard Deviation

- Video: Calculating the Mean, Variance, and Standard Deviation

Exercise 18.10 (MeanDifferenceQuestion)**.** The difference between μ and x-bar?

Exercise 18.11 (EstimateMeanQuestion)**.** How do you calculate the sample mean?

Exercise 18.12 (SigmaSquaredQuestion)**.** What is sigma squared?

Exercise 18.13 (EstimatedSDQuestion)**.** What is the formula for the estimated standard deviation?

Exercise 18.14 (VarianceDifferenceQuestion)**.** Difference between the variance and the estimated variance?

18.1.2. Mathematical Models

- Video: What is a mathematical model?

Exercise 18.15 (ModelBenefitsQuestion)**.** What are the benefits of using models?

18.1.2.1. Sampling from a Distribution

- Vidoe: Sampling from a Distribution, Clearly Explained!!!

Exercise 18.16 (SampleDefinitionQuestion)**.** What is a sample in statistics?

18.1.3. Hypothesis Testing and the Null-Hypothesis

Exercise 18.17 (RejectHypothesisQuestion)**.** What does it mean to reject a hypothesis?

Exercise 18.18 (NullHypothesisQuestion)**.** What is a null hypothesis?

Exercise 18.19 (BetterDrugQuestion)**.** How can you show that you have found a better drug?

18.1.3.1. Alternative Hypotheses, Main Ideas

18.1.3.2. p-values: What they are and how to interpret them

Exercise 18.20 (PValueIntroductionQuestion)**.** What is the reason for introducing the p-value?

Exercise 18.21 (PValueRangeQuestion)**.** Is there any range for p-values? Can it be negative?

Exercise 18.22 (PValueRangeQuestion)**.** Is there any range for p-values? Can it be negative?

Exercise 18.23 (TypicalPValueQuestion)**.** What are typical values of the p-value and what does it mean? 5%?

Exercise 18.24 (FalsePositiveQuestion)**.** What is a false-positive?

18.1.3.3. How to calculate p-values

Exercise 18.25 (CalculatePValueQuestion)**.** How to calculate p-value?

Exercise 18.26 (SDCalculationQuestion)**.** What is the SD if the mean is 155 and in the range from 142 - 169 there are 95% of the data?

Exercise 18.27 (SidedPValueQuestion)**.** When do we need the two-sided p-value and when the one-sided?

Exercise 18.28 (CoinTestQuestion)**.** Test a coin with Tail-Head-Head. What is the p-value?

Exercise 18.29 (BorderPValueQuestion)**.** If you get exactly the 0.05 border value, can you reject?

Exercise 18.30 (OneSidedPValueCautionQuestion)**.** Why should you be careful with a one-sided p-test?

Exercise 18.31 (BinomialDistributionQuestion)**.** What is the binomial distribution?

18.1.3.4. p-hacking: What it is and how to avoid it

Exercise 18.32 (PHackingWaysQuestion)**.** Name two typical ways of p-hacking.

Exercise 18.33 (AvoidPHackingQuestion)**.** How can p-hacking be avoided?

Exercise 18.34 (MultipleTestingProblemQuestion)**.** What is the multiple testing problem?

18.1.3.5. Covariance

Exercise 18.35 (CovarianceDefinitionQuestion)**.** What is covariance?

Exercise 18.36 (CovarianceMeaningQuestion)**.** What is the meaning of covariance?

Exercise 18.37 (CovarianceVarianceRelationshipQuestion)**.** What is the relationship between covariance and variance?

Exercise 18.38 (HighCovarianceQuestion)**.** If covariance is high, is there a strong relationship?

Exercise 18.39 (ZeroCovarianceQuestion)**.** What if the covariance is zero?

Exercise 18.40 (NegativeCovarianceQuestion)**.** Can covariance be negative?

Exercise 18.41 (NegativeVarianceQuestion)**.** Can variance be negative?

18.1.3.6. Pearson's Correlation

Video: [Pearson's Correlation, Clearly Explained]

Exercise 18.42 (CorrelationValueQuestion)**.** What do you do if the correlation value is 10?

Exercise 18.43 (CorrelationRangeQuestion)**.** What is the possible range of correlation values?

Exercise 18.44 (CorrelationFormulaQuestion)**.** What is the formula for correlation?

18.1.3.7. Boxplots

18.1.4. June, 18th 2024

18.1.4.1. Statistical Power

- Video: Statistical Power, Clearly Explained

Exercise 18.45 (UnderstandingStatisticalPower)**.** What is the definition of power in a statistical test?

Exercise 18.46 (DistributionEffectOnPower)**.** What is the implication for power analysis if the samples come from the same distribution?

Exercise 18.47 (IncreasingPower)**.** How can you increase the power if the distributions are very similar?

Exercise 18.48 (PreventingPHacking)**.** What should be done to avoid p-hacking when the distributions are close to each other?

Exercise 18.49 (SampleSizeAndPower)**.** If there is overlap and the sample size is small, will the power be high or low?

18.1.4.2. Power Analysis

- Video: Power Analysis, Clearly Explained!!!

Exercise 18.50 (FactorsAffectingPower)**.** Which are the two main factors that affect power?

Exercise 18.51 (PurposeOfPowerAnalysis)**.** What does power analysis tell us?

Exercise 18.52 (ExperimentRisks)**.** What are the two risks faced when performing an experiment?

Exercise 18.53 (PerformingPowerAnalysis)**.** How do you perform a power analysis?

18.1.4.3. The Central Limit Theorem

- Video: The Central Limit Theorem, Clearly Explained!!!

Exercise 18.54 (CentralLimitTheoremExplanation)**.** What does the Central Limit Theorem state?

18.1.4.4. Boxplots

- Video: Boxplots are Awesome

Exercise 18.55 (MedianInBoxplot)**.** What is represented by the middle line in a boxplot?

Exercise 18.56 (BoxContentInBoxplot)**.** What does the box in a boxplot represent?

18.1.4.5. R-squared

- Video: R-squared, Clearly Explained

Exercise 18.57 (RSquaredDefinition)**.** What is R-squared? Show the formula.

Exercise 18.58 (NegativeRSquared)**.** Can the R-squared value be negative?

Exercise 18.59 (RSquaredCalculation)**.** Perform a calculation involving R-squared.

18.1.4.6. The main ideas of fitting a line to data (The main ideas of least squares and linear regression.)

- Video: The main ideas of fitting a line to data (The main ideas of least squares and linear regression.)

Exercise 18.60 (LeastSquaresMeaning)**.** What is the meaning of the least squares method?

18.1.4.7. Linear Regression

- Video: Linear Regression, Clearly Explained

18.1.4.8. Multiple Regression

- Video: Multiple Regression, Clearly Explained

18.1.4.9. A Gentle Introduction to Machine Learning

- Video: A Gentle Introduction to Machine Learning

Exercise 18.61 (RegressionVsClassification)**.** What is the difference between regression and classification?

18.1.5. Maximum Likelihood

Maximum likelihood estimation is a method used to estimate the parameters of a statistical model. It is based on the principle of choosing the parameter values that maximize the likelihood of the observed data. The likelihood function represents the probability of observing the data given the model parameters. By maximizing this likelihood, we can find the parameter values that best explain the observed data.

Example 18.1 (Maximum Likelihood Estimation: Bernoulli Experiment)**.** Bernoulli experiment for the event A, repeated n times, with the probability of success p. Result given as n tuple with entries A and $\overline{A}$. A appears k times. The probability of this event is given by

$$L(p) = p^k(1-p)^{n-k} \tag{18.1}$$

Applying maximum likelihood estimation, we find the maximum of the likelihood function $L(p)$, i.e., we are trying to find the value of p that maximizes the probability of observing the data. This value will be denoted as $\hat{p}$.

Differentiating the likelihood function with respect to p and setting the derivative to zero, we find the maximum likelihood estimate $\hat{p}$. We get

$$\frac{d}{dp}L(p) = kp^{k-1}(1-p)^{n-k} - p^k(n-k)(1-p)^{n-k-1} \tag{18.2}$$

$$= p^{k-1}(1-p)^{n-k-1}\left(k(1-p) - p(n-k)\right) = 0 \tag{18.3}$$

Because

$$p \neq 0 \text{ and } (1-p)p \neq 0,$$

we can divide by $p^{k-1}(1-p)^{n-k-1}$ and get

$$k(1-p) - p(n-k) = 0. \tag{18.4}$$

Solving for p gives

$$\hat{p} = \frac{k}{n} \tag{18.5}$$

Therefore, the maximum likelihood estimate for the probability of success in a Bernoulli experiment is the ratio of the number of successes to the total number of trials.

□

Example 18.2 (Maximum Likelihood Estimation: Normal Distribution)**.** Random variable $X \sim \mathcal{N}(\mu, \sigma^2)$ with n observations $x_1, x_2, \dots, x_n$. The likelihood function is given by

$$L(x_1, x_2, \dots, x_n, \mu, \sigma^2) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x_i-\mu)^2}{2\sigma^2}\right) \tag{18.6}$$

Taking the logarithm of the likelihood function, we get

$$\log L(x_1, x_2, \dots, x_n, \mu, \sigma^2) = -\frac{n}{2}\log(2\pi) - \frac{n}{2}\log(\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2 \tag{18.7}$$

Partial derivative with respect to μ is

$$\frac{\partial}{\partial\mu}\log L(x_1, x_2, \dots, x_n, \mu, \sigma^2) = \frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i-\mu) = 0 \tag{18.8}$$

We obtain the maximum likelihood estimate for μ as

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{18.9}$$

The partial derivative with respect to σ^2 is

$$\frac{\partial}{\partial \sigma^2} \log L(x_1, x_2, \dots, x_n, \mu, \sigma^2) = -\frac{n}{2\sigma^2} + \frac{1}{2(\sigma^2)^2} \sum_{i=1}^{n} (x_i - \mu)^2 = 0 \tag{18.10}$$

This can be simplified to

$$-n + \frac{1}{\sigma^2} \sum_{i=1}^{n} (x_i - \mu)^2 = 0 \tag{18.11}$$

$$\Rightarrow n\sigma^2 = \sum_{i=1}^{n} (x_i - \mu)^2 \tag{18.12}$$

Using the maximum likelihood estimate for μ, we get

$$\hat{\sigma}^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \hat{\mu})^2 \tag{18.13}$$

$$= \frac{n-1}{n} \frac{\sum_{i=1}^{n} (x_i - \hat{\mu})^2 n - 1}{=} \frac{n-1}{n} s^2, \tag{18.14}$$

where

$$s = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \overline{x})}{n-1}} \tag{18.15}$$

is the sample standard deviation. We obtain the maximum likelihood estimate for σ^2 as

$$\hat{\sigma}^2 = \frac{n-1}{n} s^2 \tag{18.16}$$

- Video: Maximum Likelihood, clearly explained!!!

Exercise 18.62 (LikelihoodConcept)**.** What is the idea of likelihood?

- Video: Probability is not Likelihood. Find out why!!!

Exercise 18.63 (ProbabilityVsLikelihood)**.** What is the difference between probability and likelihood?

18.1.5.1. Cross-Validation

- Video: Machine Learning Fundamentals: Cross Validation

Exercise 18.64 (TrainVsTestData)**.** What is the difference between training and testing data?

Exercise 18.65 (SingleValidationIssue)**.** What is the problem if you validate the model only once?

Exercise 18.66 (FoldDefinition)**.** What is a fold in cross-validation?

Exercise 18.67 (LeaveOneOutValidation)**.** What is leave-one-out cross-validation?

18.1.5.2. The Confusion Matrix

- Video: Machine Learning Fundamentals: The Confusion Matrix

Exercise 18.68 (DrawingConfusionMatrix)**.** Draw the confusion matrix.

18.1.5.3. Sensitivity and Specificity

- Video: Machine Learning Fundamentals: Sensitivity and Specificity

Exercise 18.69 (SensitivitySpecificityCalculation1)**.** Calculate the sensitivity and specificity for a given confusion matrix.

Exercise 18.70 (SensitivitySpecificityCalculation2)**.** Calculate the sensitivity and specificity for a given confusion matrix.

18.1.5.4. Bias and Variance

- Video: Machine Learning Fundamentals: Bias and Variance

Exercise 18.71 (BiasAndVariance)**.** What are bias and variance?

18.1.5.5. Mutual Information

- Video: Mutual Information, Clearly Explained

Exercise 18.72 (MutualInformationExample)**.** Provide an example and calculate if mutual information is high or low.

18.1.6. June, 25th 2024

18.1.6.1. Principal Component Analysis (PCA)

- Video: Principal Component Analysis (PCA), Step-by-Step

Exercise 18.73 (WhatIsPCA)**.** What is PCA?

Exercise 18.74 (ScreePlotExplanation)**.** What is a scree plot?

- Vidoe: PCA - Practical Tips

Exercise 18.75 (LeastSquaresInPCA)**.** Does PCA use least squares?

Exercise 18.76 (PCASteps)**.** Which steps are performed by PCA?

Exercise 18.77 (EigenvaluePC1)**.** What is the eigenvalue of the first principal component?

Exercise 18.78 (DifferencesBetweenPoints)**.** Are the differences between red and yellow the same as the differences between red and blue points?

- Video: PCA in Python

Exercise 18.79 (ScalingInPCA)**.** How to scale data in PCA?

Exercise 18.80 (DetermineNumberOfComponents)**.** How to determine the number of principal components?

Exercise 18.81 (LimitingNumberOfComponents)**.** How is the number of principal components limited?

18.1.7. t-SNE

- Video: t-SNE, Clearly Explained

Exercise 18.82 (WhyUseTSNE)**.** Why use t-SNE?

Exercise 18.83 (MainIdeaOfTSNE)**.** What is the main idea of t-SNE?

Exercise 18.84 (BasicConceptOfTSNE)**.** What is the basic concept of t-SNE?

Exercise 18.85 (TSNESteps)**.** What are the steps in t-SNE?

18.1.8. K-means clustering

- Video: K-means clustering

Exercise 18.86 (HowKMeansWorks)**.** How does K-means clustering work?

Exercise 18.87 (QualityOfClusters)**.** How can the quality of the resulting clusters be calculated?

Exercise 18.88 (IncreasingK)**.** Why is it not a good idea to increase k too much?

18.1.9. DBSCAN

- Video: Clustering with DBSCAN, Clearly Explained!!!

Exercise 18.89 (CorePointInDBSCAN)**.** What is a core point in DBSCAN?

Exercise 18.90 (AddingVsExtending)**.** What is the difference between adding and extending in DBSCAN?

Exercise 18.91 (OutliersInDBSCAN)**.** What are outliers in DBSCAN?

18.1.10. K-nearest neighbors

- Video: StatQuest: K-nearest neighbors, Clearly Explained

Exercise 18.92 (AdvantagesAndDisadvantagesOfK)**.** What are the advantages and disadvantages of k = 1 and k = 100 in K-nearest neighbors?

18.1.11. Naive Bayes

- Video: Naive Bayes, Clearly Explained!!!

Exercise 18.93 (NaiveBayesFormula)**.** What is the formula for Naive Bayes?

Exercise 18.94 (CalculateProbabilities)**.** Calculate the probabilities for a given example using Naive Bayes.

18.1.12. Gaussian Naive Bayes

- Video: Gaussian Naive Bayes, Clearly Explained!!!

Exercise 18.95 (UnderflowProblem)**.** Why is underflow a problem in Gaussian Naive Bayes?

18.1.13. July, 2nd 2024

18.1.13.1. Decision and Classification Trees, Clearly Explained

18.1.13.2. StatQuest: Decision Trees, Part 2 - Feature Selection and Missing Data

18.1.13.3. Regression Trees, Clearly Explained!!!

18.1.13.4. How to Prune Regression Trees, Clearly Explained!!!

18.1.13.5. Trees

Exercise 18.96 (Tree Usage)**.** For what can we use trees?

18.1.13.6. Decision Trees

Exercise 18.97 (Tree Usage)**.** Based on a shown tree graph:

- How can you use this tree?
- What is the root node?
- What are branches and internal nodes?
- What are the leafs?
- Are the leafs pure or impure?
- Which of the leafs is more impure?

Exercise 18.98 (Tree Feature Importance)**.** Is the most or least important feature on top?

Exercise 18.99 (Tree Feature Imputation)**.** How can you fill a gap/missing data?

Solution 18.1 (Tree Feature Imputation)*.*

- Mean
- Median
- Comparing to column with high correlation

18.1.13.7. Regression Trees

Exercise 18.100 (Regression Tree Limitations)**.** What are limitations?

Exercise 18.101 (Regression Tree Score)**.** How is the tree score calculated?

Exercise 18.102 (Regression Tree Alpha Value Small)**.** What can we say about the tree if the alpha value is small?

Exercise 18.103 (Regression Tree Increase Alpha Value)**.** What happens if you increase alpha?

Exercise 18.104 (Regression Tree Pruning)**.** What is the meaning of pruning?

18.1.14. Additional Videos

- Odds and Log(Odds), Clearly Explained!!!
- One-Hot, Label, Target and K-Fold Target Encoding, Clearly Explained!!!
- Maximum Likelihood for the Exponential Distribution, Clearly Explained!!!
- ROC and AUC, Clearly Explained!
- Entropy (for data science) Clearly Explained!!!
- Classification Trees in Python from Start to Finish: Long live video!

18.2. Introduction to Statistical Learning

> **Note**
>
> Parts of this course are based on the book **An Introduction to Statistical Learning**, James et al. (2014). Some of the figures in this presentation are taken from **An Introduction to Statistical Learning** (Springer, 2013) with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.

18.2.1. Opening Remarks and Examples

- Artificial Intelligence (AI)
- Machine learning (ML)
- Deep Learning (DL)

Figure 18.1.: AI, ML, and DL. Taken fron Chollet and Allaire (2018)

- 1980's neural networks.
- Statistical learning.
- IBM Watson supercomputer.

Statistical learning problems include:

1. Identification of prostate cancer through PSA and other measurements such as age, Gleason score, etc. Scatter plots help reveal the nature of the data and its correlations. Using transformed data (log scale) can highlight typos in the data; for example, a patient with a 449-gram prostate. Recommendation: Always examine the data before conducting any sophisticated analysis.
2. Classification of phonemes, specifically between "aa" and "ao."
3. Prediction of heart attacks, which can be visualized through colored scatter plots.
4. Detection of email spam, based on the frequency of words within the messages, using 57 features.
5. Identification of numbers in handwritten zip codes, which involves pattern recognition.
6. Classification of tissue samples into cancer classes based on gene expression profiles, utilizing heat maps for visualization.

7. Establishing the relationship between salary and demographic variables like income (wage) versus age, year, and education level, employing regression models.
8. Classification of pixels in LANDSAT images by their usage, using nearest neighbor methods.

18.2.1.1. Supervised and Unsupervised Learning

Two important types: supervised and unsupervised learning. There is even more, e.g., semi-supervised learning.

18.2.1.1.1. Starting point

- Outcome measurement Y (dependent variable, response, target).
- Vector of p predictor measurements X (inputs, regressors, covariates, features, independent variables).
- Training data $(x_1, y1), \dots, (x_N, y_N)$. These are observations (examples, instances) of these measurements.

In the *regression* problem, Y is quantitative (e.g., price, blood pressure). In the *classification* problem, Y takes values in a finite, unordered set (e.g., survived/died, digit 0-9, cancer class of tissue sample).

18.2.1.1.2. Philosophy

It is important to understand the ideas behind the various techniques, in order to know how and when to use them. One has to understand the simpler methods first, in order to grasp the more sophisticated ones. It is important to accurately assess the performance of a method, to know how well or how badly it is working (simpler methods often perform as well as fancier ones!) This is an exciting research area, having important applications in science, industry and finance. Statistical learning is a fundamental ingredient in the training of a modern data scientist.

18.3. Basics

18.3.1. Histograms

Creating a histogram and calculating the probabilities from a dataset can be approached with scientific precision

1. Data Collection: Obtain the dataset you wish to analyze. This dataset could represent any quantitative measure, such to examine its distribution.

2. Decide on the Number of Bins: The number of bins influences the histogram's granularity. There are several statistical rules to determine an optimal number of bins:

   - Square-root rule: suggests using the square root of the number of data points as the number of bins.
   - Sturges' formula: $k = 1 + 3.322 \log_{10}(n)$, where n is the number of data points and k is the suggested number of bins.
   - Freedman-Diaconis rule: uses the interquartile range (IQR) and the cube root of the number of data points n to calculate bin width as $2\frac{IQR}{n^{1/3}}$.

3. Determine Range and Bin Width: Calculate the range of data by subtracting the minimum data point value from the maximum. Divide this range by the number of bins to determine the width of each bin.

4. Allocate Data Points to Bins: Iterate through the data, sorting each data point into the appropriate bin based on its value.

5. Draw the Histogram: Use a histogram to visualize the frequency or relative frequency (probability) of data points within each bin.

6. Calculate Probabilities: The relative frequency of data within each bin represents the probability of a randomly selected data point falling within that bin's range.

Below is a Python script that demonstrates how to generate a histogram and compute probabilities using the `matplotlib` library for visualization and `numpy` for data manipulation.

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

%matplotlib inline

# Sample data: Randomly generated for demonstration
data = np.random.normal(0, 1, 1000)  # 1000 data points with a normal distribution

# Step 2: Decide on the number of bins
num_bins = int(np.ceil(1 + 3.322 * np.log10(len(data))))  # Sturges' formula

# Step 3: Determine range and bin width -- handled internally by matplotlib

# Steps 4 & 5: Sort data into bins and draw the histogram
fig, ax = plt.subplots()
n, bins, patches = ax.hist(data, bins=num_bins, density=True, alpha=0.75, edgecolor='black')

# Calculate probabilities (relative frequencies) manually, if needed
```

```
bin_width = np.diff(bins)  # np.diff finds the difference between adjacent bin boundar
probabilities = n * bin_width  # n is already normalized to form a probability density

# Adding labels and title for clarity
ax.set_xlabel('Data Value')
ax.set_ylabel('Probability Density')
ax.set_title('Histogram with Probability Density')
```

```
Text(0.5, 1.0, 'Histogram with Probability Density')
```

(a) Histogram with Probability Density

(b)

Figure 18.2.

```
for i, prob in enumerate(probabilities):
    print(f"Bin {i+1} Probability: {prob:.4f}")

# Ensure probabilities sum to 1 (or very close, due to floating-point arithmetic)
print(f"Sum of probabilities: {np.sum(probabilities)}")
```

```
Bin 1 Probability: 0.0060
Bin 2 Probability: 0.0320
Bin 3 Probability: 0.0730
Bin 4 Probability: 0.1470
Bin 5 Probability: 0.2400
Bin 6 Probability: 0.2350
Bin 7 Probability: 0.1500
Bin 8 Probability: 0.0860
Bin 9 Probability: 0.0250
Bin 10 Probability: 0.0050
Bin 11 Probability: 0.0010
Sum of probabilities: 1.0
```

This code segment goes through the necessary steps to generate a histogram and calculate probabilities for a synthetic dataset. It demonstrates important scientific and computational practices including binning, visualization, and probability calculation in Python.

Key Points: - The histogram represents the distribution of data, with the histogram's bins outlining the data's spread and density. - The option `density=True` in `ax.hist()` normalizes the histogram so that the total area under the histogram sums to 1, thereby converting frequencies to probability densities. - The choice of bin number and width has a significant influence on the histogram's shape and the insights that can be drawn from it, highlighting the importance of selecting appropriate binning strategies based on the dataset's characteristics and the analysis objectives.

18.3.2. Probability Distributions

What happens when we use smaller bins in a histogram? The histogram becomes more detailed, revealing the distribution of data points with greater precision. However, as the bin size decreases, the number of data points within each bin may decrease, leading to sparse or empty bins. This sparsity can make it challenging to estimate probabilities accurately, especially for data points that fall within these empty bins.

Advantages, when using a probability distribution, include:

- Blanks can be filled
- Probabilities can be calculated
- Parameters are sufficiemnt to describe the distribution, e.g., mean and variance for the normal distribution

Probability distributions offer a powerful solution to the challenges posed by limited data in estimating probabilities. When data is scarce, constructing a histogram to determine the probability of certain outcomes can lead to inaccurate or unreliable results due to the lack of detail in the dataset. However, collecting vast amounts

of data to populate a histogram for more precise estimates can often be impractical, time-consuming, and expensive.

A probability distribution is a mathematical function that provides the probabilities of occurrence of different possible outcomes for an experiment. It is a more efficient approach to understanding the likelihood of various outcomes than relying solely on extensive data collection. For continuous data, this is often represented graphically by a smooth curve.

18.3.2.1. The Normal Distribution: A Common Example

A commonly encountered probability distribution is the normal distribution, known for its characteristic bell-shaped curve. This curve represents how the values of a variable are distributed: most of the observations cluster around the mean (or center) of the distribution, with frequencies gradually decreasing as values move away from the mean.

The normal distribution is particularly useful because of its defined mathematical properties. It is determined entirely by its mean (mu, μ) and its standard deviation (sigma, σ). The area under the curve represents probability, making it possible to calculate the likelihood of a random variable falling within a specific range.

18.3.2.2. Practical Example: Estimating Probabilities

Consider we are interested in the heights of adults in a population. Instead of measuring the height of every adult (which would be impractical), we can use the normal distribution to estimate the probability of adults' heights falling within certain intervals, assuming we know the mean and standard deviation of the heights.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
mu = 170  # e.g., mu height of adults in cm
sd = 10  # e.g., standard deviation of heights in cm
heights = np.linspace(mu - 3*sd, mu + 3*sd, 1000)
# Calculate the probability density function for the normal distribution
pdf = norm.pdf(heights, mu, sd)
# Plot the normal distribution curve
plt.plot(heights, pdf, color='blue', linewidth=2)
plt.fill_between(heights, pdf, where=(heights >= mu - 2 * sd) & (heights <= mu + 2*sd)
plt.xlabel('Height (cm)')
plt.ylabel('Probability Density')
plt.show()
```

Figure 18.3.: Normal Distribution Curve with Highlighted Probability Area. 95 percent of the data falls within two standard deviations of the mean.

This Python code snippet generates a plot of the normal distribution for adult heights, with a mean of 170 cm and a standard deviation of 10 cm. It visually approximates a histogram with a blue bell-shaped curve, and highlights (in grey) the area under the curve between $\mu \pm 2 \times \sigma$. This area corresponds to the probability of randomly selecting an individual whose height falls within this range.

By using the area under the curve, we can efficiently estimate probabilities without needing to collect and analyze a vast amount of data. This method not only saves time and resources but also provides a clear and intuitive way to understand and communicate statistical probabilities.

18.3.3. Discrete Distributions

Discrete probability distributions are essential tools in statistics, providing a mathematical foundation to model and analyze situations with discrete outcomes. Histograms, which can be seen as discrete distributions with data organized into bins, offer a way to visualize and estimate probabilities based on the collected data. However, they come with limitations, especially when data is scarce or when we encounter gaps in the data (blank spaces in histograms). These gaps can make it challenging to accurately estimate probabilities.

A more efficient approach, especially for discrete data, is to use mathematical equations—particularly those defining discrete probability distributions—to calculate probabilities directly, thus bypassing the intricacies of data collection and histogram interpretation.

18.3.3.1. Bernoulli Distribution

The Bernoulli distribution, named after Swiss scientist Jacob Bernoulli, is a discrete probability distribution, which takes value 1 with success probability p and value 0 with failure probability $q = 1 - p$. So if X is a random variable with this distribution, we have:

$$P(X = 1) = 1 - P(X = 0) = p = 1 - q.$$

18.3.3.2. Binomial Distribution

The Binomial Distribution is a prime example of a discrete probability distribution that is particularly useful for binary outcomes (e.g., success/failure, yes/no, pumpkin pie/blueberry pie). It leverages simple mathematical principles to calculate the probability of observing a specific number of successes (preferred outcomes) in a fixed number of trials, given the probability of success in each trial.

18.3.3.3. An Illustrative Example: Pie Preference

Consider a scenario from "StatLand" where 70% of people prefer pumpkin pie over blueberry pie. The question is: What is the probability that, out of three people asked, the first two prefer pumpkin pie and the third prefers blueberry pie?

Using the concept of the Binomial Distribution, the probability of such an outcome can be calculated without the need to layout every possible combination by hand. This process not only simplifies calculations but also provides a clear and precise method to determine probabilities in scenarios involving discrete choices. We will use Python to calculate the probability of observing exactly two out of three people prefer pumpkin pie, given the 70% preference rate:

```
from scipy.stats import binom
n = 3  # Number of trials (people asked)
p = 0.7  # Probability of success (preferring pumpkin pie)
x = 2  # Number of successes (people preferring pumpkin pie)
# Probability calculation using Binomial Distribution
prob = binom.pmf(x, n, p)
print(f"The probability that exactly 2 out of 3 people prefer pumpkin pie is: {prob:.3
```

```
The probability that exactly 2 out of 3 people prefer pumpkin pie is: 0.441
```

This code uses the `binom.pmf()` function from `scipy.stats` to calculate the probability mass function (PMF) of observing exactly `x` successes in `n` trials, where each trial has a success probability of `p`.

A Binomial random variable is the sum of n independent, identically distributed Bernoulli random variables, each with probability p of success. We may indicate a random variable X with Bernoulli distribution using the notation $X \sim \mathrm{Bi}(1, \theta)$. Then, the notation for the Binomial is $X \sim \mathrm{Bi}(n, \theta)$. Its probability and distribution functions are, respectively,

$$p_X(x) = \binom{n}{x}\theta^x(1-\theta)^{n-x}, \qquad F_X(x) = \Pr\{X \leq x\} = \sum_{i=0}^{x}\binom{n}{i}\theta^i(1-\theta)^{n-i}.$$

The mean of the binomial distribution is $\mathrm{E}[X] = n\theta$. The variance of the distribution is $\mathrm{Var}[X] = n\theta(1-\theta)$ (see next section).

A process consists of a sequence of n independent trials, i.e., the outcome of each trial does not depend on the outcome of previous trials. The outcome of each trial is either a success or a failure. The probability of success is denoted as p, and p is constant for each trial. Coin tossing is a classical example for this setting.

The binomial distribution is a statistical distribution giving the probability of obtaining a specified number of successes in a binomial experiment; written Binomial(n, p), where n is the number of trials, and p the probability of success in each.

Definition 18.1 (Binomial Distribution)**.** The binomial distribution with parameters n and p, where n is the number of trials, and p the probability of success in each, is

$$p(x) = \binom{n}{k}p^x(1-p)^{n-x} \qquad x = 0, 1, \ldots, n. \tag{18.17}$$

The mean μ and the variance σ^2 of the binomial distribution are

$$\mu = np \tag{18.18}$$

and

$$\sigma^2 = np(1-p). \tag{18.19}$$

Note, the Bernoulli distribution is simply Binomial(1,p).

18.4. Continuous Distributions

Our considerations regarding probability distributions, expectations, and standard deviations will be extended from discrete distributions to continuous distributions. One simple example of a continuous distribution is the uniform distribution. Continuous distributions are defined by probability density functions.

18.4.1. Distribution functions: PDFs and CDFs

The density for a continuous distribution is a measure of the relative probability of "getting a value close to x." Probability density functions f and cumulative distribution function F are related as follows.

$$f(x) = \frac{d}{dx}F(x) \tag{18.20}$$

18.4.2. Expectation (Continuous)

Definition 18.2 (Expectation (Continuous))**.**

$$\mathrm{E}(X) = \int_{-\infty}^{\infty} x f(x)\, dx \tag{18.21}$$

18.4.3. Variance and Standard Deviation (Continuous)

Definition 18.3 (Variance (Continuous))**.** Variance can be calculated with $\mathrm{E}(X)$ and

$$\mathrm{E}(X^2) = \int_{-\infty}^{\infty} x^2 f(x)\, dx \tag{18.22}$$

as

$$\mathrm{Var}(X) = \mathrm{E}(X^2) - [E(X)]^2.$$

□

Definition 18.4 (Standard Deviation (Continuous))**.** Standard deviation can be calculated as

$$\mathrm{sd}(X) = \sqrt{\mathrm{Var}(X)}.$$

□

18.4.4. Uniform Distribution

This variable is defined in the interval $[a, b]$. We write it as $X \sim U[a, b]$. Its density and cumulative distribution functions are, respectively,

$$f_X(x) = \frac{I_{[a,b]}(x)}{b-a}, \qquad F_X(x) = \frac{1}{b-a}\int_{-\infty}^{x} I_{[a,b]}(t)\mathrm{d}t = \frac{x-a}{b-a},$$

where $I_{[a,b]}(\cdot)$ is the indicator function of the interval $[a, b]$. Note that, if we set $a = 0$ and $b = 1$, we obtain $F_X(x) = x$, $x \in [0, 1]$.

A typical example is the following: the cdf of a continuous r.v. is uniformly distributed in $[0,1]$. The proof of this statement is as follows: For $u \in [0,1]$, we have

$$\begin{aligned}\Pr\{F_X(X) \leq u\} &= \Pr\{F_X^{-1}(F_X(X)) \leq F_X^{-1}(u)\} = \Pr\{X \leq F_X^{-1}(u)\} \\ &= F_X(F_X^{-1}(u)) = u.\end{aligned}$$

This means that, when X is continuous, there is a one-to-one relationship (given by the cdf) between $x \in D_X$ and $u \in [0,1]$.

The *uniform distribution* has a constant density over a specified interval, say $[a,b]$. The uniform $U(a,b)$ distribution has density

$$f(x) = \begin{cases} 1/(b-a) & \text{if } a < x < b, \\ 0 & \text{otherwise} \end{cases} \tag{18.23}$$

18.4.5. Normal Distribution

Definition 18.5 (Normal Distribution)**.** This variable is defined on the support $D_X = \mathbb{R}$ and its density function is given by

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\}.$$

The density function is identified by the pair of parameters (μ, σ^2), where $\mu \in \mathbb{R}$ is the mean (or location parameter) and $\sigma^2 > 0$ is the variance (or dispersion parameter) of X. □

The density function is symmetric around μ. The normal distribution belongs to the location-scale family distributions. This means that, if $Z \sim N(0,1)$ (read, Z has a standard normal distribution; i.e., with $\mu = 0$ and $\sigma^2 = 1$), and we consider the linear transformation $X = \mu + \sigma Z$, then $X \sim N(\mu, \sigma^2)$ (read, X has a normal distribution with mean μ and variance σ^2). This means that one can obtain the probability of any interval $(-\infty, x]$, $x \in R$ for any normal distribution (i.e., for any pair of the parameters μ and σ) once the quantiles of the standard normal distribution are known. Indeed

$$\begin{aligned}F_X(x) &= \Pr\{X \leq x\} = \Pr\left\{\frac{X-\mu}{\sigma} \leq \frac{x-\mu}{\sigma}\right\} \\ &= \Pr\left\{Z \leq \frac{x-\mu}{\sigma}\right\} = F_Z\left(\frac{x-\mu}{\sigma}\right) \qquad x \in \mathbb{R}.\end{aligned}$$

The quantiles of the standard normal distribution are available in any statistical program. The density and cumulative distribution function of the standard normal r.v.~at point x are usually denoted by the symbols $\phi(x)$ and $\Phi(x)$.

The standard normal distribution is based on the *standard normal density function*

$$\varphi(z) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z^2}{2}\right). \tag{18.24}$$

An important application of the standardization introduced in Equation 18.24 reads as follows. In case the distribution of X is approximately normal, the distribution of X^{*} is approximately standard normal. That is

$$P(X \leq b) = P(\frac{X-\mu}{\sigma} \leq \frac{b-\mu}{\sigma}) = P(X^* \leq \frac{b-\mu}{\sigma})$$

The probability $P(X \leq b)$ can be approximated by $\Phi(\frac{b-\mu}{\sigma})$, where Φ is the standard normal cumulative distribution function.

If X is a normal random variable with mean μ and variance σ^2, i.e., $X \sim N(\mu, \sigma^2)$, then

$$X = \mu + \sigma Z \text{ where } Z \sim N(0,1). \tag{18.25}$$

If $Z \sim N(0,1)$ and $X \sim N(\mu, \sigma^2)$, then

$$X = \mu + \sigma Z.$$

The probability of getting a value in a particular interval is the area under the corresponding part of the curve. Consider the density function of the normal distribution. It can be plotted using the following commands. The result is shown in Figure 18.4.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
x = np.arange(-4, 4, 0.1)
# Calculating the normal distribution's density function values for each point in x
y = norm.pdf(x, 0, 1)
plt.plot(x, y, linestyle='-', linewidth=2)
plt.title('Normal Distribution')
plt.xlabel('X')
plt.ylabel('Density')
plt.grid(True)
plt.show()
```

Figure 18.4.: Normal Distribution Density Function

The *cumulative distribution function* (CDF) describes the probability of "hitting" x or less in a given distribution. We consider the CDF function of the normal distribution. It can be plotted using the following commands. The result is shown in Figure 18.5.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

# Generating a sequence of numbers from -4 to 4 with 0.1 intervals
x = np.arange(-4, 4, 0.1)

# Calculating the cumulative distribution function value of the normal distribution for each point i
y = norm.cdf(x, 0, 1)  # mean=0, stddev=1

# Plotting the results. The equivalent of 'type="l"' in R (line plot) becomes the default plot type
plt.plot(x, y, linestyle='-', linewidth=2)
plt.title('Normal Distribution CDF')
plt.xlabel('X')
plt.ylabel('Cumulative Probability')
plt.grid(True)
```

```
plt.show()
```

Figure 18.5.: Normal Distribution Cumulative Distribution Function

18.4.6. The Mean, the Median, and the Mode

18.4.7. The Exponential Distribution

The exponential distribution is a continuous probability distribution that describes the time between events in a Poisson process, where events occur continuously and independently at a constant average rate. It is characterized by a single parameter, the rate parameter λ, which represents the average number of events per unit time.

18.4.8. Population and Estimated Parameters

18.4.9. Calculating the Mean, Variance, and Standard Deviation

18.4.10. What is a Mathematical Model?

18.4.11. Sampling from a Distribution

18.4.12. Hypothesis Testing and the Null Hypothesis

18.4.13. Alternative Hypotheses

18.4.14. p-values: What They Are and How to Interpret Them

18.4.15. How to Calculate p-values

18.4.16. p-hacking: What It Is and How to Avoid It

18.4.17. Covariance

18.4.18. Pearson's Correlation

18.4.19. Boxplots

18.4.20. R-squared

18.4.21. The Main Ideas of Fitting a Line to Data

18.4.22. Linear Regression

18.4.23. Multiple Regression

18.5. Supervised Learning

Objectives of supervised learning: On the basis of the training data we would like to:

- Accurately predict unseen test cases.
- Understand which inputs affect the outcome, and how.
- Assess the quality of our predictions and inferences.

Note: Supervised means Y is known.

Exercise 18.105.

- Do children learn supervised?
- When do you learn supervised?
- Can learning be unsupervised?

18.5.0.0.1. Unsupervised Learning

No outcome variable, just a set of predictors (features) measured on a set of samples. The objective is more fuzzy—find groups of samples that behave similarly, find features that behave similarly, find linear combinations of features with the most variation. It is difficult to know how well your are doing. Unsupervised learning different from supervised learning, but can be useful as a pre-processing step for supervised learning. Clustering and principle component analysis are important techniques.

Unsupervised: Y is unknown, there is no Y, no trainer, no teacher, but: distances between the inputs values (features). A distance (or similarity) measure is necessary.

18.5.0.0.2. Statistical Learning

We consider supervised learning first.

Figure 18.6.: Sales as a function of TV, radio and newspaper. Taken from James et al. (2014)

Sales figures from a marketing campaign, see Figure 18.6. Trend shown using regression. First seems to be stronger than the third.

Can we predict $Y =$ Sales using these three? Perhaps we can do better using a model

$$Y = Sales \approx f(X_1 = TV, X_2 = Radio, X_3 = Newspaper)$$

modeling the joint relationsship.

Here Sales is a response or target that we wish to predict. We generically refer to the response as Y. TV is a feature, or input, or predictor; we name it X_1. Likewise name Radio as X_2, and so on. We can refer to the input vector collectively as

$$X = \begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix}$$

Now we write our model as

$$Y = f(X) + \epsilon$$

where ϵ captures measurement errors and other discrepancies.

What is f good for? With a good f we can make predictions of Y at new points $X = x$. We can understand which components of $X = (X_1, X_2, ... X_p)$ are important in explaining Y, and which are irrelevant.

For example, Seniority and Years of Education have a big impact on Income, but Marital Status typically does not. Depending on the complexity of f, we may be able to understand how each component X_j of X affects Y.

18.5.1. Statistical Learning and Regression

18.5.1.1. Regression Function

Consider Figure 18.7. Is there an ideal $f(X)$? In particular, what is a good value for $f(X)$ at any selected value of X, say $X = 4$? There can be many Y values at $X = 4$. A good value is

$$f(4) = E(Y|X = 4).$$

$E(Y|X = 4)$ means **expected value** (average) of Y given $X = 4$.

The ideal $f(x) = E(Y|X = x)$ is called the **regression function**. Read: The regression function gives the conditional expectation of Y given X.

The regression function $f(x)$ is also defined for the vector X; e.g., $f(x) = f(x_1, x_2, x_3) = E(Y|X_1 = x_1, X_2 = x_2, X_3 = x_3)$.

Figure 18.7.: Scatter plot of 2000 points (population). What is a good function f? There are many function values at $X = 4$. A function can return only one value. We can take the mean from these values as a return value. Taken from James et al. (2014)

18.5.2. Optimal Predictor

The regression function is the **ideal** or **optimal predictor** of Y with regard to mean-squared prediction error: It means that $f(x) = E(Y|X = x)$ is the function that minimizes

$$E[(Y - g(X))^2|X = x]$$

over all functions g at all points $X = x$.

18.5.2.1. Residuals, Reducible and Irreducible Error

At each point X we make mistakes:

$$\epsilon = Y - f(x)$$

is the **residual**. Even if we knew $f(x)$, we would still make errors in prediction, since at each $X = x$ there is typically a distribution of possible Y values as is illustrated in Figure 18.7.

For any estimate $\hat{f}(x)$ of $f(x)$, we have

$$E\left[(Y - \hat{f}(X))^2|X = x\right] = \left[f(x) - \hat{f}(x)\right]^2 + \text{var}(\epsilon),$$

and $\left[f(x) - \hat{f}(x)\right]^2$ is the **reducible** error, because it depends on the model (changing the model f might reduce this error), and $\text{var}(\epsilon)$ is the **irreducible** error.

18.5.2.2. Local Regression (Smoothing)

Typically we have few if any data points with $X = 4$ exactly. So we cannot compute $E(Y|X = x)$! Idea: Relax the definition and let

$$\hat{f}(x) = Ave(Y|X \in N(x)),$$

where $N(x)$ is some neighborhood of x, see Figure 18.8.

Figure 18.8.: Relaxing the definition. There is no Y value at $X = 4$. Taken from James et al. (2014)

Nearest neighbor averaging can be pretty good for small p, i.e., $p \leq 4$ and large-ish N. We will discuss smoother versions, such as kernel and spline smoothing later in the course.

18.5.3. Curse of Dimensionality and Parametric Models

Local, e.g., nearest neighbor, methods can be lousy when p is large. Reason: **the curse of dimensionality**, i.e., nearest neighbors tend to be far away in high dimensions. We need to get a reasonable fraction of the N values of y_i to average to bring the variance down—e.g., 10%. A 10% neighborhood in high dimensions need no longer be local, so we lose the spirit of estimating $E(Y|X = x)$ by local averaging, see Figure 18.9. If the curse of dimensionality does not exist, nearest neighbor models would be perfect prediction models.

We will use structured (parametric) models to deal with the curse of dimensionality. The linear model is an important example of a parametric model:

$$f_L(X) = \beta_0 + \beta_1 X_1 + ... + \beta_p X_p.$$

A linear model is specified in terms of $p + 1$ parameters $ _1, _2, ..., _p$. We estimate the parameters by fitting the model to *training data*. Although it is almost never correct, a linear model often serves as a good and interpretable approximation to the unknown true function $f(X)$.

Figure 18.9.: A 10% neighborhood in high dimensions need no longer be local. Left: Values of two variables x_1 and x_2, uniformly distributed. Form two 10% neighborhoods: (a) the first is just involving x_1 ignoring x_2. (b) is the neighborhood in two dimension. Notice that the radius of the circle is much larger than the lenght of the interval in one dimension. Right: radius plotted against fraction of the volume. In 10 dim, you have to break out the interval $[-1; +1]$ to get 10% of the data. Taken from James et al. (2014)

The linear model is avoiding the curse of dimensionality, because it is not relying on any local properties. Linear models belong to the class of *model-based* approaches: they replace the problem of estimating f with estimating a fixed set of coefficients β_i, with $i = 1, 2, \ldots, p$.

Figure 18.10.: A linear model $\hat{f}_L$ gives a reasonable fit. Taken from James et al. (2014)

Figure 18.11.: A quadratic model $\hat{f}_Q$ fits slightly better. Taken from James et al. (2014)

A linear model

$$\hat{f}_L(X) = \hat{\beta}_0 + \hat{\beta}_1 X$$

gives a reasonable fit, see Figure 18.10. A quadratic model

$$\hat{f}_Q(X) = \hat{\beta}_0 + \hat{\beta}_1 X + \hat{\beta}_2 X^2$$

gives a slightly improved fit, see Figure 18.11.

Figure 18.12 shows a simulated example. Red points are simulated values for income from the model

$$income = f(education, seniority) + \epsilon$$

Figure 18.12.: The true model. Red points are simulated values for income from the model, f is the blue surface. Taken from James et al. (2014)

Figure 18.13.: Linear regression fit to the simulated data (red points). Taken from James et al. (2014)

f is the blue surface.

The linear regression model

$$\hat{f}(education, seniority) = \hat{\beta}_0 + \hat{\beta}_1 \times education + \hat{\beta}_2 \times seniority$$

captures the important information. But it does not capture everything. More flexible regression model

$$\hat{f}_S(education, seniority)$$

fit to the simulated data. Here we use a technique called a **thin-plate spline** to fit a flexible surface. Even more flexible spline regression model

$$\hat{f}_S(education, seniority)$$

fit to the simulated data. Here the fitted model makes no errors on the training data! Also known as **overfitting**.

Figure 18.14.: Thin-plate spline models $\hat{f}_S(education, seniority)$ fitted to the model from Figure 18.12. Taken from James et al. (2014)

18.5.3.1. Trade-offs

- Prediction accuracy versus interpretability: Linear models are easy to interpret; thin-plate splines are not.
- Good fit versus over-fit or under-fit: How do we know when the fit is just right?

Figure 18.15.: Thin-plate spline models $\hat{f}_S(education, seniority)$ fitted to the model from Figure 18.12. The model makes no errors on the training data (overfitting). Taken from James et al. (2014)

- Parsimony (Occam's razor) versus black-box: We often prefer a simpler model involving fewer variables over a black-box predictor involving them all.

The trad-offs are visualized in Figure 18.16.

18.5.4. Assessing Model Accuracy and Bias-Variance Trade-off

Suppose we fit a model $f(x)$ to some training data $Tr = \{x_i, y_i\}_1^N$, and we wish to see how well it performs. We could compute the average squared prediction error over Tr:

$$MSE_{Tr} = Ave_{i \in Tr}[y_i - \hat{f}(x_i)]^2.$$

This may be biased toward more overfit models. Instead we should, if possible, compute it using fresh **test data** $Te == \{x_i, y_i\}_1^N$:

$$MSE_{Te} = Ave_{i \in Te}[y_i - \hat{f}(x_i)]^2.$$

The red curve, which illustrated the test error, can be estimated by holding out some data to get the test-data set.

Figure 18.16.: Interpretability versus flexibility. Flexibility corresponds with the number of model parameters. Taken from James et al. (2014)

Figure 18.17.: Black curve is truth. Red curve on right is $MSETe$, grey curve is $MSETr$. Orange, blue and green curves/squares correspond to fits of different flexibility. The dotted line represents the irreducible error, i.e., $var(\epsilon)$. Taken from James et al. (2014)

Figure 18.18.: Here, the truth is smoother. Black curve is truth. Red curve on right is $MSETe$, grey curve is $MSETr$. Orange, blue and green curves/squares correspond to fits of different flexibility. The dotted line represents the irreducible error, i.e., $var(\epsilon)$. Taken from James et al. (2014)

Figure 18.19.: Here the truth is wiggly and the noise is low, so the more flexible fits do the best. Black curve is truth. Red curve on right is $MSETe$, grey curve is $MSETr$. Orange, blue and green curves/squares correspond to fits of different flexibility. The dotted line represents the irreducible error, i.e., $var(\epsilon)$. Taken from James et al. (2014)

18.5.4.1. Bias-Variance Trade-off

Suppose we have fit a model $f(x)$ to some training data Tr, and let (x_0, y_0) be a test observation drawn from the population. If the true model is

$$Y = f(X) + \epsilon \qquad \text{with } f(x) = E(Y|X = x),$$

then

$$E\left(y_0 - \hat{f}(x_0)\right)^2 = \text{var}(\hat{f}(x_0)) + [Bias(\hat{f}(x_0))]^2 + \text{var}(\epsilon). \tag{18.26}$$

Here, $\text{var}(\epsilon)$ is the irreducible error. The reducible error consists of two components:

- $\text{var}(\hat{f}(x_0))$ is the variance that comes from different training sets. Different training sets result in different functions $\hat{f}$.
- $Bias(\hat{f}(x_0)) = E[\hat{f}(x_0)] - f(x_0)$.

The expectation averages over the variability of y_0 as well as the variability in Tr. Note that

$$Bias(\hat{f}(x_0)) = E[\hat{f}(x_0)] - f(x_0).$$

Typically as the flexibility of $\hat{f}$ increases, its variance increases (because the fits differ from training set to trainig set), and its bias decreases. So choosing the flexibility based on average test error amounts to a bias-variance trade-off, see Figure 18.20.

Figure 18.20.: Bias-variance trade-off for the three examples. Taken from James et al. (2014)

If we add the two components (reducible and irreducible error), we get the MSE in Figure 18.20 as can be seen in Equation 18.26.

18.5.5. Classification Problems and K-Nearest Neighbors

In classification we have a qualitative response variable.

Figure 18.21.: Classification. Taken from James et al. (2014)

Here the response variable Y is qualitative, e.g., email is one of $C = (spam, ham)$, where ham is good email, digit class is one of $C = \{0, 1, \dots, 9\}$. Our goals are to:

- Build a classifier $C(X)$ that assigns a class label from C to a future unlabeled observation X.
- Assess the uncertainty in each classification
- Understand the roles of the different predictors among $X = (X_1, X_2, \dots, X_p)$.

Simulation example depicted in@fig-0218a. Y takes two values, zero and one, and X has only one value. Big sample: each single vertical bar indicates an occurrance of a zero (orange) or one (blue) as a function of the Xs. Black curve generated the data: it is the probability of generating a one. For high values of X, the probability of ones is increasing. What is an ideal classifier $C(X)$?

Suppose the K elements in C are numbered $1, 2, \dots, K$. Let

$$p_k(x) = Pr(Y = k|X = x), k = 1, 2, \dots, K.$$

These are the **conditional class probabilities** at x; e.g. see little barplot at $x = 5$. Then the **Bayes optimal classifier** at x is

$$C(x) = j \qquad \text{if } p_j(x) = \max\{p_1(x), p_2(x), \dots, p_K(x)\}.$$

At $x = 5$ there is an 80% probability of one, and an 20% probability of a zero. So, we classify this point to the class with the highest probability, the majority class.

Nearest-neighbor averaging can be used as before. This is illustrated in Fig.~**??**. Here, we consider 100 points only. Nearest-neighbor averaging also breaks down as dimension grows. However, the impact on $\hat{C}(x)$ is less than on $\hat{p}_k(x)$, $k = 1, \dots, K$.

Figure 18.22.: Classification. Taken from James et al. (2014)

18.5.5.1. Classification: Some Details

Average number of errors made to measure the performance. Typically we measure the performance of $\hat{C}(x)$ using the **misclassification error rate**:

$$Err_{Te} = Ave_{i \in Te} I[y_i \neq \hat{C}(x_i)].$$

The Bayes classifier (using the true $p_k(x)$) has smallest error (in the population).

18.5.6. k-Nearest Neighbor Classification

Consider k-nearest neighbors in two dimensions. Orange and blue dots label the true class memberships of the underlying points in the 2-dim plane. Dotted line is the decision boundary, that is the contour with equal probability for both classes.

Nearest-neighbor averaging in 2-dim. At any given point we want to classify, we spread out a little neighborhood, say $K = 10$ points from the neighborhood and calulated the percentage of blue and orange. We assign the color with the highest probability to this point. If this is done for every point in the plane, we obtain the solid black curve as the esitmated decsion boundary.

We can use $K = 1$. This is the **nearest-neighbor classifier**. The decision boundary is piecewise linear. Islands occur. Approximation is rather noisy.

$K = 100$ leads to a smooth decision boundary. But gets uninteresting.

Figure 18.23.: K-nearest neighbors in two dimensions. Taken from James et al. (2014)

K large means higher bias, so $1/K$ is chosen, because we go from low to high complexity on the x-error, see Figure 18.26. Horizontal dotted line is the base error.

18.5.7. Minkowski Distance

The Minkowski distance of order p (where p is an integer) between two points $X = (x_1, x_2, \dots, x_n)$ and $Y = (y_1, y_2, \dots, y_n) \in \mathbb{R}^n$ is defined as:

$$D(X, Y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}.$$

18.5.8. Unsuperivsed Learning: Classification

18.5.8.1. k-Means Algorithm

The k-means algorithm is an unsupervised learning algorithm that has a loose relationship to the k-nearest neighbor classifier. The k-means algorithm works as follows:

Figure 18.24.: K-nearest neighbors in two dimensions. Taken from James et al. (2014)

Figure 18.25.: K-nearest neighbors in two dimensions. Taken from James et al. (2014)

Figure 18.26.: K-nearest neighbors classification error. Taken from James et al. (2014)

- Step 1: Randomly choose k centers. Assign points to cluster.
- Step 2: Determine the distances of each data point to the centroids and re-assign each point to the closest cluster centroid based upon minimum distance
- Step 3: Calculate cluster centroids again
- Step 4: Repeat steps 2 and 3 until we reach global optima where no improvements are possible and no switching of data points from one cluster to other.

The basic principle of the k-means algorithm is illustrated in Figure 18.27, Figure 18.28, Figure 18.29, and Figure 18.30.

Figure 18.27.: k-means algorithm. Step 1. Randomly choose k centers. Assign points to cluster. k initial 'means'(in this case $k = 3$) are randomly generated within the data domain (shown in color). Attribution: I, Weston.pace, CC BY-SA 3.0 http://creativecommons.org/licenses/by-sa/3.0/, via Wikimedia Commons

Figure 18.28.: k-means algorithm. Step 2. k clusters are created by associating every observation with the nearest mean. The partitions here represent the Voronoi diagram generated by the means. Attribution: I, Weston.pace, CC BY-SA 3.0 http://creativecommons.org/licenses/by-sa/3.0/, via Wikimedia Commons

Figure 18.29.: k-means algorithm. Step 3. The centroid of each of the k clusters becomes the new mean. Attribution: I, Weston.pace, CC BY-SA 3.0 http://creativecommons.org/licenses/by-sa/3.0/, via Wikimedia Commons

Figure 18.30.: k-means algorithm. Step 4. Steps 2 and 3 are repeated until convergence has been reached. Attribution: I, Weston.pace, CC BY-SA 3.0 http://creativecommons.org/licenses/by-sa/3.0/, via Wikimedia Commons

Part IV.

Machine Learning and AI

19. Machine Learning and Artificial Intelligence

19.1. Jupyter Notebooks

- The Jupyter-Notebook version of this file can be found here: malai.ipynb

19.2. Videos

19.2.1. June, 11th 2024

- Happy Halloween (Neural Networks Are Not Scary)
- The Essential Main Ideas of Neural Networks

19.2.2. June, 18th 2024

- The Chain Rule
- Gradient Descent, Step-by-Step
- Neural Networks Pt. 2: Backpropagation Main Ideas

19.2.2.1. Gradient Descent

Exercise 19.1 (GradDescStepSize)**.** How is the step size calculated?

Exercise 19.2 (GradDescIntercept)**.** How to calculate the new intercept?

Exercise 19.3 (GradDescIntercept)**.** When does the gradient descend stop?

19.2.2.2. Backpropagation

Exercise 19.4 (ChainRuleAndGradientDescent)**.** What are the key components involved in backpropagation?

Exercise 19.5 (BackpropagationNaming)**.** Why is it called backpropagation?

19.2.2.3. ReLU

Exercise 19.6 (Graph ReLU)**.** Draw the graph of a ReLU function.

- Backpropagation Details Pt. 1: Optimizing 3 parameters simultaneously.
- Backpropagation Details Pt. 2: Going bonkers with The Chain Rule
- Neural Networks Pt. 3: ReLU In Action!!!
- Neural Networks Pt. 4: Multiple Inputs and Outputs
- Neural Networks Part 5: ArgMax and SoftMax
- Tensors for Neural Networks, Clearly Explained!!!
- Essential Matrix Algebra for Neural Networks, Clearly Explained!!!
- The StatQuest Introduction to PyTorch

19.2.2.4. PyTorch Links

- StatQuest: Introduction to Coding Neural Networks with PyTorch
- ML-AI Pytorch Introduction

19.2.3. June, 25th 2024

19.2.4. CNNs

19.2.4.1. Neural Networks Part 8: Image Classification with Convolutional Neural Networks (CNNs)

Exercise 19.7 (CNNImageRecognition)**.** Why are classical neural networks poor at image recognition?

Exercise 19.8 (CNNFiltersInitialization)**.** How are the filter values in CNNs initialized and optimized?

Exercise 19.9 (CNNFilterInitialization)**.** How are the filter values determined in Convolutional Neural Networks (CNNs)?

Exercise 19.10 (GenNNStockPrediction)**.** What is a limitation of using classical neural networks for stock market prediction?

19.2.5. RNN

19.2.5.1. Recurrent Neural Networks (RNNs), Clearly Explained!!!

Exercise 19.11 (RNNUnrolling)**.** How does the unrolling process work in Recurrent Neural Networks (RNNs)?

Exercise 19.12 (RNNReliability)**.** Why do Recurrent Neural Networks (RNNs) sometimes fail to work reliably?

19.2.6. LSTM

19.2.6.1. Long Short-Term Memory (LSTM), Clearly Explained

Exercise 19.13 (LSTMSigmoidTanh)**.** What are the differences between the sigmoid and tanh activation functions?

Exercise 19.14 (LSTMSigmoidTanh)**.** What is the ?

Exercise 19.15 (LSTMGates)**.** What are the gates in an LSTM network and their functions?

Exercise 19.16 (LSTMLongTermInfo)**.** In which gate is long-term information used in an LSTM network?

Exercise 19.17 (LSTMUpdateGates)**.** In which Gates is it updated in an LSTM?

19.2.7. Pytorch/Lightning

19.2.7.1. Introduction to Coding Neural Networks with PyTorch and Lightning

Exercise 19.18 (PyTorchRequiresGrad)**.** What does `requires_grad` mean in PyTorch?

19.2.8. July, 2nd 2024

- Word Embedding and Word2Vec, Clearly Explained!!!
- Sequence-to-Sequence (seq2seq) Encoder-Decoder Neural Networks, Clearly Explained!!!
- Attention for Neural Networks, Clearly Explained!!!

19.2.8.1. Embeddings

Exercise 19.19 (NN Strings)**.** Can neural networks process strings?

Exercise 19.20 (Embedding Definition)**.** What is the meaning of word embedding?

Exercise 19.21 (Embedding Dimensions)**.** Why do we need high dimension in word embedding?

19.2.8.2. Sequence to Sequence

Exercise 19.22 (LSTM)**.** Why are LSTMs used?

Exercise 19.23 (Teacher Forcing)**.** Why is teacher forcing used?

Exercise 19.24 (Attention)**.** What is the idea of attention?

19.2.9. Additional Lecture (July, 9th 2024)?

- Transformer Neural Networks, ChatGPT's foundation, Clearly Explained!!!
- Decoder-Only Transformers, ChatGPTs specific Transformer, Clearly Explained!!!
- The matrix math behind transformer neural networks, one step at a time!!!
- Word Embedding in PyTorch + Lightning

19.2.9.1. Transformers

Exercise 19.25 (ChatGPT)**.** What kind of transformer does ChatGPT use?

Exercise 19.26 (Translation)**.** What kind of NN are used for translation?

Exercise 19.27 (Difference Encoder-Decoder and Decoder Only.)**.** What is the encoder-decoder transformer and the decoder only transformer?

Exercise 19.28 (Weights)**.** How are the weights initialized (a) and trained (b)?

Exercise 19.29 (Order of Words)**.** How is the word order preserved?

Exercise 19.30 (Relationship Between Words)**.** How is the relationship between words modeled?

Exercise 19.31 (Masked Self Attention)**.** What is masked self-attention?

Exercise 19.32 (Softmax)**.** Why is Softmax used to calculate percentage of similarities?

Exercise 19.33 (Softmax Output)**.** How is the percentage output of softmax in Transformers used?

Exercise 19.34 (V´s)**.** What is done with the scaled V´s that we get for each token so far (example: "is","what")?

Exercise 19.35 (Residual Connections)**.** What are residual connections?

Exercise 19.36 (Generate Known Word in Sequence)**.** Why do we want to generate the word in the sequence that comes after "what" that we already know? (Example from video)

Exercise 19.37 (Masked-Self-Attention Values and Bypass)**.** How do we use the two values ("masked-self-attention values + bypass") which we have for each input? (Example from video: ("What", "is", "StatQuest"))

19.2.10. Additional Videos

- The SoftMax Derivative, Step-by-Step!!!
- Neural Networks Part 6: Cross Entropy
- Neural Networks Part 7: Cross Entropy Derivatives and Backpropagation

19.2.11. All Videos in a Playlist

- Full Playlist ML-AI

19.3. The StatQuest Introduction to PyTorch

The following code is taken from The StatQuest Introduction to PyTorch. Attribution goes to Josh Starmer, the creator of StatQuest, see Josh Starmer.

```
import torch # torch provides basic functions, from setting a random seed (for reprodu
import torch.nn as nn # torch.nn allows us to create a neural network.
import torch.nn.functional as F # nn.functional give us access to the activation and l
from torch.optim import SGD # optim contains many optimizers. Here, we're using SGD, s

import matplotlib.pyplot as plt ## matplotlib allows us to draw graphs.
import seaborn as sns ## seaborn makes it easier to draw nice-looking graphs.

%matplotlib inline
```

Building a neural network in PyTorch means creating a new class with two methods: init() and forward(). The init() method defines and initializes all of the parameters that we want to use, and the forward() method tells PyTorch what should happen during a forward pass through the neural network.

19.3.1. Build a Simple Neural Network in PyTorch

`__init__()` is the class constructor function, and we use it to initialize the weights and biases.

```
## create a neural network class by creating a class that inherits from nn.Module.
class BasicNN(nn.Module):

    def __init__(self): # __init__() is the class constructor function, and we use it

        super().__init__() # initialize an instance of the parent class, nn.Model.

        ## Now create the weights and biases that we need for our neural network.
        ## Each weight or bias is an nn.Parameter, which gives us the option to optimi
        ## requires_grad, which is short for "requires gradient", to True. Since we do
        ## parameters now, we set requires_grad=False.
        ##
        ## NOTE: Because our neural network is already fit to the data, we will input
        ## for each weight and bias. In contrast, if we had not already fit the neural
        ## we might start with a random initalization of the weights and biases.
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
        self.b00 = nn.Parameter(torch.tensor(-0.85), requires_grad=False)
        self.w01 = nn.Parameter(torch.tensor(-40.8), requires_grad=False)
```

```
        self.w10 = nn.Parameter(torch.tensor(12.6), requires_grad=False)
        self.b10 = nn.Parameter(torch.tensor(0.0), requires_grad=False)
        self.w11 = nn.Parameter(torch.tensor(2.7), requires_grad=False)

        self.final_bias = nn.Parameter(torch.tensor(-16.), requires_grad=False)


    def forward(self, input): ## forward() takes an input value and runs it though the neural networ
                              ## illustrated at the top of this notebook.

        ## the next three lines implement the top of the neural network (using the top node in the h
        input_to_top_relu = input * self.w00 + self.b00
        top_relu_output = F.relu(input_to_top_relu)
        scaled_top_relu_output = top_relu_output * self.w01

        ## the next three lines implement the bottom of the neural network (using the bottom node in
        input_to_bottom_relu = input * self.w10 + self.b10
        bottom_relu_output = F.relu(input_to_bottom_relu)
        scaled_bottom_relu_output = bottom_relu_output * self.w11

        ## here, we combine both the top and bottom nodes from the hidden layer with the final bias.
        input_to_final_relu = scaled_top_relu_output + scaled_bottom_relu_output + self.final_bias

        output = F.relu(input_to_final_relu)

        return output # output is the predicted effectiveness for a drug dose.
```

Once we have created the class that defines the neural network, we can create an actual neural network and print out its parameters, just to make sure things are what we expect.

```
## create the neural network.
model = BasicNN()

## print out the name and value for each parameter
for name, param in model.named_parameters():
    print(name, param.data)
```

```
w00 tensor(1.7000)
b00 tensor(-0.8500)
w01 tensor(-40.8000)
w10 tensor(12.6000)
b10 tensor(0.)
```

```
w11 tensor(2.7000)
final_bias tensor(-16.)
```

19.3.2. Use the Neural Network and Graph the Output

Now that we have a neural network, we can use it on a variety of doses to determine which will be effective. Then we can make a graph of these data, and this graph should match the green bent shape fit to the training data that's shown at the top of this document. So, let's start by making a sequence of input doses…

```
## now create the different doses we want to run through the neural network.
## torch.linspace() creates the sequence of numbers between, and including, 0 and 1.
input_doses = torch.linspace(start=0, end=1, steps=11)

# now print out the doses to make sure they are what we expect...
input_doses
```

```
tensor([0.0000, 0.1000, 0.2000, 0.3000, 0.4000, 0.5000, 0.6000, 0.7000, 0.8000,
        0.9000, 1.0000])
```

Now that we have input_doses, let's run them through the neural network and graph the output…

```
## create the neural network.
model = BasicNN()

## now run the different doses through the neural network.
output_values = model(input_doses)

## Now draw a graph that shows the effectiveness for each dose.
##
## First, set the style for seaborn so that the graph looks cool.
sns.set(style="whitegrid")

## create the graph (you might not see it at this point, but you will after we save it
sns.lineplot(x=input_doses,
     y=output_values,
     color='green',
     linewidth=2.5)

## now label the y- and x-axes.
plt.ylabel('Effectiveness')
plt.xlabel('Dose')
```

```
## optionally, save the graph as a PDF.
# plt.savefig('BasicNN.pdf')
```

```
Text(0.5, 0, 'Dose')
```

The graph shows that the neural network fits the training data. In other words, so far, we don't have any bugs in our code.

19.3.3. Optimize (Train) a Parameter in the Neural Network and Graph the Output

Now that we know how to create and use a simple neural network, and we can graph the output relative to the input, let's see how to train a neural network. The first thing we need to do is tell PyTorch which parameter (or parameters) we want to train, and we do that by setting requiresgrad=True. In this example, we'll train finalbias.

Now we create a neural network by creating a class that inherits from nn.Module.

NOTE: This code is the same as before, except we changed the class name to BasicNN_train and we modified final_bias in two ways:

```
1) we set the value of the tensor to 0, and
2) we set "requires_grad=True".
```

Now let's graph the output of BasicNN_train, which is currently not optimized, and compare it to the graph we drew earlier of the optimized neural network.

```
class BasicNN_train(nn.Module):

    def __init__(self): # __init__ is the class constructor function, and we use it to

        super().__init__() # initialize an instance of the parent class, nn.Module.

        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
        self.b00 = nn.Parameter(torch.tensor(-0.85), requires_grad=False)
        self.w01 = nn.Parameter(torch.tensor(-40.8), requires_grad=False)

        self.w10 = nn.Parameter(torch.tensor(12.6), requires_grad=False)
        self.b10 = nn.Parameter(torch.tensor(0.0), requires_grad=False)
        self.w11 = nn.Parameter(torch.tensor(2.7), requires_grad=False)

        ## we want to modify final_bias to demonstrate how to optimize it with backpro
        ## The optimal value for final_bias is -16...
#         self.final_bias = nn.Parameter(torch.tensor(-16.), requires_grad=False)
        ## ...so we set it to 0 and tell Pytorch that it now needs to calculate the gr
        self.final_bias = nn.Parameter(torch.tensor(0.), requires_grad=True)

    def forward(self, input):

        input_to_top_relu = input * self.w00 + self.b00
        top_relu_output = F.relu(input_to_top_relu)
        scaled_top_relu_output = top_relu_output * self.w01

        input_to_bottom_relu = input * self.w10 + self.b10
        bottom_relu_output = F.relu(input_to_bottom_relu)
        scaled_bottom_relu_output = bottom_relu_output * self.w11

        input_to_final_relu = scaled_top_relu_output + scaled_bottom_relu_output + sel

        output = F.relu(input_to_final_relu)

        return output
```

```
## create the neural network.
model = BasicNN_train()
```

```
## now run the different doses through the neural network.
output_values = model(input_doses)

## Now draw a graph that shows the effectiveness for each dose.
##
## set the style for seaborn so that the graph looks cool.
sns.set(style="whitegrid")

## create the graph (you might not see it at this point, but you will after we save it as a PDF).
sns.lineplot(x=input_doses,
             y=output_values.detach(), ## NOTE: because final_bias has a gradident, we call detach()
                                       ## to return a new tensor that only has the value and not the
             color='green',
             linewidth=2.5)

## now label the y- and x-axes.
plt.ylabel('Effectiveness')
plt.xlabel('Dose')

## lastly, save the graph as a PDF.
# plt.savefig('BasicNN_train.pdf')
```

```
Text(0.5, 0, 'Dose')
```

The graph shows that when the dose is 0.5, the output from the unoptimized neural network is 17, which is wrong, since the output value should be 1. So, now that we have a parameter we can optimize, let's create some training data that we can use to optimize it.

```
## create the training data for the neural network.
inputs = torch.tensor([0., 0.5, 1.])
labels = torch.tensor([0., 1., 0.])
```

..and now let's use that training data to train (or optimize) final_bias.

```
## create the neural network we want to train.
model = BasicNN_train()

optimizer = SGD(model.parameters(), lr=0.1) ## here we're creating an optimizer to tra
                                            ## NOTE: There are a bunch of different wa
                                            ## In this example, we'll use Stochastic (
                                            ## another popular algortihm is Adam (whic

print("Final bias, before optimization: " + str(model.final_bias.data) + "\n")

## this is the optimization loop. Each time the optimizer sees all of the training dat
for epoch in range(100):
```

```
    ## we create and initialize total_loss for each epoch so that we can evaluate how well model fit
    ## training data. At first, when the model doesn't fit the training data very well, total_loss
    ## will be large. However, as gradient descent improves the fit, total_loss will get smaller and
    ## If total_loss gets really small, we can decide that the model fits the data well enough and s
    ## optimizing the fit. Otherwise, we can just keep optimizing until we reach the maximum number
    total_loss = 0

    ## this internal loop is where the optimizer sees all of the training data and where we
    ## calculate the total_loss for all of the training data.
    for iteration in range(len(inputs)):

        input_i = inputs[iteration] ## extract a single input value (a single dose)...
        label_i = labels[iteration] ## ...and its corresponding label (the effectiveness for the dos

        output_i = model(input_i) ## calculate the neural network output for the input (the single d

        loss = (output_i - label_i)**2 ## calculate the loss for the single value.
                                       ## NOTE: Because output_i = model(input_i), "loss" has a conn
                                       ## and the derivative (calculated in the next step) is kept a
                                       ## in "model".

        loss.backward() # backward() calculates the derivative for that single value and adds it to

        total_loss += float(loss) # accumulate the total loss for this epoch.


    if (total_loss < 0.0001):
        print("Num steps: " + str(epoch))
        break

    optimizer.step() ## take a step toward the optimal value.
    optimizer.zero_grad() ## This zeroes out the gradient stored in "model".
                          ## Remember, by default, gradients are added to the previous step (the gra
                          ## and we took advantage of this process to calculate the derivative one d
                          ## NOTE: "optimizer" has access to "model" because of how it was created w
                          ## (made earlier): optimizer = SGD(model.parameters(), lr=0.1).
                          ## ALSO NOTE: Alternatively, we can zero out the gradient with model.zero_
    if epoch % 10 == 0:
        print("Step: " + str(epoch) + " Final Bias: " + str(model.final_bias.data) + "\n")
    ## now go back to the start of the loop and go through another epoch.

print("Total loss: " + str(total_loss))
print("Final bias, after optimization: " + str(model.final_bias.data))
```

```
Final bias, before optimization: tensor(0.)

Step: 0 Final Bias: tensor(-3.2020)

Step: 10 Final Bias: tensor(-14.6348)

Step: 20 Final Bias: tensor(-15.8623)

Step: 30 Final Bias: tensor(-15.9941)

Num steps: 34
Total loss: 6.58966600894928e-05
Final bias, after optimization: tensor(-16.0019)
```

So, if everything worked correctly, the optimizer should have converged on final_bias = 16.0019 after 34 steps, or epochs. BAM!

Lastly, let's graph the output from the optimized neural network and see if it's the same as what we started with. If so, then the optimization worked.

```
## run the different doses through the neural network
output_values = model(input_doses)

## set the style for seaborn so that the graph looks cool.
sns.set(style="whitegrid")

## create the graph (you might not see it at this point, but you will after we save it
sns.lineplot(x=input_doses,
     y=output_values.detach(), ## NOTE: we call detach() because final_bias has a grad
     color='green',
     linewidth=2.5)

## now label the y- and x-axes.
plt.ylabel('Effectiveness')
plt.xlabel('Dose')

## lastly, save the graph as a PDF.
# plt.savefig('BascNN_optimized.pdf')
```

```
Text(0.5, 0, 'Dose')
```

And we see that the optimized model results in the same graph that we started with, so the optimization worked as expected.

19.4. Build a Long Short-Term Memory unit by hand using PyTorch + Lightning

The following code is based on Long Short-Term Memory with PyTorch + Lightning and StatQuest: Long Short-Term Memory (LSTM) with PyTorch + Lightning!!!. Attribution goes to Josh Starmer, the creator of StatQuest, see Josh Starmer.

```
import torch # torch will allow us to create tensors.
import torch.nn as nn # torch.nn allows us to create a neural network.
import torch.nn.functional as F # nn.functional give us access to the activation and loss functions.
from torch.optim import Adam # optim contains many optimizers. This time we're using Adam

import lightning as L # lightning has tons of cool tools that make neural networks easier
from torch.utils.data import TensorDataset, DataLoader # these are needed for the training data
```

A Long Short-Term Memory (LSTM) unit is a type of neural network, and that means we need to create a new class. To make it easy to train the LSTM, this class will inherit from LightningModule and we'll create the following methods:

- `init()` to initialize the Weights and Biases and keep track of a few other house keeping things.
- `lstm_unit()` to do the LSTM math. For example, to calculate the percentage of the long-term memory to remember.
- `forward()` to make a forward pass through the unrolled LSTM. In other words forward() calls `lstm_unit()` for each data point.
- `configure_optimizers()` to configure the opimimizer. In the past, we have use SGD (Stochastic Gradient Descent), however, in this tutorial we'll change things up and use Adam, another popular algorithm for optimizing the Weights and Biases.
- `training_step()` to pass the training data to forward(), calculate the loss and to keep track of the loss values in a log file.

```
class LSTMbyHand(L.LightningModule):

    def __init__(self):
        super().__init__()
        L.seed_everything(seed=42)

        ## NOTE: nn.LSTM() uses random values from a uniform distribution to initiali
        ## Here we can do it 2 different ways 1) Normal Distribution and 2) Uniform Di
        ## We'll start with the Normal distribution.
        mean = torch.tensor(0.0)
        std = torch.tensor(1.0)

        ## NOTE: In this case, I'm only using the normal distribution for the Weights
        ## All Biases are initialized to 0.
        ##
        ## These are the Weights and Biases in the first stage, which determines what
        ## of the long-term memory the LSTM unit will remember.
        self.wlr1 = nn.Parameter(torch.normal(mean=mean, std=std), requires_grad=True)
        self.wlr2 = nn.Parameter(torch.normal(mean=mean, std=std), requires_grad=True)
        self.blr1 = nn.Parameter(torch.tensor(0.), requires_grad=True)

        ## These are the Weights and Biases in the second stage, which determines the
        ## potential long-term memory and what percentage will be remembered.
        self.wpr1 = nn.Parameter(torch.normal(mean=mean, std=std), requires_grad=True)
        self.wpr2 = nn.Parameter(torch.normal(mean=mean, std=std), requires_grad=True)
        self.bpr1 = nn.Parameter(torch.tensor(0.), requires_grad=True)

        self.wp1 = nn.Parameter(torch.normal(mean=mean, std=std), requires_grad=True)
        self.wp2 = nn.Parameter(torch.normal(mean=mean, std=std), requires_grad=True)
        self.bp1 = nn.Parameter(torch.tensor(0.), requires_grad=True)
```

```
        ## These are the Weights and Biases in the third stage, which determines the
        ## new short-term memory and what percentage will be sent to the output.
        self.wo1 = nn.Parameter(torch.normal(mean=mean, std=std), requires_grad=True)
        self.wo2 = nn.Parameter(torch.normal(mean=mean, std=std), requires_grad=True)
        self.bo1 = nn.Parameter(torch.tensor(0.), requires_grad=True)

        ## We can also initialize all Weights and Biases using a uniform distribution. This is
        ## how nn.LSTM() does it.
#         self.wlr1 = nn.Parameter(torch.rand(1), requires_grad=True)
#         self.wlr2 = nn.Parameter(torch.rand(1), requires_grad=True)
#         self.blr1 = nn.Parameter(torch.rand(1), requires_grad=True)

#         self.wpr1 = nn.Parameter(torch.rand(1), requires_grad=True)
#         self.wpr2 = nn.Parameter(torch.rand(1), requires_grad=True)
#         self.bpr1 = nn.Parameter(torch.rand(1), requires_grad=True)

#         self.wp1 = nn.Parameter(torch.rand(1), requires_grad=True)
#         self.wp2 = nn.Parameter(torch.rand(1), requires_grad=True)
#         self.bp1 = nn.Parameter(torch.rand(1), requires_grad=True)

#         self.wo1 = nn.Parameter(torch.rand(1), requires_grad=True)
#         self.wo2 = nn.Parameter(torch.rand(1), requires_grad=True)
#         self.bo1 = nn.Parameter(torch.rand(1), requires_grad=True)


    def lstm_unit(self, input_value, long_memory, short_memory):
        ## lstm_unit does the math for a single LSTM unit.

        ## NOTES:
        ## long term memory is also called "cell state"
        ## short term memory is also called "hidden state"

        ## 1) The first stage determines what percent of the current long-term memory
        ##    should be remembered
        long_remember_percent = torch.sigmoid((short_memory * self.wlr1) +
                                              (input_value * self.wlr2) +
                                              self.blr1)

        ## 2) The second stage creates a new, potential long-term memory and determines what
        ##    percentage of that to add to the current long-term memory
        potential_remember_percent = torch.sigmoid((short_memory * self.wpr1) +
                                                   (input_value * self.wpr2) +
                                                   self.bpr1)
        potential_memory = torch.tanh((short_memory * self.wp1) +
```

```
                                    (input_value * self.wp2) +
                                    self.bp1)

        ## Once we have gone through the first two stages, we can update the long-term
        updated_long_memory = ((long_memory * long_remember_percent) +
                       (potential_remember_percent * potential_memory))

        ## 3) The third stage creates a new, potential short-term memory and determine
        ##    percentage of that should be remembered and used as output.
        output_percent = torch.sigmoid((short_memory * self.wo1) +
                                     (input_value * self.wo2) +
                                     self.bo1)
        updated_short_memory = torch.tanh(updated_long_memory) * output_percent

        ## Finally, we return the updated long and short-term memories
        return([updated_long_memory, updated_short_memory])

    def forward(self, input):
        ## forward() unrolls the LSTM for the training data by calling lstm_unit() for
        ## that we have. forward() also keeps track of the long and short-term memorie
        ## the final short-term memory, which is the 'output' of the LSTM.

        long_memory = 0 # long term memory is also called "cell state" and indexed wit
        short_memory = 0 # short term memory is also called "hidden state" and indexed
        day1 = input[0]
        day2 = input[1]
        day3 = input[2]
        day4 = input[3]

        ## Day 1
        long_memory, short_memory = self.lstm_unit(day1, long_memory, short_memory)

        ## Day 2
        long_memory, short_memory = self.lstm_unit(day2, long_memory, short_memory)

        ## Day 3
        long_memory, short_memory = self.lstm_unit(day3, long_memory, short_memory)

        ## Day 4
        long_memory, short_memory = self.lstm_unit(day4, long_memory, short_memory)

        ##### Now return short_memory, which is the 'output' of the LSTM.
        return short_memory
```

```
    def configure_optimizers(self): # this configures the optimizer we want to use for backpropagati
        # return Adam(self.parameters(), lr=0.1) # NOTE: Setting the learning rate to 0.1 trains way
                                                 # using the default learning rate, lr=0.001, which
                                                 # training. However, if we use the default value, w
                                                 # the exact same Weights and Biases that I used in
                                                 # the LSTM Clearly Explained StatQuest video. So we
                                                 # default value.
        return Adam(self.parameters())


    def training_step(self, batch, batch_idx): # take a step during gradient descent.
        input_i, label_i = batch # collect input
        output_i = self.forward(input_i[0]) # run input through the neural network
        loss = (output_i - label_i)**2 ## loss = sum of squared residual
        # Logging the loss and the predicted values so we can evaluate the training:
        self.log("train_loss", loss)
        ## NOTE: Our dataset consists of two sequences of values representing Company A and Company
        ## For Company A, the goal is to predict that the value on Day 5 = 0, and for Company B,
        ## the goal is to predict that the value on Day 5 = 1. We use label_i, the value we want to
        ## predict, to keep track of which company we just made a prediction for and
        ## log that output value in a company specific file
        if (label_i == 0):
            self.log("out_0", output_i)
        else:
            self.log("out_1", output_i)
        return loss
```

Once we have created the class that defines an LSTM, we can use it to create a model and print out the randomly initialized Weights and Biases. Then, just for fun, we'll see what those random Weights and Biases predict for Company A and Company B. If they are good predictions, then we're done! However, the chances of getting good predictions from random values is very small.

```
## Create the model object, print out parameters and see how well
## the untrained LSTM can make predictions...
model = LSTMbyHand()

print("Before optimization, the parameters are...")
for name, param in model.named_parameters():
    print(name, param.data)

print("\nNow let's compare the observed and predicted values...")
## NOTE: To make predictions, we pass in the first 4 days worth of stock values
## in an array for each company. In this case, the only difference between the
```

```
## input values for Company A and B occurs on the first day. Company A has 0 and
## Company B has 1.
print("Company A: Observed = 0, Predicted =",
      model(torch.tensor([0., 0.5, 0.25, 1.])).detach())
print("Company B: Observed = 1, Predicted =",
      model(torch.tensor([1., 0.5, 0.25, 1.])).detach())
```

```
Before optimization, the parameters are...
wlr1 tensor(0.3367)
wlr2 tensor(0.1288)
blr1 tensor(0.)
wpr1 tensor(0.2345)
wpr2 tensor(0.2303)
bpr1 tensor(0.)
wp1 tensor(-1.1229)
wp2 tensor(-0.1863)
bp1 tensor(0.)
wo1 tensor(2.2082)
wo2 tensor(-0.6380)
bo1 tensor(0.)

Now let's compare the observed and predicted values...
Company A: Observed = 0, Predicted = tensor(-0.0377)
Company B: Observed = 1, Predicted = tensor(-0.0383)
```

With the unoptimized paramters, the predicted value for Company A, -0.0377, isn't terrible, since it is relatively close to the observed value, 0. However, the predicted value for Company B, -0.0383, is terrible, because it is relatively far from the observed value, 1. So, that means we need to train the LSTM.

19.4.1. Train the LSTM unit and use Lightning and TensorBoard to evaluate: Part 1 - Getting Started

Since we are using Lightning training, training the LSTM we created by hand is pretty easy. All we have to do is create the training data and put it into a DataLoader…

```
## create the training data for the neural network.
inputs = torch.tensor([[0., 0.5, 0.25, 1.], [1., 0.5, 0.25, 1.]])
labels = torch.tensor([0., 1.])

dataset = TensorDataset(inputs, labels)
dataloader = DataLoader(dataset)
```

```
# show the training data
for i, (input_i, label_i) in enumerate(dataloader):
    print("Training data: ", input_i, label_i)
```

```
Training data:  tensor([[0.0000, 0.5000, 0.2500, 1.0000]]) tensor([0.])
Training data:  tensor([[1.0000, 0.5000, 0.2500, 1.0000]]) tensor([1.])
```

...and then create a Lightning Trainer, L.Trainer, and fit it to the training data. NOTE: We are starting with 2000 epochs. This may be enough to successfully optimize all of the parameters, but it might not. We'll find out after we compare the predictions to the observed values.

```
trainer = L.Trainer(max_epochs=2000) # with default learning rate, 0.001 (this tiny learning rate ma
trainer.fit(model, train_dataloaders=dataloader)
```

```
Training: |          | 0/? [00:00<?, ?it/s]
```

Now that we've trained the model with 2000 epochs, we can see how good the predictions are...

```
print("\nNow let's compare the observed and predicted values...")
print("Company A: Observed = 0, Predicted =", model(torch.tensor([0., 0.5, 0.25, 1.])).detach())
print("Company B: Observed = 1, Predicted =", model(torch.tensor([1., 0.5, 0.25, 1.])).detach())
```

```
Now let's compare the observed and predicted values...
Company A: Observed = 0, Predicted = tensor(0.4342)
Company B: Observed = 1, Predicted = tensor(0.6171)
```

Unfortunately, these predictions are terrible. So it seems like we'll have to do more training. However, it would be awesome if we could be confident that more training will actually improve the predictions. If not, we can spare ourselves a lot of time, and potentially money, and just give up. So, before we dive into more training, let's look at the loss values and predictions that we saved in log files with TensorBoard. TensorBoard will graph everything that we logged during training, making it super easy to see if things are headed in the right direction or not.

To get TensorBoard working:

- First, check to see if the TensorBoard plugin is installed. If it's not, install it with the following command: pip install tensorboard
- Next, run the following command: `tensorboard --logdir lightning_logs`

NOTE: If your graphs look messed up and you see a bunch of different lines, instead of just one red line per graph, then check where this notebook is saved for a directory called `lightning_logs`. Delete `lightning_logs` and the re-run everything in this notebook. One source of problems with the graphs is that every time we train a model, a new batch of log files is created and stored in lightning_logs and TensorBoard, by default, will plot all of them. You can turn off unwanted log files in TensorBoard, and we'll do this later on in this notebook, but for now, the easiest thing to do is to start with a clean slate.

Anyway, if we look at the loss (trainloss), we see that it is going down, which is good, but it still has further to go. When we look at the predictions for Company A (out0), we see that they started out pretty good, close to 0, but then got really bad early on in training, shooting all the way up to 0.5, but are starting to get smaller. In contrast, when we look at the predictions for Company B (out_1), we see that they started out really bad, close to 0, but have been getting better ever since and look like they could continue to get better if we kept training.

In summary, the graphs seem to suggest that if we continued training our model, the predictions would improve. So let's add more epochs to the training.

19.4.2. Optimizing (Training) the Weights and Biases in the LSTM that we made by hand: Part 2 - Adding More Epochs without Starting Over

The good news is that because we're using Lightning, we can pick up where we left off training without having to start over from scratch. This is because when we train with Lightning, it creates checkpoint files that keep track of the Weights and Biases as they change. As a result, all we have to do to pick up where we left off is tell the Trainer where the checkpoint files are located. This is awesome and will save us a lot of time since we don't have to retrain the first 2000 epochs. So let's add an additional 1000 epochs to the training.

```
## First, find where the most recent checkpoint files are stored
path_to_checkpoint = trainer.checkpoint_callback.best_model_path ## By default, "best"
print("The new trainer will start where the last left off, and the check point data is
      path_to_checkpoint + "\n")

## Then create a new Lightning Trainer
trainer = L.Trainer(max_epochs=3000) # Before, max_epochs=2000, so, by setting it to 3
## And then call fit() using the path to the most recent checkpoint files
## so that we can pick up where we left off.
trainer.fit(model, train_dataloaders=dataloader, ckpt_path=path_to_checkpoint)
```

```
The new trainer will start where the last left off, and the check point data is here:
```

```
Training: |          | 0/? [00:00<?, ?it/s]
```

Now that we have added 1000 epochs to the training, let's check the predictions…

```
print("\nNow let's compare the observed and predicted values...")
print("Company A: Observed = 0, Predicted =", model(torch.tensor([0., 0.5, 0.25, 1.])).detach())
print("Company B: Observed = 1, Predicted =", model(torch.tensor([1., 0.5, 0.25, 1.])).detach())
```

```
Now let's compare the observed and predicted values...
Company A: Observed = 0, Predicted = tensor(0.2708)
Company B: Observed = 1, Predicted = tensor(0.7534)
```

The blue lines in each graph represents the values we logged during the extra 1000 epochs. The loss is getting smaller and the predictions for both companies are improving! Hooray!!! However, because it looks like there is even more room for improvement, let's add 2000 more epochs to the training.

```
## First, find where the most recent checkpoint files are stored
path_to_checkpoint = trainer.checkpoint_callback.best_model_path ## By default, "best" = "most recen
print("The new trainer will start where the last left off, and the check point data is here: " +
      path_to_checkpoint + "\n")

## Then create a new Lightning Trainer
trainer = L.Trainer(max_epochs=5000) # Before, max_epochs=3000, so, by setting it to 5000, we're add
## And then call fit() using the path to the most recent checkpoint files
## so that we can pick up where we left off.
trainer.fit(model, train_dataloaders=dataloader, ckpt_path=path_to_checkpoint)
```

```
The new trainer will start where the last left off, and the check point data is here: /Users/bartz/w
```

```
Training: |          | 0/? [00:00<?, ?it/s]
```

Now that we have added 2000 more epochs to the training (for a total of 5000 epochs), let's check the predictions.

```
print("\nNow let's compare the observed and predicted values...")
print("Company A: Observed = 0, Predicted =", model(torch.tensor([0., 0.5, 0.25, 1.])).detach())
print("Company B: Observed = 1, Predicted =", model(torch.tensor([1., 0.5, 0.25, 1.])).detach())
```

```
Now let's compare the observed and predicted values...
Company A: Observed = 0, Predicted = tensor(0.0022)
Company B: Observed = 1, Predicted = tensor(0.9693)
```

The prediction for Company A is super close to 0, which is exactly what we want, and the prediction for Company B is close to 1, which is also what we want.

The dark red lines show how things changed when we added an additional 2000 epochs to the training, for a total of 5000 epochs. Now we see that the loss (train_loss) and the predictions for each company appear to be tapering off, suggesting that adding more epochs may not improve the predictions much, so we're done!

Lastly, let's print out the final estimates for the Weights and Biases. In theory, they should be the same (within rounding error) as what we used in the StatQuest on Long Short-Term Memory and seen in the diagram of the LSTM unit at the top of this Jupyter notebook.

```
print("After optimization, the parameters are...")
for name, param in model.named_parameters():
    print(name, param.data)
```

```
After optimization, the parameters are...
wlr1 tensor(2.7043)
wlr2 tensor(1.6307)
blr1 tensor(1.6234)
wpr1 tensor(1.9983)
wpr2 tensor(1.6525)
bpr1 tensor(0.6204)
wp1 tensor(1.4122)
wp2 tensor(0.9393)
bp1 tensor(-0.3217)
wo1 tensor(4.3848)
wo2 tensor(-0.1943)
bo1 tensor(0.5935)
```

19.5. Using and optimzing the PyTorch LSTM, nn.LSTM()

Now that we know how to create an LSTM unit by hand, train it, and then use it to make good predictions, let's learn how to take advantage of PyTorch's `nn.LSTM()` function. For the most part, using nn.LSTM() allows us to simplify the init() function and the forward() function. The other big difference is that this time, we're not going

to try and recreate the parameter values we used in the StatQuest on Long Short-Term Memory, and that means we can set the learning rate for the Adam to 0.1. This will speed up training a lot. Everything else stays the same.

```
## Instead of coding an LSTM by hand, let's see what we can do with PyTorch's nn.LSTM()
class LightningLSTM(L.LightningModule):

    def __init__(self): # __init__() is the class constructor function, and we use it to initialize

        super().__init__() # initialize an instance of the parent class, LightningModule.

        L.seed_everything(seed=42)

        ## input_size = number of features (or variables) in the data. In our example
        ##              we only have a single feature (value)
        ## hidden_size = this determines the dimension of the output
        ##               in other words, if we set hidden_size=1, then we have 1 output node
        ##               if we set hidden_size=50, then we hve 50 output nodes (that can then be 50
        ##               nodes to a subsequent fully connected neural network.
        self.lstm = nn.LSTM(input_size=1, hidden_size=1)


    def forward(self, input):
        ## transpose the input vector
        input_trans = input.view(len(input), 1)

        lstm_out, temp = self.lstm(input_trans)

        ## lstm_out has the short-term memories for all inputs. We make our prediction with the last
        prediction = lstm_out[-1]
        return prediction


    def configure_optimizers(self): # this configures the optimizer we want to use for backpropagati
        return Adam(self.parameters(), lr=0.1) ## we'll just go ahead and set the learning rate to 0


    def training_step(self, batch, batch_idx): # take a step during gradient descent.
        input_i, label_i = batch # collect input
        output_i = self.forward(input_i[0]) # run input through the neural network
        loss = (output_i - label_i)**2 ## loss = squared residual
        self.log("train_loss", loss)

        if (label_i == 0):
            self.log("out_0", output_i)
```

```
        else:
            self.log("out_1", output_i)

        return loss
```

Now let's create the model and print out the initial Weights and Biases and predictions.

```
model = LightningLSTM() # First, make model from the class

## print out the name and value for each parameter
print("Before optimization, the parameters are...")
for name, param in model.named_parameters():
    print(name, param.data)

print("\nNow let's compare the observed and predicted values...")
print("Company A: Observed = 0, Predicted =", model(torch.tensor([0., 0.5, 0.25, 1.])
print("Company B: Observed = 1, Predicted =", model(torch.tensor([1., 0.5, 0.25, 1.])
```

```
Before optimization, the parameters are...
lstm.weight_ih_l0 tensor([[ 0.7645],
        [ 0.8300],
        [-0.2343],
        [ 0.9186]])
lstm.weight_hh_l0 tensor([[-0.2191],
        [ 0.2018],
        [-0.4869],
        [ 0.5873]])
lstm.bias_ih_l0 tensor([ 0.8815, -0.7336,  0.8692,  0.1872])
lstm.bias_hh_l0 tensor([ 0.7388,  0.1354,  0.4822, -0.1412])

Now let's compare the observed and predicted values...
Company A: Observed = 0, Predicted = tensor([0.6675])
Company B: Observed = 1, Predicted = tensor([0.6665])
```

As expected, the predictions are bad, so we will train the model. However, because we've increased the learning rate to 0.1, we only need to train for 300 epochs.

```
## NOTE: Because we have set Adam's learning rate to 0.1, we will train much, much fas
## Before, with the hand made LSTM and the default learning rate, 0.001, it took about
## the model. Now, with the learning rate set to 0.1, we only need 300 epochs. Now, be
## we have to tell the trainer add stuff to the log files every 2 steps (or epoch, sin
## because the default, updating the log files every 50 steps, will result in a terrib
```

```
trainer = L.Trainer(max_epochs=300, log_every_n_steps=2)

trainer.fit(model, train_dataloaders=dataloader)

print("After optimization, the parameters are...")
for name, param in model.named_parameters():
    print(name, param.data)
```

```
Training: |          | 0/? [00:00<?, ?it/s]


After optimization, the parameters are...
lstm.weight_ih_l0 tensor([[3.5364],
        [1.3869],
        [1.5390],
        [1.2488]])
lstm.weight_hh_l0 tensor([[5.2070],
        [2.9577],
        [3.2652],
        [2.0678]])
lstm.bias_ih_l0 tensor([-0.9143,  0.3724, -0.1815,  0.6376])
lstm.bias_hh_l0 tensor([-1.0570,  1.2414, -0.5685,  0.3092])
```

Now that training is done, let's print out the new predictions…

```
print("\nNow let's compare the observed and predicted values...")
print("Company A: Observed = 0, Predicted =", model(torch.tensor([0., 0.5, 0.25, 1.])).detach())
print("Company B: Observed = 1, Predicted =", model(torch.tensor([1., 0.5, 0.25, 1.])).detach())
```

```
Now let's compare the observed and predicted values...
Company A: Observed = 0, Predicted = tensor([6.8527e-05])
Company B: Observed = 1, Predicted = tensor([0.9809])
```

…and, as we can see, after just 300 epochs, the LSTM is making great predictions. The prediction for Company A is close to the observed value 0 and the prediction for Company B is close to the observed value 1.

Lastly, let's go back to TensorBoard to see the latest graphs. NOTE: To make it easier to see what we just did, deselect version0, version1 and version2 and make sure version3 is checked on the left-hand side of the page, under where it says Runs. This allows us to just look at the log files from the most recent training, which only went for 300 epochs.

In all three graphs, the loss (trainloss) and the predictions for Company A (out0) and Company B (out_1) started to taper off after 500 steps, or just 250 epochs, suggesting that adding more epochs may not improve the predictions much, so we're done!

Part V.

Introduction to Hyperparameter Tuning

20. Hyperparameter Tuning

20.1. Structure of the Hyperparameter Tuning Chapters

The first part is structured as follows:

The concept of the hyperparameter tuning is described in Section 20.2.

Hyperparameter tuning with sklearn in Python is described in Chapter 21.

Hyperparameter tuning with river in Python is described in Chapter 24.

This part of the book is concluded with a description of the most recent `PyTorch` hyperparameter tuning approach, which is the integration of `spotpython` into the `PyTorch Lightning` training workflow. Hyperparameter tuning with PyTorch Lightning in Python is described in Chapter 32. This is considered as the most effective, efficient, and flexible way to integrate `spotpython` into the `PyTorch` training workflow.

Figure 20.1 shows the graphical user interface of `spotpython` that is used in this book.

Figure 20.1.: spot GUI

20.2. Goals of Hyperparameter Tuning

The goal of hyperparameter tuning is to optimize the hyperparameters in a way that improves the performance of the machine learning or deep learning model. Hyperparameters are parameters that are not learned during the training process, but are set before the training process begins. Hyperparameter tuning is an important, but often

difficult and computationally intensive task. Changing the architecture of a neural network or the learning rate of an optimizer can have a significant impact on the performance.

Hyperparameter tuning is referred to as "hyperparameter optimization" (HPO) in the literature. However, since we do not consider the optimization, but also the understanding of the hyperparameters, we use the term "hyperparameter tuning" in this book. See also the discussion in Chapter 2 of Bartz et al. (2022), which lays the groundwork and presents an introduction to the process of tuning Machine Learning and Deep Learning hyperparameters and the respective methodology. Since the key elements such as the hyperparameter tuning process and measures of tunability and performance are presented in Bartz et al. (2022), we refer to this chapter for details.

The simplest, but also most computationally expensive, hyperparameter tuning approach uses manual search (or trial-and-error (Meignan et al. 2015)). Commonly encountered is simple random search, i.e., random and repeated selection of hyperparameters for evaluation, and lattice search ("grid search"). In addition, methods that perform directed search and other model-free algorithms, i.e., algorithms that do not explicitly rely on a model, e.g., evolution strategies (Bartz-Beielstein et al. 2014) or pattern search (Lewis, Torczon, and Trosset 2000) play an important role. Also, "hyperband", i.e., a multi-armed bandit strategy that dynamically allocates resources to a set of random configurations and uses successive bisections to stop configurations with poor performance (Li et al. 2016), is very common in hyperparameter tuning. The most sophisticated and efficient approaches are the Bayesian optimization and surrogate model based optimization methods, which are based on the optimization of cost functions determined by simulations or experiments.

We consider a surrogate optimization based hyperparameter tuning approach that uses the Python version of the SPOT ("Sequential Parameter Optimization Toolbox") (Bartz-Beielstein, Lasarczyk, and Preuss 2005), which is suitable for situations where only limited resources are available. This may be due to limited availability and cost of hardware, or due to the fact that confidential data may only be processed locally, e.g., due to legal requirements. Furthermore, in our approach, the understanding of algorithms is seen as a key tool for enabling transparency and explainability. This can be enabled, for example, by quantifying the contribution of machine learning and deep learning components (nodes, layers, split decisions, activation functions, etc.). Understanding the importance of hyperparameters and the interactions between multiple hyperparameters plays a major role in the interpretability and explainability of machine learning models. SPOT provides statistical tools for understanding hyperparameters and their interactions. Last but not least, it should be noted that the SPOT software code is available in the open source `spotpython` package on github[1], allowing replicability of the results. This tutorial describes the Python variant of SPOT, which is called `spotpython`. The R implementation is described in Bartz et al. (2022). SPOT is an established open source software that has been maintained for more than 15 years (Bartz-Beielstein, Lasarczyk, and Preuss 2005) (Bartz et al. 2022).

[1] https://github.com/sequential-parameter-optimization

Part VI.

Hyperparameter Tuning with Sklearn

21. HPT: sklearn

21.1. Introduction to sklearn

22. HPT: sklearn SVC on Moons Data

This chapter is a tutorial for the Hyperparameter Tuning (HPT) of a `sklearn` SVC model on the Moons dataset.

22.1. Step 1: Setup

Before we consider the detailed experimental setup, we select the parameters that affect run time, initial design size and the device that is used.

> Caution: Run time and initial design size should be increased for real experiments
>
> - MAX_TIME is set to one minute for demonstration purposes. For real experiments, this should be increased to at least 1 hour.
> - INIT_SIZE is set to 5 for demonstration purposes. For real experiments, this should be increased to at least 10.

```
MAX_TIME = 1
INIT_SIZE = 10
PREFIX = "10"
```

22.2. Step 2: Initialization of the Empty `fun_control` Dictionary

`spotpython` supports the visualization of the hyperparameter tuning process with TensorBoard. The following example shows how to use TensorBoard with `spotpython`. The `fun_control` dictionary is the central data structure that is used to control the optimization process. It is initialized as follows:

```
from spotpython.utils.init import fun_control_init
from spotpython.hyperparameters.values import set_control_key_value
from spotpython.utils.eda import print_res_table
fun_control = fun_control_init(
```

```
    PREFIX=PREFIX,
    TENSORBOARD_CLEAN=True,
    max_time=MAX_TIME,
    fun_evals=inf,
    tolerance_x = np.sqrt(np.spacing(1)))
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_02_12_22_20
```

> Tip: TensorBoard
>
> - Since the `spot_tensorboard_path` argument is not `None`, which is the default, `spotpython` will log the optimization process in the TensorBoard folder.
> - The `TENSORBOARD_CLEAN` argument is set to `True` to archive the TensorBoard folder if it already exists. This is useful if you want to start a hyperparameter tuning process from scratch. If you want to continue a hyperparameter tuning process, set `TENSORBOARD_CLEAN` to `False`. Then the TensorBoard folder will not be archived and the old and new TensorBoard files will shown in the TensorBoard dashboard.

22.3. Step 3: SKlearn Load Data (Classification)

Randomly generate classification data.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons, make_circles, make_classification
n_features = 2
n_samples = 500
target_column = "y"
ds =  make_moons(n_samples, noise=0.5, random_state=0)
X, y = ds
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
train = pd.DataFrame(np.hstack((X_train, y_train.reshape(-1, 1))))
test = pd.DataFrame(np.hstack((X_test, y_test.reshape(-1, 1))))
train.columns = [f"x{i}" for i in range(1, n_features+1)] + [target_column]
test.columns = [f"x{i}" for i in range(1, n_features+1)] + [target_column]
train.head()
```

	x1	x2	y
0	1.960101	0.383172	0.0
1	2.354420	-0.536942	1.0
2	1.682186	-0.332108	0.0
3	1.856507	0.687220	1.0
4	1.925524	0.427413	1.0

```
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
cm = plt.cm.RdBu
cm_bright = ListedColormap(["#FF0000", "#0000FF"])
ax = plt.subplot(1, 1, 1)
ax.set_title("Input data")
# Plot the training points
ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright, edgecolors="k")
# Plot the testing points
ax.scatter(
    X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alpha=0.6, edgecolors="k"
)
ax.set_xlim(x_min, x_max)
ax.set_ylim(y_min, y_max)
ax.set_xticks(())
ax.set_yticks(())
plt.tight_layout()
plt.show()
```

```
n_samples = len(train)
# add the dataset to the fun_control
fun_control.update({"data": None, # dataset,
                "train": train,
                "test": test,
                "n_samples": n_samples,
                "target_column": target_column})
```

22.4. Step 4: Specification of the Preprocessing Model

Data preprocesssing can be very simple, e.g., you can ignore it. Then you would choose the `prep_model` "None":

```
prep_model = None
fun_control.update({"prep_model": prep_model})
```

A default approach for numerical data is the `StandardScaler` (mean 0, variance 1). This can be selected as follows:

```
from sklearn.preprocessing import StandardScaler
prep_model = StandardScaler
fun_control.update({"prep_model": prep_model})
```

Even more complicated pre-processing steps are possible, e.g., the follwing pipeline:

```
categorical_columns = []
one_hot_encoder = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
prep_model = ColumnTransformer(
         transformers=[
             ("categorical", one_hot_encoder, categorical_columns),
         ],
         remainder=StandardScaler,
     )
```

22.5. Step 5: Select Model (algorithm) and core_model_hyper_dict

The selection of the algorithm (ML model) that should be tuned is done by specifying the its name from the `sklearn` implementation. For example, the `SVC` support vector machine classifier is selected as follows:

```
from spotpython.hyperparameters.values import add_core_model_to_fun_control
from spotpython.hyperdict.sklearn_hyper_dict import SklearnHyperDict
from sklearn.svm import SVC
add_core_model_to_fun_control(core_model=SVC,
                              fun_control=fun_control,
                              hyper_dict=SklearnHyperDict,
                              filename=None)
```

Now `fun_control` has the information from the JSON file. The corresponding entries for the `core_model` class are shown below.

```
fun_control['core_model_hyper_dict']
```

```
{'C': {'type': 'float',
  'default': 1.0,
  'transform': 'None',
  'lower': 0.1,
  'upper': 10.0},
 'kernel': {'levels': ['linear', 'poly', 'rbf', 'sigmoid'],
  'type': 'factor',
  'default': 'rbf',
  'transform': 'None',
  'core_model_parameter_type': 'str',
```

```
  'lower': 0,
  'upper': 3},
 'degree': {'type': 'int',
  'default': 3,
  'transform': 'None',
  'lower': 3,
  'upper': 3},
 'gamma': {'levels': ['scale', 'auto'],
  'type': 'factor',
  'default': 'scale',
  'transform': 'None',
  'core_model_parameter_type': 'str',
  'lower': 0,
  'upper': 1},
 'coef0': {'type': 'float',
  'default': 0.0,
  'transform': 'None',
  'lower': 0.0,
  'upper': 0.0},
 'shrinking': {'levels': [0, 1],
  'type': 'factor',
  'default': 0,
  'transform': 'None',
  'core_model_parameter_type': 'bool',
  'lower': 0,
  'upper': 1},
 'probability': {'levels': [0, 1],
  'type': 'factor',
  'default': 0,
  'transform': 'None',
  'core_model_parameter_type': 'bool',
  'lower': 0,
  'upper': 1},
 'tol': {'type': 'float',
  'default': 0.001,
  'transform': 'None',
  'lower': 0.0001,
  'upper': 0.01},
 'cache_size': {'type': 'float',
  'default': 200,
  'transform': 'None',
  'lower': 100,
  'upper': 400},
 'break_ties': {'levels': [0, 1],
  'type': 'factor',
```

```
'default': 0,
'transform': 'None',
'core_model_parameter_type': 'bool',
'lower': 0,
'upper': 1}}
```

sklearn Model Selection

The following sklearn models are supported by default:

- RidgeCV
- RandomForestClassifier
- SVC
- LogisticRegression
- KNeighborsClassifier
- GradientBoostingClassifier
- GradientBoostingRegressor
- ElasticNet

They can be imported as follows:

```
from sklearn.linear_model import RidgeCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.linear_model import ElasticNet
```

22.6. Step 6: Modify hyper_dict Hyperparameters for the Selected Algorithm aka core_model

spotpython provides functions for modifying the hyperparameters, their bounds and factors as well as for activating and de-activating hyperparameters without re-compilation of the Python source code. These functions were described in Section D.15.1.

22.6.1. Modify hyperparameter of type numeric and integer (boolean)

Numeric and boolean values can be modified using the `modify_hyper_parameter_bounds` method.

> **`sklearn Model` Hyperparameters**
>
> The hyperparameters of the `sklearn SVC` model are described in the sklearn documentation.

- For example, to change the `tol` hyperparameter of the `SVC` model to the interval [1e-5, 1e-3], the following code can be used:

```
from spotpython.hyperparameters.values import modify_hyper_parameter_bounds
modify_hyper_parameter_bounds(fun_control, "tol", bounds=[1e-5, 1e-3])
modify_hyper_parameter_bounds(fun_control, "probability", bounds=[0, 0])
fun_control["core_model_hyper_dict"]["tol"]
```

```
{'type': 'float',
 'default': 0.001,
 'transform': 'None',
 'lower': 1e-05,
 'upper': 0.001}
```

22.6.2. Modify hyperparameter of type factor

Factors can be modified with the `modify_hyper_parameter_levels` function. For example, to exclude the `sigmoid` kernel from the tuning, the `kernel` hyperparameter of the `SVC` model can be modified as follows:

```
from spotpython.hyperparameters.values import modify_hyper_parameter_levels
modify_hyper_parameter_levels(fun_control, "kernel", ["poly", "rbf"])
fun_control["core_model_hyper_dict"]["kernel"]
```

```
{'levels': ['poly', 'rbf'],
 'type': 'factor',
 'default': 'rbf',
 'transform': 'None',
 'core_model_parameter_type': 'str',
 'lower': 0,
 'upper': 1}
```

22.6.3. Optimizers

Optimizers are described in Section 4.2.

22.7. Step 7: Selection of the Objective (Loss) Function

There are two metrics:

1. `metric_river` is used for the river based evaluation via `eval_oml_iter_progressive`.
2. `metric_sklearn` is used for the sklearn based evaluation.

```
from sklearn.metrics import mean_absolute_error, accuracy_score, roc_curve, roc_auc_score, log_loss,
fun_control.update({
                "metric_sklearn": log_loss,
                "weights": 1.0,
                })
```

`metric_sklearn`: Minimization and Maximization

- Because the `metric_sklearn` is used for the sklearn based evaluation, it is important to know whether the metric should be minimized or maximized.
- The `weights` parameter is used to indicate whether the metric should be minimized or maximized.
- If `weights` is set to `-1.0`, the metric is maximized.
- If `weights` is set to `1.0`, the metric is minimized, e.g., `weights = 1.0` for `mean_absolute_error`, or `weights = -1.0` for `roc_auc_score`.

22.7.1. Predict Classes or Class Probabilities

If the key `"predict_proba"` is set to `True`, the class probabilities are predicted. `False` is the default, i.e., the classes are predicted.

```
fun_control.update({
                "predict_proba": False,
                })
```

22.8. Step 8: Calling the SPOT Function

22.8.1. The Objective Function

The objective function is selected next. It implements an interface from `sklearn`'s training, validation, and testing methods to `spotpython`.

```
from spotpython.fun.hypersklearn import HyperSklearn
fun = HyperSklearn().fun_sklearn
```

The following code snippet shows how to get the default hyperparameters as an array, so that they can be passed to the `Spot` function.

```
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array
X_start = get_default_hyperparameters_as_array(fun_control)
```

22.8.2. Run the `Spot` Optimizer

The class `Spot` [SOURCE] is the hyperparameter tuning workhorse. It is initialized with the following parameters:

- `fun`: the objective function
- `fun_control`: the dictionary with the control parameters for the objective function
- `design`: the experimental design
- `design_control`: the dictionary with the control parameters for the experimental design
- `surrogate`: the surrogate model
- `surrogate_control`: the dictionary with the control parameters for the surrogate model
- `optimizer`: the optimizer
- `optimizer_control`: the dictionary with the control parameters for the optimizer

> **Note: Total run time**
>
> The total run time may exceed the specified `max_time`, because the initial design (here: `init_size` = INIT_SIZE as specified above) is always evaluated, even if this takes longer than `max_time`.

```
from spotpython.utils.init import design_control_init, surrogate_control_init
design_control = design_control_init()
set_control_key_value(control_dict=design_control,
                        key="init_size",
                        value=INIT_SIZE,
                        replace=True)

surrogate_control = surrogate_control_init(noise=True,
                                           n_theta=2)
from spotpython.spot import Spot
spot_tuner = Spot(fun=fun,
                  fun_control=fun_control,
                  design_control=design_control,
                  surrogate_control=surrogate_control)
spot_tuner.run(X_start=X_start)
```

```
spotpython tuning: 6.436366676628063 [----------] 3.77%
spotpython tuning: 6.436366676628063 [#---------] 5.75%
spotpython tuning: 6.436366676628063 [#---------] 7.59%
spotpython tuning: 6.436366676628063 [#---------] 9.38%
spotpython tuning: 6.436366676628063 [#---------] 11.21%
spotpython tuning: 6.436366676628063 [#---------] 12.98%
spotpython tuning: 6.436366676628063 [#---------] 14.82%
spotpython tuning: 6.436366676628063 [##--------] 18.22%
spotpython tuning: 6.436366676628063 [##--------] 19.90%
spotpython tuning: 6.436366676628063 [##--------] 21.78%
spotpython tuning: 6.436366676628063 [##--------] 23.50%
spotpython tuning: 6.436366676628063 [###-------] 25.17%
spotpython tuning: 6.436366676628063 [###-------] 26.98%
spotpython tuning: 6.436366676628063 [###-------] 31.78%
spotpython tuning: 6.436366676628063 [###-------] 33.55%
spotpython tuning: 6.436366676628063 [####------] 36.24%
spotpython tuning: 6.436366676628063 [####------] 38.84%
spotpython tuning: 6.436366676628063 [####------] 44.89%
spotpython tuning: 6.436366676628063 [#####-----] 47.35%
spotpython tuning: 6.436366676628063 [#####-----] 53.39%
spotpython tuning: 6.436366676628063 [######----] 55.59%
spotpython tuning: 6.436366676628063 [######----] 57.51%
spotpython tuning: 6.436366676628063 [######----] 59.49%
spotpython tuning: 6.436366676628063 [######----] 61.36%
spotpython tuning: 6.436366676628063 [######----] 63.06%
spotpython tuning: 6.436366676628063 [######----] 64.79%
spotpython tuning: 6.436366676628063 [#######---] 66.54%
spotpython tuning: 6.436366676628063 [#######---] 68.26%
```

```
spotpython tuning: 6.436366676628063 [#######---] 69.88%
spotpython tuning: 6.436366676628063 [#######---] 71.56%
spotpython tuning: 6.436366676628063 [#######---] 73.22%
spotpython tuning: 6.436366676628063 [########--] 75.40%
spotpython tuning: 6.436366676628063 [########--] 77.63%
spotpython tuning: 6.436366676628063 [########--] 79.93%
spotpython tuning: 6.436366676628063 [########--] 82.19%
spotpython tuning: 6.436366676628063 [########--] 83.98%
spotpython tuning: 6.436366676628063 [#########-] 86.65%
spotpython tuning: 6.436366676628063 [#########-] 88.78%
spotpython tuning: 6.436366676628063 [#########-] 90.27%
spotpython tuning: 6.436366676628063 [#########-] 91.98%
spotpython tuning: 6.436366676628063 [#########-] 93.42%
spotpython tuning: 6.436366676628063 [##########] 95.46%
spotpython tuning: 6.436366676628063 [##########] 97.06%
spotpython tuning: 6.436366676628063 [##########] 99.33%
spotpython tuning: 6.436366676628063 [##########] 100.00% Done...

Experiment saved to 10_res.pkl


<spotpython.spot.spot.Spot at 0x1142acfb0>
```

22.8.3. TensorBoard

Now we can start TensorBoard in the background with the following command, where `./runs` is the default directory for the TensorBoard log files:

```
tensorboard --logdir="./runs"
```

> **Tip: TENSORBOARD_PATH**
>
> The TensorBoard path can be printed with the following command:
>
> ```
> from spotpython.utils.init import get_tensorboard_path
> get_tensorboard_path(fun_control)
> ```
>
> ```
> 'runs/'
> ```

We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

The TensorBoard plot illustrates how `spotpython` can be used as a microscope for the internal mechanisms of the surrogate-based optimization process. Here, one important parameter, the learning rate θ of the Kriging surrogate [SOURCE] is plotted against the number of optimization steps.

Figure 22.1.: TensorBoard visualization of the spotpython optimization process and the surrogate model.

22.9. Step 9: Results

After the hyperparameter tuning run is finished, the results can be saved and reloaded with the following commands:

```
from spotpython.utils.file import save_pickle, load_pickle
from spotpython.utils.init import get_experiment_name
experiment_name = get_experiment_name(PREFIX)
SAVE_AND_LOAD = False
if SAVE_AND_LOAD == True:
    save_pickle(spot_tuner, experiment_name)
    spot_tuner = load_pickle(experiment_name)
```

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress(log_y=True, filename="./figures/" + experiment_name+"_progres
```

Results can also be printed in tabular form.

```
print_res_table(spot_tuner)
```

```
| name        | type   | default   |   lower |   upper | tuned                  | trans
|-------------|--------|-----------|---------|---------|------------------------|------
| C           | float  | 1.0       |     0.1 |    10.0 | 1.3459476182876375     | None
| kernel      | factor | rbf       |     0.0 |     1.0 | rbf                    | None
| degree      | int    | 3         |     3.0 |     3.0 | 3.0                    | None
| gamma       | factor | scale     |     0.0 |     1.0 | scale                  | None
| coef0       | float  | 0.0       |     0.0 |     0.0 | 0.0                    | None
| shrinking   | factor | 0         |     0.0 |     1.0 | 1                      | None
| probability | factor | 0         |     0.0 |     0.0 | 0                      | None
| tol         | float  | 0.001     |   1e-05 |   0.001 | 2.988661226661179e-05  | None
| cache_size  | float  | 200.0     |   100.0 |   400.0 | 174.45504889441855     | None
| break_ties  | factor | 0         |     0.0 |     1.0 | 0                      | None
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=0.0025, filename="./figures/" + experiment_name+"
```

22.10. Get Default Hyperparameters

The default hyperparameters, whihc will be used for a comparion with the tuned hyperparameters, can be obtained with the following commands:

```
from spotpython.hyperparameters.values import get_one_core_model_from_X
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array
X_start = get_default_hyperparameters_as_array(fun_control)
model_default = get_one_core_model_from_X(X_start, fun_control, default=True)
model_default
```

```
SVC(cache_size=200.0, shrinking=False)
```

22.11. Get SPOT Results

In a similar way, we can obtain the hyperparameters found by `spotpython`.

```
from spotpython.hyperparameters.values import get_one_core_model_from_X
X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
model_spot = get_one_core_model_from_X(X, fun_control)
```

22.11.1. Plot: Compare Predictions

```
from spotpython.plot.validation import plot_roc
plot_roc(model_list=[model_default, model_spot], fun_control= fun_control, model_names
```

```
from spotpython.plot.validation import plot_confusion_matrix
plot_confusion_matrix(model=model_default, fun_control=fun_control, title = "Default")
```

```
plot_confusion_matrix(model=model_spot, fun_control=fun_control, title="SPOT")
```

```
min(spot_tuner.y), max(spot_tuner.y)
```

```
(np.float64(6.436366676628063), np.float64(10.813096016735146))
```

22.11.2. Detailed Hyperparameter Plots

```
spot_tuner.plot_important_hyperparameter_contour(filename=None)
```

```
C:  0.8944193924194259
kernel:  100.0
gamma:  0.005227140802824073
shrinking:  0.005813092525617361
```

```
tol:  0.07399846774416982
cache_size:  0.005227140802824073
break_ties:  0.005227140802824073
```

x6: break_ties
x1: kernel
x1: kernel
x0: C
x4: tol
x0: C
x3: shrinking
x0: C

x2: gamma
x0: C

x0: C
x2: gamma

x5: cache_size
x0: C

x0: C
x5: cache_size

x6: break_ties
x0: C

x0: C
x6: break_ties

x5: cache_size
x4: tol

x4: tol
x5: cache_size

x6: break_ties
x4: tol
7.59808
7.59792
7.59776
7.59760
7.59744
7.59728
x4: tol
x3: shrinking
7.59728
7.59712
7.59696
7.59680
7.59664
7.59648
x3: shrinking
x5: cache_size
8.4
8.1
7.8
7.5
7.2
6.9
x3: shrinking
x5: cache_size
x6: break_ties
7.5982
7.5979
7.5976
7.5973
7.5970
7.5967
7.5964
x3: shrinking
x6: break_ties

x4: tol
x2: gamma
0.0010
0.0008
0.0006
0.0004
0.0002
0.0
0.2
0.4
0.6
0.8
1.0
7.59736
7.59720
7.59704
7.59688
7.59672
7.59656

x2: gamma
x4: tol
7.5972
7.5970
7.5968
7.5966
0.00
0.25
0.50
0.75
1.00
0.00100
0.00075
0.00050
0.00025
0.00000

x3: shrinking
x2: gamma
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
7.59745
7.59715
7.59685
7.59655
7.59625
7.59595
7.59565

x2: gamma
x3: shrinking
7.5970
7.5965
7.5960
0.00
0.25
0.50
0.75
1.00

x5: cache_size
x2: gamma
400
350
300
250
200
150
100
0.0
0.2
0.4
0.6
0.8
1.0
8.4
8.1
7.8
7.5
7.2
6.9

x2: gamma
x5: cache_size
8.5
8.0
7.5
7.0
0.00
0.25
0.50
0.75
1.00
400
300
200
100

x6: break_ties
x2: gamma
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
7.59805
7.59775
7.59745
7.59715
7.59685
7.59655

x2: gamma
x6: break_ties
7.5980
7.5975
7.5970
0.00
0.25
0.50
0.75
1.00

22.11.3. Parallel Coordinates Plot

```
spot_tuner.parallel_plot()
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

22.11.4. Plot all Combinations of Hyperparameters

- Warning: this may take a while.

```
PLOT_ALL = False
if PLOT_ALL:
    n = spot_tuner.k
    for i in range(n-1):
        for j in range(i+1, n):
            spot_tuner.plot_contour(i=i, j=j, min_z=min_z, max_z = max_z)
```

23. Step 2: Initialization of the Empty `fun_control` Dictionary

```
MAX_TIME = 1
INIT_SIZE = 20
PREFIX = "18"
```

`spotpython` supports the visualization of the hyperparameter tuning process with TensorBoard. The following example shows how to use TensorBoard with `spotpython`. The `fun_control` dictionary is the central data structure that is used to control the optimization process. It is initialized as follows:

```
from spotpython.utils.init import fun_control_init
from spotpython.hyperparameters.values import set_control_key_value
from spotpython.utils.eda import print_res_table
fun_control = fun_control_init(
    PREFIX=PREFIX,
    TENSORBOARD_CLEAN=True,
    max_time=MAX_TIME,
    fun_evals=inf,
    tolerance_x = np.sqrt(np.spacing(1)))
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_02_12_22_22_52_0
```

Tip: TensorBoard

- Since the `spot_tensorboard_path` argument is not `None`, which is the default, `spotpython` will log the optimization process in the TensorBoard folder.
- The `TENSORBOARD_CLEAN` argument is set to `True` to archive the TensorBoard folder if it already exists. This is useful if you want to start a hyperparameter tuning process from scratch. If you want to continue a hyperparameter tuning process, set `TENSORBOARD_CLEAN` to `False`. Then the TensorBoard folder will not be archived and the old and new TensorBoard files will shown in the TensorBoard dashboard.

23.1. Step 3: SKlearn Load Data (Classification)

Randomly generate classification data. Here, we use similar data as in Comparison of kernel ridge regression and SVR.

```
import numpy as np

rng = np.random.RandomState(42)

X = 5 * rng.rand(10, 1)
y = np.sin(1/X).ravel()*np.cos(X).ravel()

# Add noise to targets
y[::5] += 3 * (0.5 - rng.rand(X.shape[0] // 5))

X_plot = np.linspace(0, 5, 100000)[:, None]
```

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

n_features = 1
target_column = "y"
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
train = pd.DataFrame(np.hstack((X_train, y_train.reshape(-1, 1))))
test = pd.DataFrame(np.hstack((X_test, y_test.reshape(-1, 1))))
train.columns = [f"x{i}" for i in range(1, n_features+1)] + [target_column]
test.columns = [f"x{i}" for i in range(1, n_features+1)] + [target_column]
train.head()
```

	x1	y
0	1.872701	1.286910
1	4.330881	-0.085207
2	3.659970	-0.234389
3	3.540363	-0.256848
4	0.780093	0.681389

```
n_samples = len(train)
# add the dataset to the fun_control
fun_control.update({"data": None, # dataset,
```

```
                "train": train,
                "test": test,
                "n_samples": n_samples,
                "target_column": target_column})
```

23.2. Step 4: Specification of the Preprocessing Model

Data preprocesssing can be very simple, e.g., you can ignore it. Then you would choose the `prep_model` "None":

```
prep_model = None
fun_control.update({"prep_model": prep_model})
```

A default approach for numerical data is the `StandardScaler` (mean 0, variance 1). This can be selected as follows:

```
from sklearn.preprocessing import StandardScaler
prep_model = StandardScaler
fun_control.update({"prep_model": prep_model})
```

Even more complicated pre-processing steps are possible, e.g., the follwing pipeline:

```
categorical_columns = []
one_hot_encoder = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
prep_model = ColumnTransformer(
         transformers=[
             ("categorical", one_hot_encoder, categorical_columns),
         ],
         remainder=StandardScaler,
     )
```

23.3. Step 5: Select Model (`algorithm`) and `core_model_hyper_dict`

The selection of the algorithm (ML model) that should be tuned is done by specifying the its name from the `sklearn` implementation. For example, the `SVC` support vector machine classifier is selected as follows:

```
from spotpython.hyperparameters.values import add_core_model_to_fun_control
from spotpython.hyperdict.sklearn_hyper_dict import SklearnHyperDict
from sklearn.svm import SVR
add_core_model_to_fun_control(core_model=SVR,
                              fun_control=fun_control,
                              hyper_dict=SklearnHyperDict,
                              filename=None)
```

Now fun_control has the information from the JSON file. The corresponding entries for the core_model class are shown below.

```
fun_control['core_model_hyper_dict']
```

```
{'C': {'type': 'float',
  'default': 1.0,
  'transform': 'None',
  'lower': 0.1,
  'upper': 10.0},
 'kernel': {'levels': ['linear', 'poly', 'rbf', 'sigmoid'],
  'type': 'factor',
  'default': 'rbf',
  'transform': 'None',
  'core_model_parameter_type': 'str',
  'lower': 0,
  'upper': 3},
 'degree': {'type': 'int',
  'default': 3,
  'transform': 'None',
  'lower': 3,
  'upper': 3},
 'gamma': {'levels': ['scale', 'auto'],
  'type': 'factor',
  'default': 'scale',
  'transform': 'None',
  'core_model_parameter_type': 'str',
  'lower': 0,
  'upper': 1},
 'coef0': {'type': 'float',
  'default': 0.0,
  'transform': 'None',
  'lower': 0.0,
  'upper': 0.0},
 'epsilon': {'type': 'float',
```

```
 'default': 0.1,
 'transform': 'None',
 'lower': 0.01,
 'upper': 1.0},
'shrinking': {'levels': [0, 1],
 'type': 'factor',
 'default': 0,
 'transform': 'None',
 'core_model_parameter_type': 'bool',
 'lower': 0,
 'upper': 1},
'tol': {'type': 'float',
 'default': 0.001,
 'transform': 'None',
 'lower': 0.0001,
 'upper': 0.01},
'cache_size': {'type': 'float',
 'default': 200,
 'transform': 'None',
 'lower': 100,
 'upper': 400}}
```

sklearn Model Selection

The following sklearn models are supported by default:

- RidgeCV
- RandomForestClassifier
- SVC
- SVR
- LogisticRegression
- KNeighborsClassifier
- GradientBoostingClassifier
- GradientBoostingRegressor
- ElasticNet

They can be imported as follows:

```
from sklearn.linear_model import RidgeCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.svm import SVR
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.linear_model import ElasticNet
```

23.4. Step 6: Modify hyper_dict Hyperparameters for the Selected Algorithm aka core_model

spotpython provides functions for modifying the hyperparameters, their bounds and factors as well as for activating and de-activating hyperparameters without re-compilation of the Python source code. These functions were described in Section D.15.1.

23.4.1. Modify hyperparameter of type numeric and integer (boolean)

Numeric and boolean values can be modified using the modify_hyper_parameter_bounds method.

> **sklearn Model Hyperparameters**
>
> The hyperparameters of the sklearn SVC model are described in the sklearn documentation.

- For example, to change the tol hyperparameter of the SVC model to the interval [1e-5, 1e-3], the following code can be used:

```
from spotpython.hyperparameters.values import modify_hyper_parameter_bounds
modify_hyper_parameter_bounds(fun_control, "tol", bounds=[1e-5, 1e-3])
modify_hyper_parameter_bounds(fun_control, "epsilon", bounds=[0.1, 1.0])
# modify_hyper_parameter_bounds(fun_control, "degree", bounds=[2, 5])
fun_control["core_model_hyper_dict"]["tol"]
```

```
{'type': 'float',
 'default': 0.001,
 'transform': 'None',
 'lower': 1e-05,
 'upper': 0.001}
```

23.4.2. Modify hyperparameter of type factor

Factors can be modified with the `modify_hyper_parameter_levels` function. For example, to exclude the `sigmoid` kernel from the tuning, the `kernel` hyperparameter of the `SVR` model can be modified as follows:

```
from spotpython.hyperparameters.values import modify_hyper_parameter_levels
# modify_hyper_parameter_levels(fun_control, "kernel", ["poly", "rbf"])
modify_hyper_parameter_levels(fun_control, "kernel", ["rbf"])
fun_control["core_model_hyper_dict"]["kernel"]
```

```
{'levels': ['rbf'],
 'type': 'factor',
 'default': 'rbf',
 'transform': 'None',
 'core_model_parameter_type': 'str',
 'lower': 0,
 'upper': 0}
```

23.4.3. Optimizers

Optimizers are described in Section 4.2.

23.5. Step 7: Selection of the Objective (Loss) Function

There are two metrics:

1. `metric_river` is used for the river based evaluation via `eval_oml_iter_progressive`.
2. `metric_sklearn` is used for the sklearn based evaluation.

```
from sklearn.metrics import mean_absolute_error, accuracy_score, roc_curve, roc_auc_score, log_loss,
fun_control.update({
               "metric_sklearn": mean_squared_error,
               "weights": 1.0,
               })
```

metric_sklearn: Minimization and Maximization

- Because the metric_sklearn is used for the sklearn based evaluation, it is important to know whether the metric should be minimized or maximized.
- The weights parameter is used to indicate whether the metric should be minimized or maximized.
- If weights is set to -1.0, the metric is maximized.
- If weights is set to 1.0, the metric is minimized, e.g., weights = 1.0 for mean_absolute_error, or weights = -1.0 for roc_auc_score.

23.5.1. Predict Classes or Class Probabilities

If the key "predict_proba" is set to True, the class probabilities are predicted. False is the default, i.e., the classes are predicted.

```
fun_control.update({
                "predict_proba": False,
                })
```

23.6. Step 8: Calling the SPOT Function

23.6.1. The Objective Function

The objective function is selected next. It implements an interface from sklearn's training, validation, and testing methods to spotpython.

```
from spotpython.fun.hypersklearn import HyperSklearn
fun = HyperSklearn().fun_sklearn
```

The following code snippet shows how to get the default hyperparameters as an array, so that they can be passed to the Spot function.

```
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array
X_start = get_default_hyperparameters_as_array(fun_control)
```

23.6.2. Run the Spot Optimizer

The class Spot [SOURCE] is the hyperparameter tuning workhorse. It is initialized with the following parameters:

- `fun`: the objective function
- `fun_control`: the dictionary with the control parameters for the objective function
- `design`: the experimental design
- `design_control`: the dictionary with the control parameters for the experimental design
- `surrogate`: the surrogate model
- `surrogate_control`: the dictionary with the control parameters for the surrogate model
- `optimizer`: the optimizer
- `optimizer_control`: the dictionary with the control parameters for the optimizer

> **Note: Total run time**
>
> The total run time may exceed the specified `max_time`, because the initial design (here: `init_size` = INIT_SIZE as specified above) is always evaluated, even if this takes longer than `max_time`.

```
from spotpython.utils.init import design_control_init, surrogate_control_init
design_control = design_control_init()
set_control_key_value(control_dict=design_control,
                        key="init_size",
                        value=INIT_SIZE,
                        replace=True)

surrogate_control = surrogate_control_init(noise=True,
                                           n_theta=2)
from spotpython.spot import Spot
spot_tuner = Spot(fun=fun,
                  fun_control=fun_control,
                  design_control=design_control,
                  surrogate_control=surrogate_control)
spot_tuner.run(X_start=X_start)
```

```
spotpython tuning: 0.11831336203046443 [----------] 0.63%
spotpython tuning: 0.11831336203046443 [----------] 1.32%
spotpython tuning: 0.11831336203046443 [----------] 1.97%
spotpython tuning: 0.11831336203046443 [----------] 2.79%
spotpython tuning: 0.11467654823166384 [----------] 3.68%
spotpython tuning: 0.11467654823166384 [----------] 4.65%
spotpython tuning: 0.1052243056750549 [#---------] 5.97%
spotpython tuning: 0.10405556663192285 [#---------] 7.38%
spotpython tuning: 0.10405556663192285 [#---------] 8.92%
```

```
spotpython tuning: 0.08698965578693191 [#---------] 10.43%
spotpython tuning: 0.08423427702103133 [#---------] 12.20%
spotpython tuning: 0.08423427702103133 [#---------] 13.60%
spotpython tuning: 0.08423427702103133 [##--------] 15.02%
spotpython tuning: 0.08423427702103133 [##--------] 16.55%
spotpython tuning: 0.08423427702103133 [##--------] 18.15%
spotpython tuning: 0.08423427702103133 [##--------] 19.67%
spotpython tuning: 0.08423427702103133 [##--------] 21.16%
spotpython tuning: 0.08423427702103133 [##--------] 22.83%
spotpython tuning: 0.08423427702103133 [##--------] 24.35%
spotpython tuning: 0.08423427702103133 [###-------] 25.88%
spotpython tuning: 0.08423427702103133 [###-------] 27.62%
spotpython tuning: 0.08423427702103133 [###-------] 29.31%
spotpython tuning: 0.08423427702103133 [###-------] 30.66%
spotpython tuning: 0.08423427702103133 [###-------] 32.28%
spotpython tuning: 0.08423427702103133 [###-------] 33.92%
spotpython tuning: 0.08423427702103133 [####------] 35.53%
spotpython tuning: 0.08423427702103133 [####------] 37.13%
spotpython tuning: 0.08423427702103133 [####------] 38.87%
spotpython tuning: 0.08423427702103133 [####------] 40.77%
spotpython tuning: 0.08423427702103133 [####------] 42.71%
spotpython tuning: 0.08423427702103133 [####------] 44.51%
spotpython tuning: 0.08423427702103133 [#####-----] 46.05%
spotpython tuning: 0.08423427702103133 [#####-----] 47.73%
spotpython tuning: 0.08423427702103133 [#####-----] 49.66%
spotpython tuning: 0.08423427702103133 [#####-----] 51.35%
spotpython tuning: 0.08423427702103133 [#####-----] 53.19%
spotpython tuning: 0.08423427702103133 [#####-----] 54.98%
spotpython tuning: 0.08423427702103133 [######----] 56.70%
spotpython tuning: 0.08423427702103133 [######----] 58.68%
spotpython tuning: 0.08423427702103133 [######----] 60.37%
spotpython tuning: 0.08423427702103133 [######----] 62.22%
spotpython tuning: 0.08423427702103133 [######----] 63.80%
spotpython tuning: 0.08423427702103133 [#######---] 65.56%
spotpython tuning: 0.08423427702103133 [#######---] 67.00%
spotpython tuning: 0.08423427702103133 [#######---] 68.76%
spotpython tuning: 0.08423427702103133 [#######---] 70.49%
spotpython tuning: 0.08423427702103133 [#######---] 72.03%
spotpython tuning: 0.08423427702103133 [#######---] 73.79%
spotpython tuning: 0.08423427702103133 [########--] 75.64%
spotpython tuning: 0.08423427702103133 [########--] 77.33%
spotpython tuning: 0.08423427702103133 [########--] 79.04%
spotpython tuning: 0.08423427702103133 [########--] 80.94%
spotpython tuning: 0.08423427702103133 [########--] 82.76%
spotpython tuning: 0.08423427702103133 [########--] 84.39%
```

```
spotpython tuning: 0.08423427702103133 [#########-] 86.03%
spotpython tuning: 0.08423427702103133 [#########-] 87.77%
spotpython tuning: 0.08423427702103133 [#########-] 89.46%
spotpython tuning: 0.08423427702103133 [#########-] 91.25%
spotpython tuning: 0.08423427702103133 [#########-] 93.06%
spotpython tuning: 0.08423427702103133 [##########] 95.02%
spotpython tuning: 0.08423427702103133 [##########] 96.86%
spotpython tuning: 0.08423427702103133 [##########] 98.38%
spotpython tuning: 0.08423427702103133 [##########] 99.62%
spotpython tuning: 0.08423427702103133 [##########] 100.00% Done...

Experiment saved to 18_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x138ef4fb0>
```

23.6.3. TensorBoard

Now we can start TensorBoard in the background with the following command, where `./runs` is the default directory for the TensorBoard log files:

```
tensorboard --logdir="./runs"
```

```
from spotpython.utils.init import get_tensorboard_path
get_tensorboard_path(fun_control)
```

```
'runs/'
```

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress(log_y=True)
```

Results can also be printed in tabular form.

```
print_res_table(spot_tuner)
```

```
| name        | type   | default    |    lower |    upper | tuned               | transform
|-------------|--------|------------|----------|----------|---------------------|----------
| C           | float  | 1.0        |      0.1 |     10.0 | 5.0858474654131784  | None
| kernel      | factor | rbf        |      0.0 |      0.0 | rbf                 | None
| degree      | int    | 3          |      3.0 |      3.0 | 3.0                 | None
| gamma       | factor | scale      |      0.0 |      1.0 | scale               | None
| coef0       | float  | 0.0        |      0.0 |      0.0 | 0.0                 | None
| epsilon     | float  | 0.1        |      0.1 |      1.0 | 0.1                 | None
| shrinking   | factor | 0          |      0.0 |      1.0 | 1                   | None
| tol         | float  | 0.001      |    1e-05 |    0.001 | 0.001               | None
| cache_size  | float  | 200.0      |    100.0 |    400.0 | 331.7133108917936   | None
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=0.0025)
```

23.7. Get Default Hyperparameters

The default hyperparameters, which will be used for a comparion with the tuned hyperparameters, can be obtained with the following commands:

```
from spotpython.hyperparameters.values import get_one_core_model_from_X
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array
X_start = get_default_hyperparameters_as_array(fun_control)
model_default = get_one_core_model_from_X(X_start, fun_control, default=True)
model_default
```

```
SVR(cache_size=200.0, shrinking=False)
```

23.8. Get SPOT Results

In a similar way, we can obtain the hyperparameters found by `spotpython`.

```
from spotpython.hyperparameters.values import get_one_core_model_from_X
X_tuned = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
model_spot = get_one_core_model_from_X(X_tuned, fun_control)
```

23.8.1. Plot: Compare Predictions

```
model_default.fit(X_train, y_train)
y_default = model_default.predict(X_plot)
```

```
model_spot.fit(X_train, y_train)
y_spot = model_spot.predict(X_plot)
```

```
import matplotlib.pyplot as plt
plt.scatter(X[:100], y[:100], c="orange", label="data", zorder=1, edgecolors=(0, 0, 0)
plt.plot(
    X_plot,
    y_default,
    c="red",
    label="Default SVR")

plt.plot(
    X_plot, y_spot, c="blue", label="SPOT SVR")

plt.xlabel("data")
plt.ylabel("target")
plt.title("SVR")
_ = plt.legend()
```

23.8.2. Detailed Hyperparameter Plots

```
spot_tuner.plot_important_hyperparameter_contour(filename=None)
```

```
C:  0.013507866153520925
gamma:  0.015571687260535715
epsilon:  38.11920850020238
shrinking:  0.05498187197095143
tol:  100.0
cache_size:  0.005229658202911702
```

x4: tol
x2: epsilon
x3: shrinking
x5: cache_size

x5: cache_size
x3: shrinking

x3: shrinking
x5: cache_size

x4: tol
x1: gamma

x1: gamma
x4: tol

x2: epsilon
x1: gamma

x1: gamma
x2: epsilon

x3: shrinking
x1: gamma

x1: gamma
x3: shrinking

x5: cache_size
x1: gamma
x4: tol
x0: C
x2: epsilon
x3: shrinking

23.8.3. Parallel Coordinates Plot

```
spot_tuner.parallel_plot()
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

Part VII.

Hyperparameter Tuning with River

24. HPT: River

24.1. Introduction to River

25. Simplifying Hyperparameter Tuning in Online Machine Learning—The spotRiverGUI

25.1. Introduction

Batch Machine Learning (BML) often encounters limitations when processing substantial volumes of streaming data (Keller-McNulty 2004; Gaber, Zaslavsky, and Krishnaswamy 2005; Aggarwal 2007). These limitations become particularly evident in terms of available memory, managing drift in data streams (Bifet and Gavaldà 2007, 2009; Gama et al. 2004; Bartz-Beielstein 2024c), and processing novel, unclassified data (Bifet 2010), (Dredze, Oates, and Piatko 2010). As a solution, Online Machine Learning (OML) serves as an effective alternative to BML, adeptly addressing these constraints. OML's ability to sequentially process data proves especially beneficial for handling data streams (Bifet et al. 2010a; Masud et al. 2011; Gama, Sebastião, and Rodrigues 2013; Putatunda 2021; Bartz-Beielstein and Hans 2024).

The Online Machine Learning (OML) methods provided by software packages such as `river` (Montiel et al. 2021) or `MOA` (Bifet et al. 2010b) require the specification of many hyperparameters. To give an example, Hoeffding trees (Hoeglinger and Pears 2007), which are very popular in OML, offer a variety of "splitters" to generate subtrees. There are also several methods to limit the tree size, ensuring time and memory requirements remain manageable. Given the multitude of parameters, manually searching for the optimal hyperparameter setting can be a daunting and often futile task due to the complexity of possible combinations. This article elucidates how automatic hyperparameter optimization, or "tuning", can be achieved. Beyond optimizing the OML process, Hyperparameter Tuning (HPT) executed with the Sequential Parameter Optimization Toolbox (SPOT) enhances the explainability and interpretability of OML procedures. This can result in a more efficient, resource-conserving algorithm, contributing to the concept of "Green AI".

> **Note**
>
> Note: This document refers to `spotRiverGUI` version 0.0.26 which was released on Feb 18, 2024 on GitHub, see: https://github.com/sequential-parameter-

> optimization/spotGUI/tree/main. The GUI is under active development and new features will be added soon.

This article describes the `spotRiverGUI`, which is a graphical user interface for the `spotriver` package. The GUI allows the user to select the task, the data set, the preprocessing model, the metric, and the online machine learning model. The user can specify the experiment duration, the initial design, and the evaluation options. The GUI provides information about the data set and allows the user to save and load experiments. It also starts and stops a tensorboard process to observe the tuning online and provides an analysis of the hyperparameter tuning process. The `spotRiverGUI` releases the user from the burden of manually searching for the optimal hyperparameter setting. After providing the data, users can compare different OML algorithms from the powerful `river` package in a convenient way and tune the selected algorithm very efficiently.

This article is structured as follows:

Section 25.2 describes how to install the software. It also explains how the `spotRiverGUI` can be started. Section 25.3 describes the binary classification task and the options available in the `spotRiverGUI`. Section 25.4 provides information about the planned regression task. Section 25.5 describes how the data can be visualized in the `spotRiverGUI`. Section 25.6 provides information about saving and loading experiments. Section 25.7 describes how to start an experiment and how the associated tensorboard process can be started and stopped. Section 25.8 provides information about the analysis of the results from the hyperparameter tuning process. Section 25.9 concludes the article and provides an outlook.

25.2. Installation and Starting

25.2.1. Installation

We strongly recommend using a virtual environment for the installation of the `river`, `spotriver`, `build` and `spotRiverGUI` packages.

Miniforge, which holds the minimal installers for Conda, is a good starting point. Please follow the instructions on https://github.com/conda-forge/miniforge. Using Conda, the following commands can be used to create a virtual environment (Python 3.11 is recommended):

```
>> conda create -n myenv python=3.11
>> conda activate myenv
```

Now the `river` and `spotriver` packages can be installed:

```
>> (myenv) pip install river spotriver build
```

Although the `spotGUI` package is available on PyPI, we recommend an installation from the GitHub repository https://github.com/sequential-parameter-optimization/spotGUI, because the `spotGUI` package is under active development and new features will be added soon. The installation from the GitHub repository is done by executing the following command:

```
>> (myenv) git clone git@github.com:sequential-parameter-optimization/spotGUI.git
```

Building the `spotGUI` package is done by executing the following command:

```
>> (myenv) cd spotGUI
>> (myenv) python -m build
```

Now the `spotRiverGUI` package can be installed:

```
>> (myenv) pip install dist/spotGUI-0.0.26.tar.gz
```

25.2.2. Starting the GUI

The GUI can be started by executing the `spotRiverGUI.py` file in the `spotGUI/spotRiverGUI` directory. Change to the `spotRiverGUI` directory and start the GUI:

```
>> (myenv) cd spotGUI/spotRiverGUI
>> (myenv) python spotRiverGUI.py
```

The GUI window will open, as shown in Figure 25.1.

After the GUI window has opened, the user can select the task. Currently, `Binary Classification` is available. Further tasks like `Regression` will be available soon.

Depending on the task, the user can select the data set, the preprocessing model, the metric, and the online machine learning model.

25.3. Binary Classification

25.3.1. Binary Classification Options

If the `Binary Classification` task is selected, the user can select pre-specified data sets from the `Data` drop-down menu.

Figure 25.1.: spotriver GUI

25.3.1.1. River Data Sets

The following data sets from the `river` package are available (the descriptions are taken from the `river` package):

- `Bananas`: An artificial dataset where instances belongs to several clusters with a banana shape.There are two attributes that correspond to the x and y axis, respectively. More: https://riverml.xyz/dev/api/datasets/Bananas/.
- `CreditCard`: Credit card frauds. The datasets contains transactions made by credit cards in September 2013 by European cardholders. Feature '`Class`' is the response variable and it takes value 1 in case of fraud and 0 otherwise. More: https://riverml.xyz/dev/api/datasets/CreditCard/.
- `Elec2`: Electricity prices in New South Wales. This is a binary classification task, where the goal is to predict if the price of electricity will go up or down. This data was collected from the Australian New South Wales Electricity Market. In this market, prices are not fixed and are affected by demand and supply of the market. They are set every five minutes. Electricity transfers to/from the neighboring state of Victoria were done to alleviate fluctuations. More: https://riverml.xyz/dev/api/datasets/Elec2/.
- `Higgs`: The data has been produced using Monte Carlo simulations. The first 21 features (columns 2-22) are kinematic properties measured by the particle detectors in the accelerator. The last seven features are functions of the first 21 features; these are high-level features derived by physicists to help discriminate between the two classes. More: https://riverml.xyz/dev/api/datasets/Higgs/.
- `HTTP`: HTTP dataset of the KDD 1999 cup. The goal is to predict whether or not an HTTP connection is anomalous or not. The dataset only contains 2,211 (0.4%) positive labels. More: https://riverml.xyz/dev/api/datasets/HTTP/.

- `Phishing`: Phishing websites. This dataset contains features from web pages that are classified as phishing or not.https://riverml.xyz/dev/api/datasets/Phishing/

25.3.1.2. User Data Sets

Besides the `river` data sets described in Section 25.3.1.1, the user can also select a user-defined data set. Currently, comma-separated values (CSV) files are supported. Further formats will be supported soon. The user-defined CSV data set must be a binary classification task with the target variable in the last column. The first row must contain the column names. If the file is copied to the subdirectory `userData`, the user can select the data set from the `Data` drop-down menu.

As an example, we have provided a CSV-version of the `Phishing` data set. The file is located in the `userData` subdirectory and is called `PhishingData.csv`. It contains the columns `empty_server_form_handler`, `popup_window`, `https`, `request_from_other_domain`, `anchor_from_other_domain`, `is_popular`, `long_url`, `age_of_domain`, `ip_in_url`, and `is_phishing`. The first few lines of the file are shown below (modified due to formatting reasons):

```
empty_server_form_handler,...,is_phishing
0.0,0.0,0.0,0.0,0.0,0.5,1.0,1,1,1
1.0,0.0,0.5,0.5,0.0,0.5,0.0,1,0,1
0.0,0.0,1.0,0.0,0.5,0.5,0.0,1,0,1
0.0,0.0,1.0,0.0,0.0,1.0,0.5,0,0,1
```

Based on the required format, we can see that `is_phishing` is the target column, because it is the last column of the data set.

25.3.1.3. Stream Data Sets

Forthcoming versions of the GUI will support stream data sets, e.g, the Friedman-Drift generator (Ikonomovska 2012) or the SEA-Drift generator (Street and Kim 2001). The Friedman-Drift generator was also used in the hyperparameter tuning study in Bartz-Beielstein (2024b).

25.3.1.4. Data Set Options

Currently, the user can select the following parameters for the data sets:

- `n_total`: The total number of instances. Since some data sets are quite large, the user can select a subset of the data set by specifying the `n_total` value.
- `test_size`: The size of the test set in percent (`0.0 - 1.0`). The training set will be `1.0 - test_size`.

The target column should be the last column of the data set. Future versions of the GUI will support the selection of the `target_column` from the GUI. Currently, the value from the field `target_column` has not effect.

To compare different data scaling methods, the user can select the preprocessing model from the `Preprocessing` drop-down menu. Currently, the following preprocessing models are available:

- `StandardScaler`: Standardize features by removing the mean and scaling to unit variance.
- `MinMaxScaler`: Scale features to a range.
- `None`: No scaling is performed.

The `spotRiverGUI` will not provide sophisticated data preprocessing methods. We assume that the data was preprocessed before it is copied into the `userData` subdirectory.

25.3.2. Experiment Options

Currently, the user can select the following options for specifying the experiment duration:

- `MAX_TIME`: The maximum time in minutes for the experiment.
- `FUN_EVALS`: The number of function evaluations for the experiment. This is the number of OML-models that are built and evaluated.

If the `MAX_TIME` is reached or `FUN_EVALS` OML models are evaluated, the experiment will be stopped.

> **Initial design is always evaluated**
>
> - The initial design will always be evaluated before one of the stopping criteria is reached.
> - If the initial design is very large or the model evaluations are very time-consuming, the runtime will be larger than the `MAX_TIME` value.

Based on the `INIT_SIZE`, the number of hyperparameter configurations for the initial design can be specified. The initial design is evaluated before the first surrogate model is built. A detailed description of the initial design and the surrogate model based hyperparameter tuning can be found in Bartz-Beielstein (2024a) and in Bartz-Beielstein and Zaefferer (2022). The `spotpython` package is used for the hyperparameter tuning process. It implements a robust surrogate model based optimization method (Forrester, Sóbester, and Keane 2008).

The `PREFIX` parameter can be used to specify the experiment name.

The `spotpython` hyperparameter tuning program allows the user to specify several options for the hyperparameter tuning process. The `spotRiverGUI` will support more options in future versions. Currently, the user can specify whether the outcome from the experiment is noisy or deterministic. The corresponding parameter is called `NOISE`. The reader is referred to Bartz-Beielstein (2024b) and to the chapter "Handling Noise" (https://sequential-parameter-optimization.github.io/Hyperparameter-Tuning-Cookbook/013_num_spot_noisy.html) for further information about the `NOISE` parameter.

25.3.3. Evaluation Options

The user can select one of the following evaluation metrics for binary classification tasks from the `metric` drop-down menu:

- `accuracy_score`
- `cohen_kappa_score`
- `f1_score`
- `hamming_loss`
- `hinge_loss`
- `jaccard_score`
- `matthews_corrcoef`
- `precision_score`
- `recall_score`
- `roc_auc_score`
- `zero_one_loss`

These metrics are based on the `scikit-learn` module (Pedregosa et al. 2011), which implements several loss, score, and utility functions to measure classification performance, see https://scikit-learn.org/stable/modules/model_evaluation.html#classification-metrics. `spotRiverGUI` supports metrics that are computed from the `y_pred` and the `y_true` values. The `y_pred` values are the predicted target values, and the `y_true` values are the true target values. The `y_pred` values are generated by the online machine learning model, and the `y_true` values are the true target values from the data set.

> **Evaluation Metrics: Minimization and Maximization**
>
> - Some metrics are minimized, and some are maximized. The `spotRiverGUI` will support the user in selecting the correct metric based on the task. For example, the `accuracy_score` is maximized, and the `hamming_loss` is minimized. The user can select the metric and `spotRiverGUI` will automatically determine whether the metric is minimized or maximized.

In addition to the evaluation metric results, `spotriver` considers the time and memory consumption of the online machine learning model. The `spotRiverGUI` will support the user in selecting the time and memory consumption as additional evaluation metrics. By modifying the weight vector, which is shown in the `weights: y, time, mem` field, the user can specify the importance of the evaluation metrics. For example, the weight vector `1,0,0` specifies that only the `y` metric (e.g., accuracy) is considered. The weight vector `0,1,0` specifies that only the time metric is considered. The weight vector `0,0,1` specifies that only the memory metric is considered. The weight vector `1,1,1` specifies that all metrics are considered. Any real values (also negative ones) are allowed for the weights.

> **The weight vector**
>
> - The specification of adequate weights is highly problem dependent.
> - There is no generic setting that fits to all problems.

As described in Bartz-Beielstein (2024a), a prediction horizon is used for the comparison of the online-machine learning algorithms. The `horizon` can be specified in the `spotRiverGUI` by the user and is highly problem dependent. The `spotRiverGUI` uses the `eval_oml_horizon` method from the `spotriver` package, which evaluates the online-machine learning model on a rolling horizon basis.

In addition to the `horizon` value, the user can specify the `oml_grace_period` value. During the `oml_grace_period`, the OML-model is trained on the (small) training data set. No predictions are made during this initial training phase, but the memory and computation time are measured. Then, the OML-model is evaluated on the test data set using a given (sklearn) evaluation metric. The default value of the `oml_grace_period` is `horizon`. For convenience, the value `horizon` is also selected when the user specifies the `oml_grace_period` value as `None`.

> **The oml_grace_period**
>
> - If the `oml_grace_period` is set to the size of the training data set, the OML-model is trained on the entire training data set and then evaluated on the test data set using a given (sklearn) evaluation metric.
> - This setting might be "unfair" in some cases, because the OML-model should learn online and not on the entire training data set.
> - Therefore, a small data set is recommended for the `oml_grace_period` setting and the prediction `horizon` is a recommended value for the `oml_grace_period` setting. The reader is referred to Bartz-Beielstein (2024a) for further information about the `oml_grace_period` setting.

25.3.4. Online Machine Learning Model Options

The user can select one of the following online machine learning models from the `coremodel` drop-down menu:

- `forest.AMFClassifier`: Aggregated Mondrian Forest classifier for online learning (Mourtada, Gaiffas, and Scornet 2019). This implementation is truly online, in the sense that a single pass is performed, and that predictions can be produced anytime. More: https://riverml.xyz/dev/api/forest/AMFClassifier/.
- `tree.ExtremelyFastDecisionTreeClassifier`: Extremely Fast Decision Tree (EFDT) classifier (Manapragada, Webb, and Salehi 2018). Also referred to as the Hoeffding AnyTime Tree (HATT) classifier. In practice, despite the name, EFDTs are typically slower than a vanilla Hoeffding Tree to process data. More: https://riverml.xyz/dev/api/tree/ExtremelyFastDecisionTreeClassifier/.
- `tree.HoeffdingTreeClassifier`: Hoeffding Tree or Very Fast Decision Tree classifier (Bifet et al. 2010a; Domingos and Hulten 2000). More: https://riverml.xyz/dev/api/tree/HoeffdingTreeClassifier/.
- `tree.HoeffdingAdaptiveTreeClassifier`: Hoeffding Adaptive Tree classifier (Bifet and Gavaldà 2009). More: https://riverml.xyz/dev/api/tree/HoeffdingAdaptiveTreeClassifier/.
- `linear_model.LogisticRegression`: Logistic regression classifier. More: hhttps://riverml.xyz/dev/api/linear-model/LogisticRegression/.

The `spotRiverGUI` automatically determines the hyperparameters for the selected online machine learning model and adapts the input fields to the model hyperparameters. The user can modify the hyperparameters in the GUI. Figure 25.2 shows the `spotRiverGUI` when the `forest.AMFClassifier` is selected and Figure 25.3 shows the `spotRiverGUI` when the `tree.HoeffdingTreeClassifier` is selected.

Figure 25.2.: `spotRiverGUI` when `forest.AMFClassifier` is selected

Numerical and categorical hyperparameters are treated differently in the `spotRiverGUI`:

Figure 25.3.: `spotRiverGUI` when `tree.HoeffdingAdaptiveTreeClassifier` is selected

- The user can modify the lower and upper bounds for the numerical hyperparameters.
- There are no upper or lower bounds for categorical hyperparameters. Instead, hyperparameter values for the categorical hyperparameters are considered as sets of values, e.g., the set of `ExhaustiveSplitter`, `HistogramSplitter`, `GaussianSplitter` is provided for the `splitter` hyperparameter of the `tree.HoeffdingAdaptiveTreeClassifier` model as can be seen in Figure 25.3. The user can select the full set or any subset of the set of values for the categorical hyperparameters.

In addition to the lower and upper bounds (or the set of values for the categorical hyperparameters), the `spotRiverGUI` provides information about the `Default values` and the `Transformation` function. If the `Transformation` function is set to `None`, the values of the hyperparameters are passed to the `spot` tuner as they are. If the `Transformation` function is set to `transform_power_2_int`, the value x is transformed to 2^x before it is passed to the `spot` tuner.

Modifications of the `Default values` and `Transformation` functions values in the `spotRiverGUI` have no effect on the hyperparameter tuning process. This is intensional. In future versions, the user will be able to add their own hyperparameter dictionaries to the `spotRiverGUI`, which allows the modification of `Default values` and `Transformation` functions values. Furthermore, the `spotRiverGUI` will support more online machine learning models in future versions.

25.4. Regression

Regression tasks will be supported soon. The same workflow as for the binary classification task will be used, i.e., the user can select the data set, the preprocessing model, the metric, and the online machine learning model.

25.5. Showing the Data

The `spotRiverGUI` provides the `Show Data` button, which opens a new window and shows information about the data set. The first figure (Figure 25.4) shows histograms of the target variables in the train and test data sets. The second figure (Figure 25.5) shows scatter plots of the features in the train data set. The third figure (Figure 25.6) shows the corresponding scatter plots of the features in the test data set.

Figure 25.4.: Output from the `spotRiverGUI` when `Bananas` data is selected for the `Show Data` option

> **Size of the Displayed Data Sets**
>
> - Some data sets are quite large and the display of the data sets might take some time.
> - Therefore, a random subset of 1000 instances of the data set is displayed if the data set is larger than 1000 instances.

Showing the data is important, especially for the new / unknown data sets as can be seen in Figure 25.7, Figure 25.8, and Figure 25.9: The target variable is highly biased. The user can check whether the data set is correctly formatted and whether the target variable is correctly specified.

Figure 25.5.: Visualization of the train data. Output from the `spotRiverGUI` when `Bananas` data is selected for the `Show Data` option

Figure 25.6.: Visualization of the test data. Output from the `spotRiverGUI` when `Bananas` data is selected for the `Show Data` option

Figure 25.7.: Output from the `spotRiverGUI` when `HTTP` data is selected for the `Show Data` option. The target variable is biased.

Figure 25.8.: Output from the `spotRiverGUI` when `HTTP` data is selected for the `Show Data` option. A subset of 1000 randomly chosen data points is shown. Only a few positive events are in the data.

Figure 25.9.: Output from the `spotRiverGUI` when `HTTP` data is selected for the `Show Data` option. The test data set shows the same structure as the train data set.

In addition to the histograms and scatter plots, the `spotRiverGUI` provides textual information about the data set in the console window. e.g., for the `Bananas` data set, the following information is shown:

```
Train data summary:
                 x1           x2            y
count  3710.000000  3710.000000  3710.000000
mean     -0.016243     0.002430     0.451482
std       0.995490     1.001150     0.497708
min      -3.089839    -2.385937     0.000000
25%      -0.764512    -0.914144     0.000000
50%      -0.027259    -0.033754     0.000000
75%       0.745066     0.836618     1.000000
max       2.754447     2.517112     1.000000

Test data summary:
                 x1           x2            y
count  1590.000000  1590.000000  1590.000000
mean      0.037900    -0.005670     0.440881
std       1.009744     0.997603     0.496649
min      -2.980834    -2.199138     0.000000
25%      -0.718710    -0.911151     0.000000
```

```
50%        0.034858    -0.046502     0.000000
75%        0.862049     0.806506     1.000000
max        2.813360     3.194302     1.000000
```

25.6. Saving and Loading

25.6.1. Saving the Experiment

If the experiment should not be started immediately, the user can save the experiment by clicking on the `Save Experiment` button. The `spotRiverGUI` will save the experiment as a pickle file. The file name is generated based on the `PREFIX` parameter. The pickle file contains a set of dictionaries, which are used to start the experiment.

`spotRiverGUI` shows a summary of the selected hyperparameters in the console window as can be seen in Table 25.1.

Table 25.1.: The hyperparameter values for the `tree.HoeffdingAdaptiveTreeClassifier` model.

name	type	default	lower	upper	transform
grace_period	int	200	10	1000	None
max_depth	int	20	2	20	transform_power_2_int
delta	float	1e-07	1e-08	1e-06	None
tau	float	0.05	0.01	0.1	None
leaf_prediction	factor	nba	0	2	None
nb_threshold	int	0	0	10	None
splitter	factor	GaussianSplitter	0	2	None
bootstrap_sampling	factor	0	0	1	None
drift_window_threshold	int	300	100	500	None
drift_detector	factor	ADWIN	0	0	None
switch_significance	float	0.05	0.01	0.1	None
binary_split	factor	0	0	1	None
max_size	float	100.0	100	1000	None
memory_estimate_period	int	1000000	100000	1e+06	None
stop_mem_management	factor	0	0	1	None
remove_poor_attrs	factor	0	0	1	None
merit_preprune	factor	0	0	1	None

25.6.2. Loading an Experiment

Future versions of the `spotRiverGUI` will support the loading of experiments from the GUI. Currently, the user can load the experiment by executing the command `load_experiment`, see https://sequential-parameter-optimization.github.io/spotpython/reference/spotpython/utils/file/#spotpython.utils.file.load_experiment.

25.7. Running a New Experiment

An experiment can be started by clicking on the `Run Experiment` button. The GUI calls `run_spot_python_experiment` from `spotGUI.tuner.spotRun`. Output will be shown in the console window from which the GUI was started.

25.7.1. Starting and Stopping Tensorboard

Tensorboard (Abadi et al. 2016) is automatically started when an experiment is started. The tensorboard process can be observed in a browser by opening the http://localhost:6006 page. Tensorboard provides a visual representation of the hyperparameter tuning process. Figure 25.10 and Figure 25.11 show the tensorboard page when the `spotRiverGUI` is performing the tuning process.

`spotpython.utils.tensorboard` provides the methods `start_tensorboard` and `stop_tensorboard` to start and stop tensorboard as a background process. After the experiment is finished, the tensorboard process is stopped automatically.

25.8. Performing the Analysis

If the hyperparameter tuning process is finished, the user can analyze the results by clicking on the `Analysis` button. The following options are available:

- Progress plot
- Compare tuned versus default hyperparameters
- Importance of hyperparameters
- Contour plot
- Parallel coordinates plot

Figure 25.12 shows the progress plot of the hyperparameter tuning process. Black dots denote results from the initial design. Red dots illustrate the improvement found by the surrogate model based optimization. For binary classification tasks, the `roc_auc_score` can be used as the evaluation metric. The confusion matrix is shown

Figure 25.10.: Tensorboard visualization of the hyperparameter tuning process

Figure 25.11.: Tensorboard. Parallel coordinates plot

in Figure 25.13. The default versus tuned hyperparameters are shown in Figure 25.14. The surrogate plot is shown in Figure 25.15, Figure 25.16, and Figure 25.17.

Figure 25.12.: Progress plot of the hyperparameter tuning process

Figure 25.13.: Confusion matrix

Furthermore, the tuned hyperparameters are shown in the console window. A typical output is shown below (modified due to formatting reasons):

```
|name    |type   |default |low | up |tuned |transf |importance|stars|
|--------|-------|--------|----|----|------|-------|----------|-----|
|n_estim |int    |    3.0 |2.0 |7.0 |  3.0 | pow_2 |      0.04|     |
|step    |float  |    1.0 |0.1 |10.0|  5.12| None  |      0.21| .   |
|use_agg |factor |    1.0 |0.0 |1.0 |  0.0 | None  |     10.17| *   |
|dirichl |float  |    0.5 |0.1 |0.75|  0.37| None  |     13.64| *   |
|split_p |factor |    0.0 |0.0 |1.0 |  0.0 | None  |    100.00| *** |
```

In addition to the tuned parameters that are shown in the column `tuned`, the columns `importance` and `stars` are shown. Both columns show the most important hyperparameters based on information from the surrogate model. The `stars` column shows the importance of the hyperparameters in a graphical way. It is important to note that

Figure 25.14.: Default versus tuned hyperparameters

Figure 25.15.: Surrogate plot based on the Kriging model. `x0` and `x1` plotted against each other.

Figure 25.16.: Surrogate plot based on the Kriging model. `x1` and `x2` plotted against each other.

Figure 25.17.: Surrogate plot based an the Kriging model. `x0` and `x2` plotted against each other.

the results are based on a demo of the hyperparameter tuning process. The plots are not based on a real hyperparameter tuning process. The reader is referred to Bartz-Beielstein (2024b) for further information about the analysis of the hyperparameter tuning process.

25.9. Summary and Outlook

The `spotRiverGUI` provides a graphical user interface for the `spotriver` package. It releases the user from the burden of manually searching for the optimal hyperparameter setting. After copying a data set into the `userData` folder and starting `spotRiverGUI`, users can compare different OML algorithms from the powerful `river` package in a convenient way. Users can generate configurations on their local machines, which can be transferred to a remote machine for execution. Results from the remote machine can be copied back to the local machine for analysis.

! Benefits of the spotRiverGUI:

- Very easy to use (only the data must be provided in the correct format).
- Reproducible results.
- State-of-the-art hyperparameter tuning methods.
- Powerful analysis tools, e.g., Bayesian optimization (Forrester, Sóbester, and Keane 2008; Gramacy 2020).
- Visual representation of the hyperparameter tuning process with tensorboard.
- Most advanced online machine learning models from the `river` package.

The `river` package (Montiel et al. 2021), which is very well documented, can be downloaded from https://riverml.xyz/latest/.

The `spotRiverGUI` is under active development and new features will be added

soon. It can be downloaded from GitHub: https://github.com/sequential-parameter-optimization/spotGUI.

Interactive Jupyter Notebooks and further material about OML are provided in the GitHub repository https://github.com/sn-code-inside/online-machine-learning. This material is part of the supplementary material of the book "Online Machine Learning - A Practical Guide with Examples in Python", see https://link.springer.com/book/9789819970063 and the forthcoming book "Online Machine Learning - Eine praxisorientierte Einführung", see https://link.springer.com/book/9783658425043.

25.10. Appendix

25.10.1. Adding new Tasks

Currently, three tasks are supported in the `spotRiverGUI`: `Binary Classification`, `Regression`, and `Rules`. `Rules` was added in ver 0.6.0. Here, we document how this task updated was implemented. Adding an additional task requires modifications in the following files:

- `spotRun.py`:
  - The `river`class `rules` must be imported, i.e., `from river import forest, tree, linear_model, rules`.
  - The method `get_river_rules_core_model_names()` must be modified.
  - The `get_scenario_dict()` method must be modified.
- `CTk.py`:
  - The `task_frame` must be extended.
  - The `change_task_event()` method must be modified.

In addition, the hyperparameter dictionary in `spotriver` must be updated. This is the only modification required in the `spotriver`package.

26. river Hyperparameter Tuning: Hoeffding Tree Regressor with Friedman Drift Data

This chapter demonstrates hyperparameter tuning for river's Hoeffding Tree Regressor (HTR) with the Friedman drift data set [SOURCE]. The Hoeffding Tree Regressor is a regression tree that uses the Hoeffding bound to limit the number of splits evaluated at each node, i.e., it predicts a real value for each sample.

26.1. The Friedman Drift Data Set

We will use the Friedman synthetic dataset with concept drifts, which is described in detail in Section E.2. The following parameters are used to generate and handle the data set:

- position: The positions of the concept drifts.
- n_train: The number of samples used for training.
- n_test: The number of samples used for testing.
- seed: The seed for the random number generator.
- target_column: The name of the target column.
- drift_type: The type of the concept drift.

We will use spotriver's convert_to_df function [SOURCE] to convert the river data set to a pandas data frame. Then we add column names x1 until x10 to the first 10 columns of the dataframe and the column name y to the last column of the dataframe.

This data generation is independently repeated for the training and test data sets, because the data sets are generated with concept drifts and the usual train-test split would not work.

```
from river.datasets import synth
import pandas as pd
import numpy as np
from spotriver.utils.data_conversion import convert_to_df
```

```
n_train = 6_000
n_test = 4_000
n_samples = n_train + n_test
target_column = "y"

dataset = synth.FriedmanDrift(
   drift_type='gra',
   position=(n_train/4, n_train/2),
   seed=123
)

train = convert_to_df(dataset, n_total=n_train)
train.columns = [f"x{i}" for i in range(1, 11)] + [target_column]
```

```
dataset = synth.FriedmanDrift(
   drift_type='gra',
   position=(n_test/4, n_test/2),
   seed=123
)
test = convert_to_df(dataset, n_total=n_test)
test.columns = [f"x{i}" for i in range(1, 11)] + [target_column]
```

> **The Data Set**
>
> Data sets that are available as `pandas` dataframes can easily be passed to the `spot` hyperparameter tuner. `spotpython` requires a `train` and a `test` data set, where the column names must be identical.

We combine the train and test data sets and save them to a csv file.

```
df = pd.concat([train, test])
df.to_csv("./userData/friedman.csv", index=False)
```

The Friedman Drift data set described in this section is avaialble as a `csv` data file and can be downloaded from github: friedman.csv.

26.2. Setup

26.2.1. General Experiment Setup

To keep track of the different experiments, we use a `PREFIX` for the experiment name. The `PREFIX` is used to create a unique experiment name. The `PREFIX` is also used to

create a unique TensorBoard folder, which is used to store the TensorBoard log files.

`spotpython` allows the specification of two different types of stopping criteria: first, the number of function evaluations (`fun_evals`), and second, the maximum run time in seconds (`max_time`). Here, we will set the number of function evaluations to infinity and the maximum run time to one minute.

Furthermore, we set the initial design size (`init_size`) to 10. The initial design is used to train the surrogate model. The surrogate model is used to predict the performance of the hyperparameter configurations. The initial design is also used to train the first model. Since the `init_size` belongs to the experimental design, it is set in the `design_control` dictionary, see [SOURCE].

`max_time` is set to one minute for demonstration purposes and `init_size` is set to 10 for demonstration purposes. For real experiments, these values should be increased. Note, the total run time may exceed the specified `max_time`, because the initial design is always evaluated, even if this takes longer than `max_time`.

ℹ Summary: General Experiment Setup

The following parameters are used to specify the general experiment setup:

```
PREFIX = "024"
fun_evals = inf
max_time = 1
init_size = 10
```

26.2.2. Data Setup

We use the `StandardScaler` [SOURCE] from `river` as the data-preprocessing model. The `StandardScaler` is used to standardize the data set, i.e., it has zero mean and unit variance.

The names of the training and test data sets are `train` and `test`, respectively. They are available as `pandas` dataframes. Both must use the same column names. The column names were set to `x1` to `x10` for the features and `y` for the target column during the data set generation in Section 26.1. Therefore, the `target_column` is set to `y` (as above).

ℹ Summary: Data Setup

The following parameters are used to specify the data setup:

```
prep_model_name = "StandardScaler"
test = test
train = train
target_column = "y"
```

26.2.3. Evaluation Setup

Here we use the `mean_absolute_error` [SOURCE] as the evaluation metric. Internally, this metric is passed to the objective (or loss) function `fun_oml_horizon` [SOURCE] and further to the evaluation function `eval_oml_horizon` [SOURCE].

`spotriver` also supports additional metrics. For example, the `metric_river` is used for the river based evaluation via `eval_oml_iter_progressive` [SOURCE]. The `metric_river` is implemented to simulate the behaviour of the "original" `river` metrics.

Summary: Evaluation Setup

The following parameter are used to select the evaluation metric:

```
metric_sklearn_name = "mean_absolute_error"
```

26.2.4. River-Specific Setup

In the online-machine-learning (OML) setup, the model is trained on a fixed number of observations and then evaluated on a fixed number of observations. The `horizon` defines the number of observations that are used for the evaluation. Here, a horizon of 7*24 is used, which corresponds to one week of data.

The `oml_grace_period` defines the number of observations that are used for the initial training of the model. This value is relatively small, since the online-machine-learning is trained on the incoming data and the model is updated continuously. However, it needs a certain number of observations to start the training process. Therefore, this short training period aka `oml_grace_period` is set to the horizon, i.e., the number of observations that are used for the evaluation. In this case, we use a horizon of 7*24.

The `weights` provide a flexible way to define specific requirements, e.g., if the memory is more important than the time, the weight for the memory can be increased. `spotriver` stores information about the model' s score (metric), memory, and time. The hyperparamter tuner requires a single objective. Therefore, a weighted sum of the metric, memory, and time is computed. The weights are defined in the `weights` array.

The `weights` provide a flexible way to define specific requirements, e.g., if the memory is more important than the time, the weight for the memory can be increased.

The `weight_coeff` defines a multiplier for the results: results are multiplied by (step/n_steps)**weight_coeff, where n_steps is the total number of iterations. Results from the beginning have a lower weight than results from the end if weight_coeff > 1. If weight_coeff == 0, all results have equal weight. Note, that the `weight_coeff` is only used internally for the tuner and does not affect the results that are used for the evaluation or comparisons.

Summary: River-Specific Setup

The following parameters are used:

```
horizon = 7*24
oml_grace_period = 7*24
weights = np.array([1, 0.01, 0.01])
weight_coeff = 0.0
```

26.2.5. Model Setup

By using `core_model_name = "tree.HoeffdingTreeRegressor"`, the `river` model class `HoeffdingTreeRegressor` [SOURCE] from the `tree` module is selected. For a given `core_model_name`, the corresponding hyperparameters are automatically loaded from the associated dictionary, which is stored as a JSON file. The JSON file contains hyperparameter type information, names, and bounds. For `river` models, the hyperparameters are stored in the `RiverHyperDict`, see [SOURCE]

Alternatively, you can load a local hyper_dict. Simply set `river_hyper_dict.json` as the filename. If `filename`is set to `None`, which is the default, the hyper_dict [SOURCE] is loaded from the `spotriver` package.

How hyperparameter levels can be modified is described in Section D.15.1.

Summary: Model Setup

The following parameters are used for the model setup:

```
from spotriver.fun.hyperriver import HyperRiver
from spotriver.hyperdict.river_hyper_dict import RiverHyperDict
core_model_name = "tree.HoeffdingTreeRegressor"
hyperdict = RiverHyperDict
```

26.2.6. Objective Function Setup

The loss function (metric) values are passed to the objective function `fun_oml_horizon` [SOURCE], which combines information about the loss, required memory and time as described in Section 26.2.4.

ℹ Summary: Objective Function Setup

The following parameters are used:

```
fun = HyperRiver().fun_oml_horizon
```

26.2.7. Surrogate Model Setup

The default surrogate model is the `Kriging` model, see [SOURCE]. We specify `noise` as `True` to include noise in the model. An `anisotropic` kernel is used, which allows different length scales for each dimension, by setting `n_theta = 2`. Furthermore, the interval for the `Lambda` value is set to `[1e-3, 1e2]`.
These parameters are set in the `surrogate_control` dictionary and therefore passed to the `surrogate_control_init` function [SOURCE].

```
noise = True
n_theta = 2
min_Lambda = 1e-3
max_Lambda = 10
```

26.2.8. Summary: Setting up the Experiment

At this stage, all required information is available to set up the dictionaries for the hyperparameter tuning. Altogether, the `fun_control`, `design_control`, `surrogate_control`, and `optimize_control` dictionaries are initialized as follows:

```
from spotpython.utils.init import fun_control_init, design_control_init, surrogate_co

fun = HyperRiver().fun_oml_horizon

fun_control = fun_control_init(
    PREFIX="024",
    fun_evals=inf,
    max_time=1,

    prep_model_name="StandardScaler",
```

```
    test=test,
    train=train,
    target_column=target_column,

    metric_sklearn_name="mean_absolute_error",
    horizon=7*24,
    oml_grace_period=7*24,
    weight_coeff=0.0,
    weights=np.array([1, 0.01, 0.01]),

    core_model_name="tree.HoeffdingTreeRegressor",
    hyperdict=RiverHyperDict,
   )


design_control = design_control_init(
    init_size=10,
)

surrogate_control = surrogate_control_init(
    noise=True,
    n_theta=2,
    min_Lambda=1e-3,
    max_Lambda=10,
)

optimizer_control = optimizer_control_init()
```

26.2.9. Run the `Spot` Optimizer

The class `Spot` [SOURCE] is the hyperparameter tuning workhorse. It is initialized with the following parameters, which were specified above.

- `fun`: the objective function
- `fun_control`: the dictionary with the control parameters for the objective function
- `design_control`: the dictionary with the control parameters for the experimental design
- `surrogate_control`: the dictionary with the control parameters for the surrogate model
- `optimizer_control`: the dictionary with the control parameters for the optimizer

spotpython allows maximum flexibility in the definition of the hyperparameter tuning setup. Alternative surrogate models, optimizers, and experimental designs can be used. Thus, interfaces for the surrogate model, experimental design, and optimizer are provided. The default surrogate model is the kriging model, the default optimizer is the differential evolution, and default experimental design is the Latin hypercube design.

Summary: Spot Setup

The following parameters are used for the Spot setup. These were specified above:

```
fun = fun
fun_control = fun_control
design_control = design_control
surrogate_control = surrogate_control
optimizer_control = optimizer_control
```

```
from spotpython.spot import Spot
spot_tuner = Spot(
    fun=fun,
    fun_control=fun_control,
    design_control=design_control,
    surrogate_control=surrogate_control,
    optimizer_control=optimizer_control,
)
res = spot_tuner.run()
```

```
spotpython tuning: 3.1966832222706283 [----------] 0.65%
spotpython tuning: 2.236465386801972 [#---------] 5.01%
spotpython tuning: 2.236465386801972 [#---------] 6.37%
spotpython tuning: 2.236465386801972 [#---------] 7.01%
spotpython tuning: 2.236465386801972 [#---------] 9.32%
spotpython tuning: 2.236465386801972 [#---------] 10.01%
spotpython tuning: 2.236465386801972 [#---------] 11.78%
spotpython tuning: 2.236465386801972 [#---------] 12.82%
spotpython tuning: 2.236465386801972 [#---------] 14.44%
spotpython tuning: 2.236465386801972 [##--------] 15.33%
spotpython tuning: 2.1653924069465718 [##--------] 16.19%
spotpython tuning: 2.1653924069465718 [##--------] 22.40%
spotpython tuning: 2.1653924069465718 [###-------] 27.84%
spotpython tuning: 2.1653924069465718 [###-------] 29.38%
spotpython tuning: 2.1653924069465718 [####------] 35.20%
```

```
spotpython tuning: 2.1643344812997447 [####------] 39.02%
spotpython tuning: 2.1643344812997447 [#####-----] 45.49%
spotpython tuning: 2.1643344812997447 [########--] 77.69%
spotpython tuning: 2.1643344812997447 [##########] 100.00% Done...

Experiment saved to 024_res.pkl
```

26.3. Using the spotgui

The spotgui [github] provides a convenient way to interact with the hyperparameter tuning process. To obtain the settings from Section 26.2.8, the spotgui can be started as shown in Figure 32.1.

Figure 26.1.: spotgui

26.4. Results

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with spotpython's method plot_progress. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress(log_y=True, filename=None)
```

Results can be printed in tabular form.

```
from spotpython.utils.eda import print_res_table
print_res_table(spot_tuner)
```

```
| name                    | type   | default           |   lower |   upper | tuned
|-------------------------|--------|-------------------|---------|---------|------------
| grace_period            | int    | 200               |    10.0 |  1000.0 | 591.0
| max_depth               | int    | 20                |     2.0 |    20.0 | 2.0
| delta                   | float  | 1e-07             |   1e-08 |   1e-06 | 1e-06
| tau                     | float  | 0.05              |    0.01 |     0.1 | 0.04109785
| leaf_prediction         | factor | mean              |     0.0 |     2.0 | adaptive
| leaf_model              | factor | LinearRegression  |     0.0 |     2.0 | LinearRegre
| model_selector_decay    | float  | 0.95              |     0.9 |    0.99 | 0.955195299
| splitter                | factor | EBSTSplitter      |     0.0 |     2.0 | QOSplitter
| min_samples_split       | int    | 5                 |     2.0 |    10.0 | 6.0
| binary_split            | factor | 0                 |     0.0 |     1.0 | 1
| max_size                | float  | 500.0             |   100.0 |  1000.0 | 685.180075
| memory_estimate_period  | int    | 6                 |     3.0 |     8.0 | 4.0
| stop_mem_management     | factor | 0                 |     0.0 |     1.0 | 1
| remove_poor_attrs       | factor | 0                 |     0.0 |     1.0 | 0
| merit_preprune          | factor | 1                 |     0.0 |     1.0 | 0
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=10.0)
```

26.5. Performance of the Model with Default Hyperparameters

26.5.1. Get Default Hyperparameters and Fit the Model

The default hyperparameters, which will be used for a comparion with the tuned hyperparameters, can be obtained with the following commands:

```
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array
X_start = get_default_hyperparameters_as_array(fun_control)
```

`spotpython` tunes numpy arrays, i.e., the hyperparameters are stored in a numpy array.

```
from spotpython.hyperparameters.values import get_one_core_model_from_X
model_default = get_one_core_model_from_X(X_start, fun_control, default=True)
```

26.5.2. Evaluate the Model with Default Hyperparameters

The model with the default hyperparameters can be trained and evaluated. The evaluation function `eval_oml_horizon` [SOURCE] is the same function that was used for the

hyperparameter tuning. During the hyperparameter tuning, the evaluation function was called from the objective (or loss) function `fun_oml_horizon` [SOURCE].

```
from spotriver.evaluation.eval_bml import eval_oml_horizon

df_eval_default, df_true_default = eval_oml_horizon(
                    model=model_default,
                    train=fun_control["train"],
                    test=fun_control["test"],
                    target_column=fun_control["target_column"],
                    horizon=fun_control["horizon"],
                    oml_grace_period=fun_control["oml_grace_period"],
                    metric=fun_control["metric_sklearn"],
                )
```

The three performance criteria, i.e., score (metric), runtime, and memory consumption, can be visualized with the following commands:

```
from spotriver.evaluation.eval_bml import plot_bml_oml_horizon_metrics, plot_bml_oml_h
df_labels=["default"]
plot_bml_oml_horizon_metrics(df_eval = [df_eval_default], log_y=False, df_labels=df_la
```

26.5.3. Show Predictions of the Model with Default Hyperparameters

- Select a subset of the data set for the visualization of the predictions:

- We use the mean, m, of the data set as the center of the visualization.
- We use 100 data points, i.e., $m \pm 50$ as the visualization window.

```
m = fun_control["test"].shape[0]
a = int(m/2)-50
b = int(m/2)+50
plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b]], target_column=target_column,  df_
```

26.6. Get SPOT Results

In a similar way, we can obtain the hyperparameters found by `spotpython`.

```
from spotpython.hyperparameters.values import get_one_core_model_from_X
X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
model_spot = get_one_core_model_from_X(X, fun_control)
```

```
df_eval_spot, df_true_spot = eval_oml_horizon(
                    model=model_spot,
                    train=fun_control["train"],
                    test=fun_control["test"],
                    target_column=fun_control["target_column"],
                    horizon=fun_control["horizon"],
                    oml_grace_period=fun_control["oml_grace_period"],
```

```
                    metric=fun_control["metric_sklearn"],
                )
```

```
df_labels=["default", "spot"]
plot_bml_oml_horizon_metrics(df_eval = [df_eval_default, df_eval_spot], log_y=False, c
```

```
plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b], df_true_spot[a:b]],
```

```
from spotpython.plot.validation import plot_actual_vs_predicted
plot_actual_vs_predicted(y_test=df_true_default[target_column], y_pred=df_true_default["Prediction"]
plot_actual_vs_predicted(y_test=df_true_spot[target_column], y_pred=df_true_spot["Prediction"], titl
```

26.7. Visualize Regression Trees

```
dataset_f = dataset.take(n_samples)
print(f"n_samples: {n_samples}")
```

```
for x, y in dataset_f:
    model_default.learn_one(x, y)
```

```
n_samples: 10000
```

> Caution: Large Trees
>
> - Since the trees are large, the visualization is suppressed by default.
> - To visualize the trees, uncomment the following line.

```
# model_default.draw()
```

```
model_default.summary
```

```
{'n_nodes': 23,
 'n_branches': 11,
 'n_leaves': 12,
 'n_active_leaves': 12,
 'n_inactive_leaves': 0,
 'height': 7,
 'total_observed_weight': 14168.0}
```

26.7.1. Spot Model

```
print(f"n_samples: {n_samples}")
dataset_f = dataset.take(n_samples)
for x, y in dataset_f:
    model_spot.learn_one(x, y)
```

```
n_samples: 10000
```

> Caution: Large Trees
>
> - Since the trees are large, the visualization is suppressed by default.
> - To visualize the trees, uncomment the following line.

```
# model_spot.draw()
```

```
model_spot.summary
```

```
{'n_nodes': 11,
 'n_branches': 5,
 'n_leaves': 6,
 'n_active_leaves': 4,
 'n_inactive_leaves': 2,
 'height': 5,
 'total_observed_weight': 14168.0}
```

```
from spotpython.utils.eda import compare_two_tree_models
print(compare_two_tree_models(model_default, model_spot))
```

```
| Parameter              |   Default |   Spot |
|------------------------|-----------|--------|
| n_nodes                |        23 |     11 |
| n_branches             |        11 |      5 |
| n_leaves               |        12 |      6 |
| n_active_leaves        |        12 |      4 |
| n_inactive_leaves      |         0 |      2 |
| height                 |         7 |      5 |
| total_observed_weight |     14168 |  14168 |
```

26.8. Detailed Hyperparameter Plots

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
grace_period:  100.0
max_depth:  33.894321664565794
delta:  0.5797447283001402
tau:  1.4713856271831611
leaf_prediction:  0.001479596015874668
leaf_model:  1.931856386424804
model_selector_decay:  4.786157626640002
splitter:  1.4565900972424308
min_samples_split:  0.1382652808717584
```

```
binary_split:  1.625141357041443
max_size:  1.0916453589770803
memory_estimate_period:  0.022436269549173618
stop_mem_management:  0.047328543752895025
remove_poor_attrs:  0.010606542155714278
merit_preprune:  7.386003020754522
```

26.9. Parallel Coordinates Plots

```
spot_tuner.parallel_plot()
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

27. The Friedman Drift Data Set

This chapter demonstrates hyperparameter tuning for `river`'s `Mondrian Tree Regressor` [SOURCE] with the Friedman drift data set [SOURCE]. The `Mondrian Tree Regressor` is a regression tree, i.e., it predicts a real value for each sample.

The data set was introduced in Section 26.1.

```
from river.datasets import synth
import pandas as pd
import numpy as np
from spotriver.utils.data_conversion import convert_to_df
from spotpython.utils.eda import print_exp_table, print_res_table

n_train = 6_000
n_test = 4_000
n_samples = n_train + n_test
target_column = "y"

dataset = synth.FriedmanDrift(
   drift_type='gra',
   position=(n_train/4, n_train/2),
   seed=123
)

train = convert_to_df(dataset, n_total=n_train)
train.columns = [f"x{i}" for i in range(1, 11)] + [target_column]

dataset = synth.FriedmanDrift(
   drift_type='gra',
   position=(n_test/4, n_test/2),
   seed=123
)
test = convert_to_df(dataset, n_total=n_test)
test.columns = [f"x{i}" for i in range(1, 11)] + [target_column]
```

27.1. Setup

We will use a general experiment, data, evaluation, river-specific, objective-function, and surrogate setup similar to the setup from Section 26.2. Only the model setup differs from the setup in Section 26.2. Here we use the `Mondrian Tree Regressor` from `river`.

```
from spotriver.hyperdict.river_hyper_dict import RiverHyperDict
core_model_name = "forest.AMFRegressor"
hyperdict = RiverHyperDict
hyperdict
```

```
spotriver.hyperdict.river_hyper_dict.RiverHyperDict
```

27.1.1. Select a User Hyperdictionary

Alternatively, you can load a local hyper_dict from the "userModel" folder. Here, we have selected a copy of the JSON `MondrianHyperDict` hyperdictionary from [SOURCE] and the `MondrianHyperDict` class from [SOURCE]. The hyperparameters of the `Mondrian Tree Regressor` are defined in the `MondrianHyperDict` class, i.e., there is an key "AMFRegressor" in the `hyperdict` "mondrian_hyper_dict.json" file.

```
import sys
sys.path.insert(0, './userModel')
import mondrian_hyper_dict
hyperdict = mondrian_hyper_dict.MondrianHyperDict
hyperdict
```

```
mondrian_hyper_dict.MondrianHyperDict
```

```
from spotpython.utils.init import fun_control_init, design_control_init, surrogate_co
from spotriver.fun.hyperriver import HyperRiver

fun = HyperRiver().fun_oml_horizon

fun_control = fun_control_init(
    PREFIX="503",
    fun_evals=inf,
    max_time=1,
```

```
    prep_model_name="StandardScaler",
    test=test,
    train=train,
    target_column=target_column,

    metric_sklearn_name="mean_absolute_error",
    horizon=7*24,
    oml_grace_period=7*24,
    weight_coeff=0.0,
    weights=np.array([1, 0.01, 0.01]),

    core_model_name="forest.AMFRegressor",
    hyperdict=hyperdict,
   )


design_control = design_control_init(
    init_size=5,
)

surrogate_control = surrogate_control_init(
    noise=True,
    n_theta=2,
    min_Lambda=1e-3,
    max_Lambda=10,
)

optimizer_control = optimizer_control_init()
```

27.2. Modify hyper_dict Hyperparameters for the Selected Algorithm aka core_model

After the core_model and the hyperdict are added to the fun_control dictionary, the hyperparameter tuning can be started. However, in some settings, the user wants to modify the hyperparameters. This can be done with the set_int_hyperparameter_values, set_float_hyperparameter_values, set_boolean_hyperparameter_values, and set_factor_hyperparameter_values functions, which can be imported from from spotpython.hyperparameters.values [SOURCE].

The following code shows how hyperparameter of type float and integer can be modified. Additional examples can be found in Section D.15.1.

```
print_exp_table(fun_control)
```

```
| name            | type   |   default |   lower |   upper | transform             |
|-----------------|--------|-----------|---------|---------|-----------------------|
| n_estimators    | int    |         3 |     2   |      10 | transform_power_2_int |
| step            | float  |         1 |     0.1 |      10 | None                  |
| use_aggregation | factor |         1 |     0   |       1 | None                  |
```

```
from spotpython.hyperparameters.values import set_int_hyperparameter_values, set_float
set_int_hyperparameter_values(fun_control, "n_estimators", 2, 7)
set_float_hyperparameter_values(fun_control, "step", 0.1, 15)
print_exp_table(fun_control)
```

```
Setting hyperparameter n_estimators to value [2, 7].
Variable type is int.
Core type is None.
Calling modify_hyper_parameter_bounds().
Setting hyperparameter step to value [0.1, 15].
Variable type is float.
Core type is None.
Calling modify_hyper_parameter_bounds().
| name            | type   |   default |   lower |   upper | transform             |
|-----------------|--------|-----------|---------|---------|-----------------------|
| n_estimators    | int    |         3 |     2   |       7 | transform_power_2_int |
| step            | float  |         1 |     0.1 |      15 | None                  |
| use_aggregation | factor |         1 |     0   |       1 | None                  |
```

> **Note: Active and Inactive Hyperparameters**
>
> Hyperparameters can be excluded from the tuning procedure by selecting identical values for the lower and upper bounds.

27.2.1. Run the Spot Optimizer

```
from spotpython.spot import Spot
spot_tuner = Spot(
    fun=fun,
    fun_control=fun_control,
    design_control=design_control,
    surrogate_control=surrogate_control,
```

```
    optimizer_control=optimizer_control,
)
res = spot_tuner.run()
```

```
spotpython tuning: 2.836819933181288 [####------] 39.91%
spotpython tuning: 2.8042758105528547 [########--] 78.86%
spotpython tuning: 2.8042758105528547 [##########] 100.00% Done...

Experiment saved to 503_res.pkl
```

We can start TensorBoard in the background with the following command, where ./runs is the default directory for the TensorBoard log files:

`tensorboard --logdir="./runs"`We can access the TensorBoard web server with the following URL:

```
http://localhost:6006/
```

27.3. Results

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with `spotpython`'s method `plot_progress`. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress()
```

Results can be printed in tabular form.

```
from spotpython.utils.eda import print_res_table
print_res_table(spot_tuner)
```

```
| name            | type    |   default |   lower |   upper |              tuned | trans
|-----------------|---------|-----------|---------|---------|--------------------|------
| n_estimators    | int     |       3.0 |     2.0 |       7 |                5.0 | trans
| step            | float   |       1.0 |     0.1 |      15 | 7.514327941395616 | None
| use_aggregation | factor  |       1.0 |     0.0 |       1 |                0.0 | None
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=10.0)
```

27.4. Performance of the Model with Default Hyperparameters

27.4.1. Get Default Hyperparameters and Fit the Model

The default hyperparameters, which will be used for a comparion with the tuned hyperparameters, can be obtained with the following commands:

```
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array
X_start = get_default_hyperparameters_as_array(fun_control)
```

`spotpython` tunes numpy arrays, i.e., the hyperparameters are stored in a numpy array.

```
from spotpython.hyperparameters.values import get_one_core_model_from_X
model_default = get_one_core_model_from_X(X_start, fun_control, default=True)
```

27.4.2. Evaluate the Model with Default Hyperparameters

The model with the default hyperparameters can be trained and evaluated. The evaluation function `eval_oml_horizon` [SOURCE] is the same function that was used for the hyperparameter tuning. During the hyperparameter tuning, the evaluation function was called from the objective (or loss) function `fun_oml_horizon` [SOURCE].

```
from spotriver.evaluation.eval_bml import eval_oml_horizon

df_eval_default, df_true_default = eval_oml_horizon(
                    model=model_default,
                    train=fun_control["train"],
                    test=fun_control["test"],
                    target_column=fun_control["target_column"],
                    horizon=fun_control["horizon"],
                    oml_grace_period=fun_control["oml_grace_period"],
                    metric=fun_control["metric_sklearn"],
                )
```

The three performance criteria, i.e., score (metric), runtime, and memory consumption, can be visualized with the following commands:

```
from spotriver.evaluation.eval_bml import plot_bml_oml_horizon_metrics, plot_bml_oml_horizon_predict
df_labels=["default"]
plot_bml_oml_horizon_metrics(df_eval = [df_eval_default], log_y=False, df_labels=df_labels, metric=f
```

27.4.3. Show Predictions of the Model with Default Hyperparameters

- Select a subset of the data set for the visualization of the predictions:
  - We use the mean, m, of the data set as the center of the visualization.
  - We use 100 data points, i.e., $m \pm 50$ as the visualization window.

```
m = fun_control["test"].shape[0]
a = int(m/2)-50
b = int(m/2)+50
plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b]], target_column=targe
```

27.5. Get SPOT Results

In a similar way, we can obtain the hyperparameters found by `spotpython`.

```
from spotpython.hyperparameters.values import get_one_core_model_from_X
X = spot_tuner.to_all_dim(spot_tuner.min_X.reshape(1,-1))
model_spot = get_one_core_model_from_X(X, fun_control)
```

```
df_eval_spot, df_true_spot = eval_oml_horizon(
                    model=model_spot,
                    train=fun_control["train"],
                    test=fun_control["test"],
                    target_column=fun_control["target_column"],
                    horizon=fun_control["horizon"],
                    oml_grace_period=fun_control["oml_grace_period"],
                    metric=fun_control["metric_sklearn"],
                )
```

```
df_labels=["default", "spot"]
plot_bml_oml_horizon_metrics(df_eval = [df_eval_default, df_eval_spot], log_y=False, df_labels=df_la
```

```
plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b], df_true_spot[a:b]],
```

```
from spotpython.plot.validation import plot_actual_vs_predicted
plot_actual_vs_predicted(y_test=df_true_default[target_column], y_pred=df_true_default
plot_actual_vs_predicted(y_test=df_true_spot[target_column], y_pred=df_true_spot["Pred
```

27.6. Detailed Hyperparameter Plots

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
n_estimators:  100.0
step:  0.2871103963337583
use_aggregation:  3.312386106285798
```

27.7. Parallel Coordinates Plots

```
spot_tuner.parallel_plot()
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

Part VIII.

Hyperparameter Tuning with PyTorch Lightning

28. Basic Lightning Module

28.1. Introduction

This chapter implements a basic Pytorch Lightning module. It is based on the Lightning documentation LIGHTNINGMODULE.

A `LightningModule` organizes your `PyTorch` code into six sections:

- Initialization (`__init__` and `setup()`).
- Train Loop (`training_step()`)
- Validation Loop (`validation_step()`)
- Test Loop (`test_step()`)
- Prediction Loop (`predict_step()`)
- Optimizers and LR Schedulers (`configure_optimizers()`)

The `Trainer` automates every required step in a clear and reproducible way. It is the most important part of PyTorch Lightning. It is responsible for training, testing, and validating the model. The `Lightning` core structure looks like this:

```
net = MyLightningModuleNet()
trainer = Trainer()
trainer.fit(net)
```

There are no `.cuda()` or `.to(device)` calls required. Lightning does these for you.

```
# don't do in Lightning
x = torch.Tensor(2, 3)
x = x.cuda()
x = x.to(device)

# do this instead
x = x  # leave it alone!

# or to init a new tensor
new_x = torch.Tensor(2, 3)
new_x = new_x.to(x)
```

A LightningModule is a `torch.nn.Module` but with added functionality. For example:

```
net = Net.load_from_checkpoint(PATH)
net.freeze()
out = net(x)
```

28.2. Starter Example: Transformer

Here are the only required methods for setting up a transfomer model:

```
import lightning as L
import torch

from lightning.pytorch.demos import Transformer


class LightningTransformer(L.LightningModule):
    def __init__(self, vocab_size):
        super().__init__()
        self.model = Transformer(vocab_size=vocab_size)

    def forward(self, inputs, target):
        return self.model(inputs, target)

    def training_step(self, batch, batch_idx):
        inputs, target = batch
        output = self(inputs, target)
        loss = torch.nn.functional.nll_loss(output, target.view(-1))
        return loss

    def configure_optimizers(self):
        return torch.optim.SGD(self.model.parameters(), lr=0.1)
```

The `LightningTransformer` class is a subclass of `LightningModule`. It can be trainted as follows:

```
from lightning.pytorch.demos import WikiText2
from torch.utils.data import DataLoader

dataset = WikiText2()
dataloader = DataLoader(dataset)
```

```
model = LightningTransformer(vocab_size=dataset.vocab_size)

trainer = L.Trainer(fast_dev_run=100)
trainer.fit(model=model, train_dataloaders=dataloader)
```

```
Training: |          | 0/? [00:00<?, ?it/s]
```

28.3. Lightning Core Methods

The LightningModule has many convenient methods, but the core ones you need to know about are shown in Table 28.1.

Table 28.1.: The core methods of a LightningModule

Method	Description
`__init__` and `setup`	Initializes the model.
`forward`	Performs a forward pass through the model. To run data through your model only (separate from `training_step`).
`training_step`	Performs a complete training step.
`validation_step`	Performs a complete validation step.
`test_step`	Performs a complete test step.
`predict_step`	Performs a complete prediction step.
`configure_optimizers`	Configures the optimizers and learning-rate schedulers.

We will take a closer look at thes methods.

28.3.1. Training Step

28.3.1.1. Basics

To activate the training loop, override the `training_step()` method. If you want to calculate epoch-level metrics and log them, use `log()`.

```
class LightningTransformer(L.LightningModule):
    def __init__(self, vocab_size):
        super().__init__()
        self.model = Transformer(vocab_size=vocab_size)
```

```
    def training_step(self, batch, batch_idx):
        inputs, target = batch
        output = self.model(inputs, target)
        loss = torch.nn.functional.nll_loss(output, target.view(-1))

        # logs metrics for each training_step,
        # and the average across the epoch, to the progress bar and logger
        self.log("train_loss", loss, on_step=True, on_epoch=True, prog_bar=True, logge
        return loss
```

The `log()` method automatically reduces the requested metrics across a complete epoch and devices.

28.3.1.2. Background

- Here is the pseudocode of what the `log()` method does under the hood:

```
outs = []
for batch_idx, batch in enumerate(train_dataloader):
    # forward
    loss = training_step(batch, batch_idx)
    outs.append(loss.detach())

    # clear gradients
    optimizer.zero_grad()
    # backward
    loss.backward()
    # update parameters
    optimizer.step()

# note: in reality, we do this incrementally, instead of keeping all outputs in memory
epoch_metric = torch.mean(torch.stack(outs))
```

- In the case that you need to make use of all the outputs from each `training_step()`, override the `on_train_epoch_end()` method.

```
class LightningTransformer(L.LightningModule):
    def __init__(self, vocab_size):
        super().__init__()
        self.model = Transformer(vocab_size=vocab_size)
        self.training_step_outputs = []

    def training_step(self, batch, batch_idx):
```

```
        inputs, target = batch
        output = self.model(inputs, target)
        loss = torch.nn.functional.nll_loss(output, target.view(-1))
        preds = ...
        self.training_step_outputs.append(preds)
        return loss

    def on_train_epoch_end(self):
        all_preds = torch.stack(self.training_step_outputs)
        # do something with all preds
        ...
        self.training_step_outputs.clear()  # free memory
```

28.3.2. Validation Step

28.3.2.1. Basics

To activate the validation loop while training, override the `validation_step()` method.

```
class LightningTransformer(L.LightningModule):
    def validation_step(self, batch, batch_idx):
        inputs, target = batch
        output = self.model(inputs, target)
        loss = F.cross_entropy(y_hat, y)
        self.log("val_loss", loss)
        return loss
```

28.3.2.2. Background

- You can also run just the validation loop on your validation dataloaders by overriding `validation_step()` and calling `validate()`.

```
model = LightningTransformer(vocab_size=dataset.vocab_size)
trainer = L.Trainer()
trainer.validate(model)
```

- In the case that you need to make use of all the outputs from each `validation_step()`, override the `on_validation_epoch_end()` method. Note that this method is called before `on_train_epoch_end()`.

```
class LightningTransformer(L.LightningModule):
    def __init__(self, vocab_size):
        super().__init__()
        self.model = Transformer(vocab_size=vocab_size)
        self.validation_step_outputs = []

    def validation_step(self, batch, batch_idx):
        x, y = batch
        inputs, target = batch
        output = self.model(inputs, target)
        loss = torch.nn.functional.nll_loss(output, target.view(-1))
        pred = ...
        self.validation_step_outputs.append(pred)
        return pred

    def on_validation_epoch_end(self):
        all_preds = torch.stack(self.validation_step_outputs)
        # do something with all preds
        ...
        self.validation_step_outputs.clear()  # free memory
```

28.3.3. Test Step

The process for enabling a test loop is the same as the process for enabling a validation loop. For this you need to override the `test_step()` method. The only difference is that the test loop is only called when `test()` is used.

```
def test_step(self, batch, batch_idx):
    inputs, target = batch
    output = self.model(inputs, target)
    loss = F.cross_entropy(y_hat, y)
    self.log("test_loss", loss)
    return loss
```

28.3.4. Predict Step

28.3.4.1. Basics

By default, the `predict_step()` method runs the `forward()` method. In order to customize this behaviour, simply override the `predict_step()` method.

```
class LightningTransformer(L.LightningModule):
    def __init__(self, vocab_size):
        super().__init__()
        self.model = Transformer(vocab_size=vocab_size)

    def predict_step(self, batch):
        inputs, target = batch
        return self.model(inputs, target)
```

28.3.4.2. Background

- If you want to perform inference with the system, you can add a `forward` method to the LightningModule.
- When using forward, you are responsible to call `eval()` and use the `no_grad()` context manager.

```
class LightningTransformer(L.LightningModule):
    def __init__(self, vocab_size):
        super().__init__()
        self.model = Transformer(vocab_size=vocab_size)

    def forward(self, batch):
        inputs, target = batch
        return self.model(inputs, target)

    def training_step(self, batch, batch_idx):
        inputs, target = batch
        output = self.model(inputs, target)
        loss = torch.nn.functional.nll_loss(output, target.view(-1))
        return loss

    def configure_optimizers(self):
        return torch.optim.SGD(self.model.parameters(), lr=0.1)


model = LightningTransformer(vocab_size=dataset.vocab_size)

model.eval()
with torch.no_grad():
    batch = dataloader.dataset[0]
    pred = model(batch)
```

28.4. Lightning Extras

This section covers some additional features of Lightning.

28.4.1. Lightning: Save Hyperparameters

Often times we train many versions of a model. You might share that model or come back to it a few months later at which point it is very useful to know how that model was trained (i.e.: what learning rate, neural network, etc.).

Lightning has a standardized way of saving the information for you in checkpoints and YAML files. The goal here is to improve readability and reproducibility.

Use `save_hyperparameters()` within your `LightningModule`'s `__init__` method. It will enable Lightning to store all the provided arguments under the `self.hparams` attribute. These hyperparameters will also be stored within the model checkpoint, which simplifies model re-instantiation after training.

```
class LitMNIST(L.LightningModule):
    def __init__(self, layer_1_dim=128, learning_rate=1e-2):
        super().__init__()
        # call this to save (layer_1_dim=128, learning_rate=1e-4) to the checkpoint
        self.save_hyperparameters()

        # equivalent
        self.save_hyperparameters("layer_1_dim", "learning_rate")

        # Now possible to access layer_1_dim from hparams
        self.hparams.layer_1_dim
```

28.4.2. Lightning: Model Loading

LightningModules that have hyperparameters automatically saved with `save_hyperparameters()` can conveniently be loaded and instantiated directly from a checkpoint with `load_from_checkpoint()`:

```
# to load specify the other args
model = LitMNIST.load_from_checkpoint(PATH, loss_fx=torch.nn.SomeOtherLoss, generator_
```

28.5. Starter Example: Linear Neural Network

We will use the `LightningModule` to create a simple neural network for regression. It will be implemented as the `LightningBasic` class.

28.5.1. Hidden Layers

To specify the number of hidden layers, we will use the hyperparameter `l1` and the function `get_hidden_sizes()` [DOC] from the `spotpython` package.

```
from spotpython.hyperparameters.architecture import get_hidden_sizes
_L_in = 10
l1 = 20
max_n = 4
get_hidden_sizes(_L_in, l1, max_n)
```

```
[20, 10, 10, 5]
```

28.5.2. Hyperparameters

The argument `l1` will be treated as a hyperparameter, so it will be tuned in the following steps. Besides `l1`, additonal hyperparameters are `act_fn` and `dropout_prob`.

The arguments `_L_in`, `_L_out`, and `_torchmetric` are not hyperparameters, but are needed to create the network. The first two are specified by the data and the latter by user preferences (the desired evaluation metric).

28.5.3. The LightningBasic Class

```
import lightning as L
import torch
import torch.nn.functional as F
import torchmetrics.functional.regression
from torch import nn
from spotpython.hyperparameters.architecture import get_hidden_sizes

class LightningBasic(L.LightningModule):
    def __init__(
    self,
    l1: int,
```

```
    act_fn: nn.Module,
    dropout_prob: float,
    _L_in: int,
    _L_out: int,
    _torchmetric: str,
    *args,
    **kwargs):
        super().__init__()
        self._L_in = _L_in
        self._L_out = _L_out
        self._torchmetric = _torchmetric
        self.metric = getattr(torchmetrics.functional.regression, _torchmetric)
        # _L_in and _L_out are not hyperparameters, but are needed to create the netwo
        # _torchmetric is not a hyperparameter, but is needed to calculate the loss
        self.save_hyperparameters(ignore=["_L_in", "_L_out", "_torchmetric"])
        # set dummy input array for Tensorboard Graphs
        # set log_graph=True in Trainer to see the graph (in traintest.py)
        hidden_sizes = get_hidden_sizes(_L_in=self._L_in, l1=l1, max_n=4)
        # Create the network based on the specified hidden sizes
        layers = []
        layer_sizes = [self._L_in] + hidden_sizes
        layer_size_last = layer_sizes[0]
        for layer_size in layer_sizes[1:]:
            layers += [
                nn.Linear(layer_size_last, layer_size),
                self.hparams.act_fn,
                nn.Dropout(self.hparams.dropout_prob),
            ]
            layer_size_last = layer_size
        layers += [nn.Linear(layer_sizes[-1], self._L_out)]
        # nn.Sequential summarizes a list of modules into a single module,
        # applying them in sequence
        self.layers = nn.Sequential(*layers)

    def _calculate_loss(self, batch):
        x, y = batch
        y = y.view(len(y), 1)
        y_hat = self.layers(x)
        loss = self.metric(y_hat, y)
        return loss

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.layers(x)
```

```
    def training_step(self, batch: tuple) -> torch.Tensor:
        loss = self._calculate_loss(batch)
        self.log("train_loss", loss, on_step=True, on_epoch=True, prog_bar=True, logger=True)
        return loss

    def validation_step(self, batch: tuple) -> torch.Tensor:
        loss = self._calculate_loss(batch)
        # logs metrics for each training_step,
        # and the average across the epoch, to the progress bar and logger
        self.log("val_loss", loss, on_step=True, on_epoch=True, prog_bar=True, logger=True)
        return loss

    def test_step(self, batch, batch_idx):
        loss = self._calculate_loss(batch)
        # logs metrics for each training_step,
        # and the average across the epoch, to the progress bar and logger
        self.log("test_loss", loss, on_step=True, on_epoch=True, prog_bar=True, logger=True)
        return loss

    def predict_step(self, batch, batch_idx, dataloader_idx=0):
        x, _ = batch
        y_hat = self.layers(x)
        return y_hat

    def configure_optimizers(self):
        return torch.optim.Adam(self.layers.parameters(), lr=0.02)
```

We can instantiate the `LightningBasic` class as follows:

```
model_base = LightningBasic(
    l1=20,
    act_fn=nn.ReLU(),
    dropout_prob=0.01,
    _L_in=10,
    _L_out=1,
    _torchmetric="mean_squared_error")
```

It has the following structure:

```
print(model_base)
```

```
LightningBasic(
  (layers): Sequential(
```

```
    (0): Linear(in_features=10, out_features=20, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.01, inplace=False)
    (3): Linear(in_features=20, out_features=10, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.01, inplace=False)
    (6): Linear(in_features=10, out_features=10, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.01, inplace=False)
    (9): Linear(in_features=10, out_features=5, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.01, inplace=False)
    (12): Linear(in_features=5, out_features=1, bias=True)
  )
)
```

```python
from spotpython.plot.xai import viz_net
viz_net(net=model_base,
    device="cpu",
    filename="model_architecture700", format="png")
```

Figure 28.1.: Model architecture

28.5.4. The Data Set: Diabetes

We will use the `Diabetes` [DOC] data set from the `spotpython` package, which is a PyTorch Dataset for regression based on a data set from `scikit-learn`. It consists of DataFrame entries, which were converted to PyTorch tensors.

Ten baseline variables, age, sex, body mass index, average blood pressure, and six blood serum measurements were obtained for each of n = 442 diabetes patients, as well as the response of interest, a quantitative measure of disease progression one year after baseline.

The `Diabetes` data set has the following properties:

- Number of Instances: 442
- Number of Attributes: First 10 columns are numeric predictive values.
- Target: Column 11 is a quantitative measure of disease progression one year after baseline.
- Attribute Information:
  - age age in years
  - sex
  - bmi body mass index
  - bp average blood pressure
  - s1 tc, total serum cholesterol
  - s2 ldl, low-density lipoproteins
  - s3 hdl, high-density lipoproteins
  - s4 tch, total cholesterol / HDL
  - s5 ltg, possibly log of serum triglycerides level
  - s6 glu, blood sugar level

```
from torch.utils.data import DataLoader
from spotpython.data.diabetes import Diabetes
import torch
dataset = Diabetes(feature_type=torch.float32, target_type=torch.float32)
# Set batch size for DataLoader to 2 for demonstration purposes
batch_size = 2
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=False)
for batch in dataloader:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Batch Size: 2
```

```
---------------
Inputs: tensor([[ 0.0381,  0.0507,  0.0617,  0.0219, -0.0442, -0.0348, -0.0434, -0.0026,
          0.0199, -0.0176],
        [-0.0019, -0.0446, -0.0515, -0.0263, -0.0084, -0.0192,  0.0744, -0.0395,
         -0.0683, -0.0922]])
Targets: tensor([151.,  75.])
```

28.5.5. The DataLoaders

Before we can call the `Trainer` to fit, validate, and test the model, we need to create the `DataLoaders` for each of these steps. The `DataLoaders` are used to load the data into the model in batches and need the `batch_size`.

```
import torch
from spotpython.data.diabetes import Diabetes
from torch.utils.data import DataLoader

batch_size = 8

dataset = Diabetes(target_type=torch.float)
train1_set, test_set = torch.utils.data.random_split(dataset, [0.6, 0.4])
train_set, val_set = torch.utils.data.random_split(train1_set, [0.6, 0.4])
print(f"Full Data Set: {len(dataset)}")
print(f"Train Set: {len(train_set)}")
print(f"Validation Set: {len(val_set)}")
print(f"Test Set: {len(test_set)}")
train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True, drop_last=True, pin_memory
test_loader = DataLoader(test_set, batch_size=batch_size)
val_loader = DataLoader(val_set, batch_size=batch_size)
```

```
Full Data Set: 442
Train Set: 160
Validation Set: 106
Test Set: 176
```

28.5.6. The Trainer

Now we are ready to train the model. We will use the `Trainer` class from the `lightning` package. For demonstration purposes, we will train the model for 100 epochs only.

```
epochs = 100

trainer = L.Trainer(max_epochs=epochs, enable_progress_bar=True)
trainer.fit(model=model_base, train_dataloaders=train_loader)
```

```
Training: |          | 0/? [00:00<?, ?it/s]
```

```
trainer.validate(model_base, val_loader)
```

```
# automatically loads the best weights for you
out = trainer.test(model_base, test_loader, verbose=True)
```

```
Testing: |          | 0/? [00:00<?, ?it/s]
```

```
        Test metric               DataLoader 0

      test_loss_epoch           3428.052978515625
```

```
yhat = trainer.predict(model_base, test_loader)
# convert the list of tensors to a numpy array
yhat = torch.cat(yhat).numpy()
yhat.shape
```

28.5.7. Using a DataModule

Instead of creating the three `DataLoaders` manually, we can use the `LightDataModule` class from the `spotpython` package.

```
from spotpython.data.lightdatamodule import LightDataModule
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
```

There is a minor difference in the sizes of the data sets due to the random split as can be seen in the following code:

```
print(f"Full Data Set: {len(dataset)}")
print(f"Training set size: {len(data_module.data_train)}")
print(f"Validation set size: {len(data_module.data_val)}")
print(f"Test set size: {len(data_module.data_test)}")
```

```
Full Data Set: 442
Training set size: 160
Validation set size: 106
Test set size: 177
```

The `DataModule` can be used to train the model as follows:

```
trainer = L.Trainer(max_epochs=epochs, enable_progress_bar=False)
trainer.fit(model=model_base, datamodule=data_module)
```

```
trainer.validate(model=model_base, datamodule=data_module, verbose=True, ckpt_path=None)
```

```
       Validate metric              DataLoader 0

       val_loss_epoch             2708.508544921875
```

```
[{'val_loss_epoch': 2708.508544921875}]
```

```
trainer.test(model=model_base, datamodule=data_module, verbose=True, ckpt_path=None)
```

```
         Test metric                DataLoader 0

       test_loss_epoch            2632.136962890625
```

```
[{'test_loss_epoch': 2632.136962890625}]
```

28.6. Using spotpython with Pytorch Lightning

```
import os
from math import inf
import warnings
warnings.filterwarnings("ignore")
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_co
from spotpython.utils.eda import print_exp_table, print_res_table
from spotpython.spot import Spot
from spotpython.utils.file import get_experiment_filename
```

```
PREFIX="700"
data_set = Diabetes()
fun_control = fun_control_init(
    PREFIX=PREFIX,
    save_experiment=True,
    fun_evals=inf,
    fun_repeats=2,
    max_time=1,
    data_set=data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1,
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True,
    noise=True,
    ocba_delta = 1,  )
fun = HyperLight().fun
from spotpython.hyperparameters.values import set_hyperparameter
set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,7])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,3])

design_control = design_control_init(init_size=10, repeats=2)

print_exp_table(fun_control)

spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
```

```
res = spot_tuner.run()
spot_tuner.plot_progress()
print_res_table(spot_tuner)
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_02_12_22_37_13_0
Created spot_tensorboard_path: runs/spot_logs/700_maans08_2025-02-12_22-37-13 for SummaryWriter()
module_name: light
submodule_name: regression
model_name: NNLinearRegressor
| name           | type   | default   |   lower |   upper | transform             |
|----------------|--------|-----------|---------|---------|-----------------------|
| l1             | int    | 3         |     3   |   4     | transform_power_2_int |
| epochs         | int    | 4         |     3   |   7     | transform_power_2_int |
| batch_size     | int    | 4         |     4   |  11     | transform_power_2_int |
| act_fn         | factor | ReLU      |     0   |   5     | None                  |
| optimizer      | factor | SGD       |     0   |   2     | None                  |
| dropout_prob   | float  | 0.01      |     0   |   0.025 | None                  |
| lr_mult        | float  | 1.0       |     0.1 |  10     | None                  |
| patience       | int    | 2         |     2   |   3     | transform_power_2_int |
| batch_norm     | factor | 0         |     0   |   1     | None                  |
| initialization | factor | Default   |     0   |   4     | None                  |
Experiment saved to 700_exp.pkl

train_model result: {'val_loss': 23075.09765625, 'hp_metric': 23075.09765625}

train_model result: {'val_loss': 23175.75, 'hp_metric': 23175.75}

train_model result: {'val_loss': 3110.947021484375, 'hp_metric': 3110.947021484375}

train_model result: {'val_loss': 3212.389404296875, 'hp_metric': 3212.389404296875}

train_model result: {'val_loss': 4957.79296875, 'hp_metric': 4957.79296875}

train_model result: {'val_loss': 4993.95263671875, 'hp_metric': 4993.95263671875}

train_model result: {'val_loss': 24098.697265625, 'hp_metric': 24098.697265625}

train_model result: {'val_loss': 23989.583984375, 'hp_metric': 23989.583984375}

train_model result: {'val_loss': 22617.146484375, 'hp_metric': 22617.146484375}
```

```
train_model result: {'val_loss': 22949.05859375, 'hp_metric': 22949.05859375}

train_model result: {'val_loss': 4575.27001953125, 'hp_metric': 4575.27001953125}

train_model result: {'val_loss': 4273.986328125, 'hp_metric': 4273.986328125}

train_model result: {'val_loss': 21430.103515625, 'hp_metric': 21430.103515625}

train_model result: {'val_loss': 21671.5234375, 'hp_metric': 21671.5234375}

train_model result: {'val_loss': 3811.26171875, 'hp_metric': 3811.26171875}

train_model result: {'val_loss': 4142.75537109375, 'hp_metric': 4142.75537109375}

train_model result: {'val_loss': 20498.853515625, 'hp_metric': 20498.853515625}

train_model result: {'val_loss': 20785.01953125, 'hp_metric': 20785.01953125}

train_model result: {'val_loss': 24013.759765625, 'hp_metric': 24013.759765625}
train_model result: {'val_loss': 23966.04296875, 'hp_metric': 23966.04296875}

train_model result: {'val_loss': 4272.42431640625, 'hp_metric': 4272.42431640625}

train_model result: {'val_loss': 3318.66162109375, 'hp_metric': 3318.66162109375}
train_model result: {'val_loss': 3650.565673828125, 'hp_metric': 3650.565673828125}
spotpython tuning: 3110.947021484375 [#---------] 11.41%

train_model result: {'val_loss': 4493.92138671875, 'hp_metric': 4493.92138671875}
train_model result: {'val_loss': nan, 'hp_metric': nan}
train_model result: {'val_loss': nan, 'hp_metric': nan}
spotpython tuning: 3110.947021484375 [#---------] 14.38%

train_model result: {'val_loss': 3781.4375, 'hp_metric': 3781.4375}

train_model result: {'val_loss': 4434.3798828125, 'hp_metric': 4434.3798828125}

train_model result: {'val_loss': 23589.689453125, 'hp_metric': 23589.689453125}
spotpython tuning: 3110.947021484375 [##--------] 19.86%

train_model result: {'val_loss': 17597.259765625, 'hp_metric': 17597.259765625}
```

```
train_model result: {'val_loss': 3071.425048828125, 'hp_metric': 3071.425048828125}
train_model result: {'val_loss': 3183.198974609375, 'hp_metric': 3183.198974609375}
spotpython tuning: 3071.425048828125 [###-------] 29.80%

train_model result: {'val_loss': 6175.1220703125, 'hp_metric': 6175.1220703125}

train_model result: {'val_loss': 6293.92724609375, 'hp_metric': 6293.92724609375}
train_model result: {'val_loss': 5754.35498046875, 'hp_metric': 5754.35498046875}
spotpython tuning: 3071.425048828125 [####------] 37.43%

train_model result: {'val_loss': 3076.109375, 'hp_metric': 3076.109375}

train_model result: {'val_loss': 4634.12109375, 'hp_metric': 4634.12109375}
train_model result: {'val_loss': 5334.55615234375, 'hp_metric': 5334.55615234375}
spotpython tuning: 3071.425048828125 [#####-----] 47.63%

train_model result: {'val_loss': 2996.365966796875, 'hp_metric': 2996.365966796875}

train_model result: {'val_loss': 3090.901123046875, 'hp_metric': 3090.901123046875}
train_model result: {'val_loss': 3341.3662109375, 'hp_metric': 3341.3662109375}
spotpython tuning: 2996.365966796875 [######----] 55.70%

train_model result: {'val_loss': 3119.7001953125, 'hp_metric': 3119.7001953125}

train_model result: {'val_loss': 3148.246826171875, 'hp_metric': 3148.246826171875}
train_model result: {'val_loss': 3053.924560546875, 'hp_metric': 3053.924560546875}
spotpython tuning: 2996.365966796875 [######----] 62.82%

train_model result: {'val_loss': 3018.971435546875, 'hp_metric': 3018.971435546875}

train_model result: {'val_loss': 3167.308349609375, 'hp_metric': 3167.308349609375}
train_model result: {'val_loss': 3094.55419921875, 'hp_metric': 3094.55419921875}
spotpython tuning: 2996.365966796875 [#######---] 74.92%

train_model result: {'val_loss': 2867.625, 'hp_metric': 2867.625}

train_model result: {'val_loss': 4446.0107421875, 'hp_metric': 4446.0107421875}
train_model result: {'val_loss': 4090.0439453125, 'hp_metric': 4090.0439453125}
spotpython tuning: 2867.625 [########--] 84.26%

train_model result: {'val_loss': 2953.30224609375, 'hp_metric': 2953.30224609375}
```

```
train_model result: {'val_loss': 22691.08984375, 'hp_metric': 22691.08984375}
train_model result: {'val_loss': 22314.458984375, 'hp_metric': 22314.458984375}
spotpython tuning: 2867.625 [##########] 100.00% Done...

Experiment saved to 700_res.pkl
```

```
| name           | type   | default   |   lower |   upper | tuned               | trans
|----------------|--------|-----------|---------|---------|---------------------|------
| l1             | int    | 3         |     3.0 |     4.0 | 3.0                 | trans
| epochs         | int    | 4         |     3.0 |     7.0 | 6.0                 | trans
| batch_size     | int    | 4         |     4.0 |    11.0 | 5.0                 | trans
| act_fn         | factor | ReLU      |     0.0 |     5.0 | ReLU                | None
| optimizer      | factor | SGD       |     0.0 |     2.0 | Adadelta            | None
| dropout_prob   | float  | 0.01      |     0.0 |   0.025 | 0.0238651659902009  | None
| lr_mult        | float  | 1.0       |     0.1 |    10.0 | 0.6065428917480006  | None
| patience       | int    | 2         |     2.0 |     3.0 | 3.0                 | trans
| batch_norm     | factor | 0         |     0.0 |     1.0 | 0                   | None
| initialization | factor | Default   |     0.0 |     4.0 | kaiming_normal      | None
```

29. Details of the Lightning Module Integration in spotpython

29.1. Introduction

Based on the Diabetes Data set and the `NNLinearRegressor` model, we will provide details on the integration of the Lightning module in spotpython.

- Section 29.2: The `Hyperlight` class provides the `fun` method, which takes `X` and `fun_control` as arguments. It calls the `train_model` method.
- Section 29.3: The `train_model` method trains the model and returns the loss.
- Section 29.4: The `Trainer` class is used to train the model and validate it. It also uses the `LightDataModule` class to load the data.

29.2. 1. spotpython.fun.hyperlight.HyperLight.fun()

The class `Hyperlight` provides the method `fun`, which takes `X` (`np.ndarray`) and `fun_control` (`dict`) as arguments. It calls the

```
from math import inf
import numpy as np
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import fun_control_init
from spotpython.utils.eda import print_exp_table
from spotpython.spot import Spot
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array

PREFIX="000"

data_set = Diabetes()

fun_control = fun_control_init(
    PREFIX=PREFIX,
```

```
    save_experiment=True,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1,
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True,
    seed=42,)

print_exp_table(fun_control)

X = get_default_hyperparameters_as_array(fun_control)
# set epochs to 2^8:
# X[0, 1] = 8
# set patience to 2^10:
# X[0, 7] = 10

print(f"X: {X}")
# combine X and X to a np.array with shape (2, n_hyperparams)
# so that two values are returned
X = np.vstack((X, X, X))
print(f"X: {X}")
```

```
hyper_light = HyperLight(seed=125, log_level=50)
hyper_light.fun(X, fun_control)
```

- Using the same seed:

```
hyper_light = HyperLight(seed=125, log_level=50)
hyper_light.fun(X, fun_control)
```

- Using a different seed:

```
hyper_light = HyperLight(seed=123, log_level=50)
hyper_light.fun(X, fun_control)
```

29.3. 2. spotpython.light.trainmodel.train_model()

```
from math import inf
import numpy as np
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.utils.init import fun_control_init
from spotpython.utils.eda import print_exp_table
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array
from spotpython.hyperparameters.values import assign_values, generate_one_config_from_var_dict, get_
from spotpython.light.trainmodel import train_model
import pprint

PREFIX="000"

data_set = Diabetes()

fun_control = fun_control_init(
    PREFIX=PREFIX,
    save_experiment=True,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1,
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True,
    seed=42,)

print_exp_table(fun_control)

X = get_default_hyperparameters_as_array(fun_control)
# set epochs to 2^8:
# X[0, 1] = 8
# set patience to 2^10:
# X[0, 7] = 10

print(f"X: {X}")
# combine X and X to a np.array with shape (2, n_hyperparams)
# so that two values are returned
X = np.vstack((X, X))
var_dict = assign_values(X, get_var_name(fun_control))
for config in generate_one_config_from_var_dict(var_dict, fun_control):
```

```
    pprint.pprint(config)
    y = train_model(config, fun_control)
```

29.4. 3. Trainer: fit and validate

- Generate the `config` dictionary:

```
from math import inf
import numpy as np
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.utils.init import fun_control_init
from spotpython.utils.eda import print_exp_table
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array
from spotpython.hyperparameters.values import assign_values, generate_one_config_from_
from spotpython.light.trainmodel import train_model

PREFIX="000"

data_set = Diabetes()

fun_control = fun_control_init(
    PREFIX=PREFIX,
    save_experiment=True,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1,
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True,
    seed=42,)
print_exp_table(fun_control)
X = get_default_hyperparameters_as_array(fun_control)
# set epochs to 2^8:
X[0, 1] = 10
# set patience to 2^10:
X[0, 7] = 10
print(f"X: {X}")
var_dict = assign_values(X, get_var_name(fun_control))
```

```
config = list(generate_one_config_from_var_dict(var_dict, fun_control))[0]
config
```

```
_L_in = 10
_L_out = 1
_L_cond = 0
_torchmetric = "mean_squared_error"
```

29.4.1. Commented: Using the fun_control dictionary

```
# model = fun_control["core_model"](**config, _L_in=_L_in, _L_out=_L_out, _L_cond=_L_cond, _torchmet
# model
```

29.4.2. Using the source code:

```
import lightning as L
import torch
from torch import nn
from spotpython.hyperparameters.optimizer import optimizer_handler
import torchmetrics.functional.regression
import torch.optim as optim
from spotpython.hyperparameters.architecture import get_hidden_sizes


class NNLinearRegressor(L.LightningModule):
    def __init__(
        self,
        l1: int,
        epochs: int,
        batch_size: int,
        initialization: str,
        act_fn: nn.Module,
        optimizer: str,
        dropout_prob: float,
        lr_mult: float,
        patience: int,
        batch_norm: bool,
        _L_in: int,
        _L_out: int,
```

```
        _torchmetric: str,
        *args,
        **kwargs,
    ):
        super().__init__()
        # Attribute 'act_fn' is an instance of `nn.Module` and is already saved during
        # checkpointing. It is recommended to ignore them
        # using `self.save_hyperparameters(ignore=['act_fn'])`
        # self.save_hyperparameters(ignore=["act_fn"])
        #
        self._L_in = _L_in
        self._L_out = _L_out
        if _torchmetric is None:
            _torchmetric = "mean_squared_error"
        self._torchmetric = _torchmetric
        self.metric = getattr(torchmetrics.functional.regression, _torchmetric)
        # _L_in and _L_out are not hyperparameters, but are needed to create the netwo
        # _torchmetric is not a hyperparameter, but is needed to calculate the loss
        self.save_hyperparameters(ignore=["_L_in", "_L_out", "_torchmetric"])
        # set dummy input array for Tensorboard Graphs
        # set log_graph=True in Trainer to see the graph (in traintest.py)
        self.example_input_array = torch.zeros((batch_size, self._L_in))
        if self.hparams.l1 < 4:
            raise ValueError("l1 must be at least 4")
        hidden_sizes = get_hidden_sizes(_L_in=self._L_in, l1=l1, n=10)

        if batch_norm:
            # Add batch normalization layers
            layers = []
            layer_sizes = [self._L_in] + hidden_sizes
            for i in range(len(layer_sizes) - 1):
                current_layer_size = layer_sizes[i]
                next_layer_size = layer_sizes[i + 1]
                layers += [
                    nn.Linear(current_layer_size, next_layer_size),
                    nn.BatchNorm1d(next_layer_size),
                    self.hparams.act_fn,
                    nn.Dropout(self.hparams.dropout_prob),
                ]
            layers += [nn.Linear(layer_sizes[-1], self._L_out)]
        else:
            layers = []
            layer_sizes = [self._L_in] + hidden_sizes
            for i in range(len(layer_sizes) - 1):
```

```
            current_layer_size = layer_sizes[i]
            next_layer_size = layer_sizes[i + 1]
            layers += [
                nn.Linear(current_layer_size, next_layer_size),
                self.hparams.act_fn,
                nn.Dropout(self.hparams.dropout_prob),
            ]
        layers += [nn.Linear(layer_sizes[-1], self._L_out)]

    # Wrap the layers into a sequential container
    self.layers = nn.Sequential(*layers)

    # Initialization (Xavier, Kaiming, or Default)
    self.apply(self._init_weights)

def _init_weights(self, module):
    if isinstance(module, nn.Linear):
        if self.hparams.initialization == "xavier_uniform":
            nn.init.xavier_uniform_(module.weight)
        elif self.hparams.initialization == "xavier_normal":
            nn.init.xavier_normal_(module.weight)
        elif self.hparams.initialization == "kaiming_uniform":
            nn.init.kaiming_uniform_(module.weight)
        elif self.hparams.initialization == "kaiming_normal":
            nn.init.kaiming_normal_(module.weight)
        else:  # "Default"
            nn.init.uniform_(module.weight)
        if module.bias is not None:
            nn.init.zeros_(module.bias)

def forward(self, x: torch.Tensor) -> torch.Tensor:
    """
    Performs a forward pass through the model.

    Args:
        x (torch.Tensor): A tensor containing a batch of input data.

    Returns:
        torch.Tensor: A tensor containing the output of the model.

    """
    x = self.layers(x)
    return x
```

```
    def _calculate_loss(self, batch):
        """
        Calculate the loss for the given batch.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            mode (str, optional): The mode of the model. Defaults to "train".

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.

        """
        x, y = batch
        y = y.view(len(y), 1)
        y_hat = self(x)
        loss = self.metric(y_hat, y)
        return loss

    def training_step(self, batch: tuple) -> torch.Tensor:
        """
        Performs a single training step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.

        """
        loss = self._calculate_loss(batch)
        self.log("train_loss", loss, on_step=False, on_epoch=True, prog_bar=False)
        return loss

    def validation_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) ->
        """
        Performs a single validation step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults t

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.
```

```
        """
        loss = self._calculate_loss(batch)
        self.log("val_loss", loss, on_step=False, on_epoch=True, prog_bar=prog_bar)
        self.log("hp_metric", loss, on_step=False, on_epoch=True, prog_bar=prog_bar)
        return loss

    def test_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> torch.Tensor:
        """
        Performs a single test step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults to False.

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.
        """
        loss = self._calculate_loss(batch)
        self.log("val_loss", loss, on_step=False, on_epoch=True, prog_bar=prog_bar)
        self.log("hp_metric", loss, on_step=False, on_epoch=True, prog_bar=prog_bar)
        return loss

    def predict_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> torch.Tensor:
        """
        Performs a single prediction step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults to False.

        Returns:
            torch.Tensor: A tensor containing the prediction for this batch.
        """
        x, y = batch
        yhat = self(x)
        y = y.view(len(y), 1)
        yhat = yhat.view(len(yhat), 1)
        print(f"Predict step x: {x}")
        print(f"Predict step y: {y}")
        print(f"Predict step y_hat: {yhat}")
        # pred_loss = F.mse_loss(y_hat, y)
        # pred loss not registered
```

```
        # self.log("pred_loss", pred_loss, prog_bar=prog_bar)
        # self.log("hp_metric", pred_loss, prog_bar=prog_bar)
        # MisconfigurationException: You are trying to `self.log()`
        # but the loop's result collection is not registered yet.
        # This is most likely because you are trying to log in a `predict` hook, but i
        # If you want to manually log, please consider using `self.log_dict({'pred_los
        return (x, y, yhat)

    def configure_optimizers(self) -> torch.optim.Optimizer:
        """
        Configures the optimizer for the model.

        Notes:
            The default Lightning way is to define an optimizer as
            `optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)`.
            spotpython uses an optimizer handler to create the optimizer, which
            adapts the learning rate according to the lr_mult hyperparameter as
            well as other hyperparameters. See `spotpython.hyperparameters.optimizer.p

        Returns:
            torch.optim.Optimizer: The optimizer to use during training.

        """
        # optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)
        optimizer = optimizer_handler(optimizer_name=self.hparams.optimizer, params=se

        num_milestones = 3  # Number of milestones to divide the epochs
        milestones = [int(self.hparams.epochs / (num_milestones + 1) * (i + 1)) for i
        scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=milestones, g

        lr_scheduler_config = {
            "scheduler": scheduler,
            "interval": "epoch",
            "frequency": 1,
        }

        return {"optimizer": optimizer, "lr_scheduler": lr_scheduler_config}
```

```
model = NNLinearRegressor(**config, _L_in=_L_in, _L_out=_L_out, _L_cond=_L_cond, _torc
model
```

```
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
```

```
data_set = Diabetes()
dm = LightDataModule(
    dataset=data_set,
    batch_size=config["batch_size"],
    test_size=fun_control["test_size"],
    test_seed=fun_control["test_seed"],
    scaler=None,
)
```

- Using `callbacks` for early stopping:

```
from lightning.pytorch.callbacks.early_stopping import EarlyStopping
callbacks = [EarlyStopping(monitor="val_loss", patience=config["patience"], mode="min", strict=False
```

```
timestamp = True

from lightning.pytorch.callbacks import ModelCheckpoint
if not timestamp:
    # add ModelCheckpoint only if timestamp is False
    callbacks.append(ModelCheckpoint(dirpath=os.path.join(fun_control["CHECKPOINT_PATH"], config_id)
```

```
from spotpython.utils.eda import generate_config_id
if timestamp:
    # config id is unique. Since the model is not loaded from a checkpoint,
    # the config id is generated here with a timestamp.
    config_id = generate_config_id(config, timestamp=True)
else:
    # config id is not time-dependent and therefore unique,
    # so that the model can be loaded from a checkpoint,
    # the config id is generated here without a timestamp.
    config_id = generate_config_id(config, timestamp=False) + "_TRAIN"
```

```
from pytorch_lightning.loggers import TensorBoardLogger
import lightning as L
import os
trainer = L.Trainer(
    # Where to save models
    default_root_dir=os.path.join(fun_control["CHECKPOINT_PATH"], config_id),
    max_epochs=model.hparams.epochs,
    accelerator=fun_control["accelerator"],
    devices=fun_control["devices"],
    strategy=fun_control["strategy"],
    num_nodes=fun_control["num_nodes"],
```

```
    precision=fun_control["precision"],
    logger=TensorBoardLogger(save_dir=fun_control["TENSORBOARD_PATH"], version=config_
    callbacks=callbacks,
    enable_progress_bar=False,
    num_sanity_val_steps=fun_control["num_sanity_val_steps"],
    log_every_n_steps=fun_control["log_every_n_steps"],
    gradient_clip_val=None,
    gradient_clip_algorithm="norm",
)
```

```
trainer.fit(model=model, datamodule=dm, ckpt_path=None)
```

```
trainer.validate(model=model, datamodule=dm, verbose=True, ckpt_path=None)
```

29.4.3. DataModule

```
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.csvdataset import CSVDataset
import torch
dataset = CSVDataset(csv_file='data.csv', target_column='prognosis', feature_type=torc
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.5)
data_module.setup()
print(f"Training set size: {len(data_module.data_train)}")
```

- Generate the `config` dictionary:

```
from math import inf
import lightning as L
import numpy as np
import os
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.utils.init import fun_control_init
from spotpython.utils.eda import print_exp_table
from spotpython.hyperparameters.values import get_default_hyperparameters_as_array
from spotpython.hyperparameters.values import assign_values, generate_one_config_from_
from spotpython.light.trainmodel import train_model, generate_config_id_with_timestamp
from pytorch_lightning.loggers import TensorBoardLogger
from lightning.pytorch.callbacks.early_stopping import EarlyStopping
from spotpython.data.lightdatamodule import LightDataModule
PREFIX="000"
```

```
data_set = Diabetes()
fun_control = fun_control_init(
    PREFIX=PREFIX,
    save_experiment=True,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1,
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True,
    seed=42,)
print_exp_table(fun_control)
X = get_default_hyperparameters_as_array(fun_control)
# set epochs to 2^8:
X[0, 1] = 10
# set patience to 2^10:
X[0, 7] = 10
print(f"X: {X}")
var_dict = assign_values(X, get_var_name(fun_control))
config = list(generate_one_config_from_var_dict(var_dict, fun_control))[0]
_L_in = fun_control["_L_in"]
_L_out = fun_control["_L_out"]
_L_cond = fun_control["_L_cond"]
_torchmetric = fun_control["_torchmetric"]
model = fun_control["core_model"](**config, _L_in=_L_in, _L_out=_L_out, _L_cond=_L_cond, _torchmetri
dm = LightDataModule(
    dataset=fun_control["data_set"],
    batch_size=config["batch_size"],
    num_workers=fun_control["num_workers"],
    test_size=fun_control["test_size"],
    test_seed=fun_control["test_seed"],
    scaler=fun_control["scaler"],
)
config_id = generate_config_id_with_timestamp(config, timestamp=True)
callbacks = [EarlyStopping(monitor="val_loss", patience=config["patience"], mode="min", strict=False
trainer = L.Trainer(
    default_root_dir=os.path.join(fun_control["CHECKPOINT_PATH"], config_id),
    max_epochs=model.hparams.epochs,
    accelerator=fun_control["accelerator"],
    devices=fun_control["devices"],
    strategy=fun_control["strategy"],
```

```
    num_nodes=fun_control["num_nodes"],
    precision=fun_control["precision"],
    logger=TensorBoardLogger(save_dir=fun_control["TENSORBOARD_PATH"], version=config_
    callbacks=callbacks,
    enable_progress_bar=False,
    num_sanity_val_steps=fun_control["num_sanity_val_steps"],
    log_every_n_steps=fun_control["log_every_n_steps"],
    gradient_clip_val=None,
    gradient_clip_algorithm="norm",
)
trainer.fit(model=model, datamodule=dm, ckpt_path=None)
```

```
from math import inf
import lightning as L
import numpy as np
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.utils.init import fun_control_init
from spotpython.hyperparameters.values import assign_values, generate_one_config_from_
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.utils.scaler import TorchStandardScaler
PREFIX="000"
data_set = Diabetes()
fun_control = fun_control_init(
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1)
X = np.array([[3.0e+00, 5.0, 4.0e+00, 2.0e+00, 1.1e+01, 1.0e-02, 1.0e+00, 1.0e+01, 0.0
  0.0e+00]])
var_dict = assign_values(X, get_var_name(fun_control))
config = list(generate_one_config_from_var_dict(var_dict, fun_control))[0]
_torchmetric = "mean_squared_error"
model = fun_control["core_model"](**config, _L_in=10, _L_out=1, _L_cond=None, _torchme
dm = LightDataModule(
    dataset=data_set,
    batch_size=16,
    test_size=0.6,
    scaler=TorchStandardScaler())
trainer = L.Trainer(
```

```
    max_epochs=32,
    enable_progress_bar=False,
)
trainer.fit(model=model, datamodule=dm, ckpt_path=None)
trainer.validate(model=model, datamodule=dm, ckpt_path=None)
```

30. User Specified Basic Lightning Module With spotpython

30.1. Introduction

This chapter implements a user-defined DataModule and a user-defined neural network. Remember, that a `LightningModule` organizes your `PyTorch` code into six sections:

- Initialization (`__init__` and `setup()`).
- Train Loop (`training_step()`)
- Validation Loop (`validation_step()`)
- Test Loop (`test_step()`)
- Prediction Loop (`predict_step()`)
- Optimizers and LR Schedulers (`configure_optimizers()`)

The `Trainer` automates every required step in a clear and reproducible way. It is the most important part of PyTorch Lightning. It is responsible for training, testing, and validating the model. The `Lightning` core structure looks like this:

```
import pandas as pd
df = pd.read_pickle("./userData/Turbo_Charger_Data.pkl")
df = df.drop(columns=["M", "R"])
print(f"Features des DataFrames: {df.columns}")
print(df.shape)
```

30.1.1. Dataset

```
from sklearn.preprocessing import LabelEncoder
from lightning import LightningDataModule
import torch
from torch.utils.data import Dataset, DataLoader, random_split

class UserDataset(Dataset):
    def __init__(self, data, y_varname="N", x_varnames=None, dtype=torch.float32):
        """
```

```
        Args:
            data (pd.DataFrame):
                The user data. for example,
                generated by the `preprocess_data` function.
            y_varname (str):
                The name of the target variable.
                Default is "N".
            x_varnames (list):
                The names of the input variables.
                Default is `None`, which means all columns
                except the target variable are used.
            dtype (torch.dtype):
                The data type for the tensors.
                Default is `torch.float32`.

        Examples:
            >>> dataset = UserDataset(data)
            >>> x, y = dataset[0]
        """
        self.data = data.reset_index(drop=True)
        if x_varnames is not None:
            self.x_varnames = x_varnames
        else:
            self.x_varnames = [col for col in self.data.columns if col != y_varname]
        print(f"X variables: {self.x_varnames}")
        print(f"Y variable: {y_varname}")
        self.y_varname = y_varname
        self.dtype = dtype
        self.encoders = {}

        for var in self.x_varnames:
            if self.data[var].dtype == "object" or isinstance(self.data[var][0], str)
                le = LabelEncoder()
                self.data[var] = le.fit_transform(self.data[var])
                self.encoders[var] = le

        if self.data[self.y_varname].dtype == "object" or isinstance(self.data[self.y_
            le = LabelEncoder()
            self.data[self.y_varname] = le.fit_transform(self.data[self.y_varname])
            self.encoders[self.y_varname] = le

        # Convert entire dataset to tensors
        self.features = torch.tensor(self.data[self.x_varnames].values, dtype=self.dty
        self.targets = torch.tensor(self.data[self.y_varname].values, dtype=self.dtype
```

```
    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.features[idx], self.targets[idx]
```

```
dataset = UserDataset(df)
x, y = dataset[0]
print(x)
print(y)
```

30.1.2. DataModule

```
import lightning as L
import torch
from torch.utils.data import DataLoader, random_split, TensorDataset
from typing import Optional
from math import floor


class LightDataModule(L.LightningDataModule):
    """
    A LightningDataModule for handling data.

    Args:
        batch_size (int):
            The batch size. Required.
        dataset (torch.utils.data.Dataset, optional):
            The dataset from the torch.utils.data Dataset class.
            It must implement three functions: __init__, __len__, and __getitem__.
        test_size (float, optional):
            The test size. If test_size is float, then train_size is 1 - test_size.
            If test_size is int, then train_size is len(data_full) - test_size.
        test_seed (int):
            The test seed. Defaults to 42.
        num_workers (int):
            The number of workers. Defaults to 0.
        verbosity (int):
            The verbosity level. Defaults to 0.

    Examples:
```

```
        >>> from spotpython.data.lightdatamodule import LightDataModule
            from spotpython.data.csvdataset import CSVDataset
            from spotpython.utils.scaler import TorchStandardScaler
            import torch
            # data.csv is simple csv file with 11 samples
            dataset = CSVDataset(csv_file='data.csv', target_column='prognosis', featu
            scaler = TorchStandardScaler()
            data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.5
            data_module.setup()
            print(f"Training set size: {len(data_module.data_train)}")
            print(f"Validation set size: {len(data_module.data_val)}")
            print(f"Test set size: {len(data_module.data_test)}")
            full_train_size: 0.5
            val_size: 0.25
            train_size: 0.25
            test_size: 0.5
            Training set size: 3
            Validation set size: 3
            Test set size: 6

    References:
        See https://lightning.ai/docs/pytorch/stable/data/datamodule.html

    """

    def __init__(
        self,
        batch_size: int,
        dataset: Optional[object] = None,
        test_size: Optional[float] = None,
        test_seed: int = 42,
        num_workers: int = 0,
        verbosity: int = 0,
    ):
        super().__init__()
        self.batch_size = batch_size
        self.data_full = dataset
        self.test_size = test_size
        self.test_seed = test_seed
        self.num_workers = num_workers
        self.verbosity = verbosity

    def prepare_data(self) -> None:
        """Prepares the data for use."""
```

```
        # download
        pass

    def _setup_full_data_provided(self, stage) -> None:
        full_size = len(self.data_full)
        test_size = self.test_size

        # consider the case when test_size is a float
        if isinstance(self.test_size, float):
            full_train_size = 1.0 - self.test_size
            val_size = full_train_size * self.test_size
            train_size = full_train_size - val_size
        else:
            # test_size is an int, training size calculation directly based on it
            full_train_size = full_size - self.test_size
            val_size = floor(full_train_size * self.test_size / full_size)
            train_size = full_size - val_size - test_size

        # Assign train/val datasets for use in dataloaders
        if stage == "fit" or stage is None:
            generator_fit = torch.Generator().manual_seed(self.test_seed)
            self.data_train, self.data_val, _ = random_split(self.data_full, [train_size, val_size,
            if self.verbosity > 0:
                print(f"train_size: {train_size}, val_size: {val_size}, test_sie: {test_size} for sp
                print(f"train samples: {len(self.data_train)}, val samples: {len(self.data_val)} gen

        # Assign test dataset for use in dataloader(s)
        if stage == "test" or stage is None:
            generator_test = torch.Generator().manual_seed(self.test_seed)
            self.data_test, _, _ = random_split(self.data_full, [test_size, train_size, val_size], g
            if self.verbosity > 0:
                print(f"train_size: {train_size}, val_size: {val_size}, test_sie: {test_size} for sp
                print(f"test samples: {len(self.data_test)} generated for test data.")

        # Assign pred dataset for use in dataloader(s)
        if stage == "predict" or stage is None:
            generator_predict = torch.Generator().manual_seed(self.test_seed)
            self.data_predict, _, _ = random_split(self.data_full, [test_size, train_size, val_size]
            if self.verbosity > 0:
                print(f"train_size: {train_size}, val_size: {val_size}, test_size (= predict_size):
                print(f"predict samples: {len(self.data_predict)} generated for train & val data.")


    def setup(self, stage: Optional[str] = None) -> None:
```

```
        """
        Splits the data for use in training, validation, and testing.
        Uses torch.utils.data.random_split() to split the data.
        Splitting is based on the test_size and test_seed.
        The test_size can be a float or an int.
        If a spotpython scaler object is defined, the data will be scaled.

        Args:
            stage (Optional[str]):
                The current stage. Can be "fit" (for training and validation), "test"
                or None (for all three stages). Defaults to None.

        Examples:
            >>> from spotpython.data.lightdatamodule import LightDataModule
                from spotpython.data.csvdataset import CSVDataset
                import torch
                dataset = CSVDataset(csv_file='data.csv', target_column='prognosis', i
                data_module = LightDataModule(dataset=dataset, batch_size=5, test_size
                data_module.setup()
                print(f"Training set size: {len(data_module.data_train)}")
                Training set size: 3

        """
        self._setup_full_data_provided(stage)


    def train_dataloader(self) -> DataLoader:
        """
        Returns the training dataloader, i.e., a pytorch DataLoader instance
        using the training dataset.

        Returns:
            DataLoader: The training dataloader.

        Examples:
            >>> from spotpython.data.lightdatamodule import LightDataModule
                from spotpython.data.csvdataset import CSVDataset
                import torch
                dataset = CSVDataset(csv_file='data.csv', target_column='prognosis', i
                data_module = LightDataModule(dataset=dataset, batch_size=5, test_size
                data_module.setup()
                print(f"Training set size: {len(data_module.data_train)}")
                Training set size: 3
```

```
        """
        if self.verbosity > 0:
            print(f"LightDataModule.train_dataloader(). data_train size: {len(self.data_train)}")
        return DataLoader(self.data_train, batch_size=self.batch_size, num_workers=self.num_workers)

    def val_dataloader(self) -> DataLoader:
        """
        Returns the validation dataloader, i.e., a pytorch DataLoader instance
        using the validation dataset.

        Returns:
            DataLoader: The validation dataloader.

        Examples:
            >>> from spotpython.data.lightdatamodule import LightDataModule
                from spotpython.data.csvdataset import CSVDataset
                import torch
                dataset = CSVDataset(csv_file='data.csv', target_column='prognosis', feature_type=to
                data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.5)
                data_module.setup()
                print(f"Training set size: {len(data_module.data_val)}")
                Training set size: 3
        """
        if self.verbosity > 0:
            print(f"LightDataModule.val_dataloader(). Val. set size: {len(self.data_val)}")
        return DataLoader(self.data_val, batch_size=self.batch_size, num_workers=self.num_workers)

    def test_dataloader(self) -> DataLoader:
        """
        Returns the test dataloader, i.e., a pytorch DataLoader instance
        using the test dataset.

        Returns:
            DataLoader: The test dataloader.

        Examples:
            >>> from spotpython.data.lightdatamodule import LightDataModule
                from spotpython.data.csvdataset import CSVDataset
                import torch
                dataset = CSVDataset(csv_file='data.csv', target_column='prognosis', feature_type=to
                data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.5)
                data_module.setup()
                print(f"Test set size: {len(data_module.data_test)}")
                Test set size: 6
```

```
        """
        if self.verbosity > 0:
            print(f"LightDataModule.test_dataloader(). Test set size: {len(self.data_t
        return DataLoader(self.data_test, batch_size=self.batch_size, num_workers=seli

    def predict_dataloader(self) -> DataLoader:
        """
        Returns the predict dataloader, i.e., a pytorch DataLoader instance
        using the predict dataset.

        Returns:
            DataLoader: The predict dataloader.

        Examples:
            >>> from spotpython.data.lightdatamodule import LightDataModule
                from spotpython.data.csvdataset import CSVDataset
                import torch
                dataset = CSVDataset(csv_file='data.csv', target_column='prognosis', f
                data_module = LightDataModule(dataset=dataset, batch_size=5, test_size
                data_module.setup()
                print(f"Predict set size: {len(data_module.data_predict)}")
                Predict set size: 6

        """
        if self.verbosity > 0:
            print(f"LightDataModule.predict_dataloader(). Predict set size: {len(self
        return DataLoader(self.data_predict, batch_size=len(self.data_predict), num_wo
```

```
data_module = LightDataModule(batch_size=2, dataset=dataset, test_size=0.2)
data_module.setup()
for batch in data_module.train_dataloader():
    print(batch)
    print(f"Number of input features: {batch[0][1].shape}")
    break
```

30.2. The Neural Network: MyRegressor

```
import lightning as L
import torch
from torch import nn
from spotpython.hyperparameters.optimizer import optimizer_handler
```

```python
import torchmetrics.functional.regression
from math import ceil

class MyRegressor(L.LightningModule):
    """
    A LightningModule class for a regression neural network model.

    Attributes:
        l1 (int):
            The number of neurons in the first hidden layer.
        epochs (int):
            The number of epochs to train the model for.
        batch_size (int):
            The batch size to use during training.
        initialization (str):
            The initialization method to use for the weights.
        act_fn (nn.Module):
            The activation function to use in the hidden layers.
        optimizer (str):
            The optimizer to use during training.
        dropout_prob (float):
            The probability of dropping out a neuron during training.
        lr_mult (float):
            The learning rate multiplier for the optimizer.
        patience (int):
            The number of epochs to wait before early stopping.
        _L_in (int):
            The number of input features.
        _L_out (int):
            The number of output classes.
        _torchmetric (str):
            The metric to use for the loss function. If `None`,
            then "mean_squared_error" is used.
        layers (nn.Sequential):
            The neural network model.

    """

    def __init__(
        self,
        l1: int,
        epochs: int,
        batch_size: int,
        initialization: str,
```

```
        act_fn: nn.Module,
        optimizer: str,
        dropout_prob: float,
        lr_mult: float,
        patience: int,
        _L_in: int,
        _L_out: int,
        _torchmetric: str,
        *args,
        **kwargs,
    ):
        """
        Initializes the MyRegressor object.

        Args:
            l1 (int):
                The number of neurons in the first hidden layer.
            epochs (int):
                The number of epochs to train the model for.
            batch_size (int):
                The batch size to use during training.
            initialization (str):
                The initialization method to use for the weights.
            act_fn (nn.Module):
                The activation function to use in the hidden layers.
            optimizer (str):
                The optimizer to use during training.
            dropout_prob (float):
                The probability of dropping out a neuron during training.
            lr_mult (float):
                The learning rate multiplier for the optimizer.
            patience (int):
                The number of epochs to wait before early stopping.
            _L_in (int):
                The number of input features. Not a hyperparameter, but needed to crea
            _L_out (int):
                The number of output classes. Not a hyperparameter, but needed to crea
            _torchmetric (str):
                The metric to use for the loss function. If `None`,
                then "mean_squared_error" is used.

        Returns:
            (NoneType): None
```

```
        Raises:
            ValueError: If l1 is less than 4.

        """
        super().__init__()
        self._L_in = _L_in
        self._L_out = _L_out
        if _torchmetric is None:
            _torchmetric = "mean_squared_error"
        self._torchmetric = _torchmetric
        self.metric = getattr(torchmetrics.functional.regression, _torchmetric)
        # _L_in and _L_out are not hyperparameters, but are needed to create the network
        # _torchmetric is not a hyperparameter, but is needed to calculate the loss
        self.save_hyperparameters(ignore=["_L_in", "_L_out", "_torchmetric"])
        # set dummy input array for Tensorboard Graphs
        # set log_graph=True in Trainer to see the graph (in traintest.py)
        self.example_input_array = torch.zeros((batch_size, self._L_in))
        if self.hparams.l1 < 4:
            raise ValueError("l1 must be at least 4")
        hidden_sizes = [l1 * 2, l1, ceil(l1/2)]
        # Create the network based on the specified hidden sizes
        layers = []
        layer_sizes = [self._L_in] + hidden_sizes
        layer_size_last = layer_sizes[0]
        for layer_size in layer_sizes[1:]:
            layers += [
                nn.Linear(layer_size_last, layer_size),
                self.hparams.act_fn,
                nn.Dropout(self.hparams.dropout_prob),
            ]
            layer_size_last = layer_size
        layers += [nn.Linear(layer_sizes[-1], self._L_out)]
        self.layers = nn.Sequential(*layers)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Performs a forward pass through the model.

        Args:
            x (torch.Tensor): A tensor containing a batch of input data.

        Returns:
            torch.Tensor: A tensor containing the output of the model.
```

```
        """
        x = self.layers(x)
        return x

    def _calculate_loss(self, batch):
        """
        Calculate the loss for the given batch.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.

        """
        x, y = batch
        y = y.view(len(y), 1)
        y_hat = self(x)
        loss = self.metric(y_hat, y)
        return loss

    def training_step(self, batch: tuple) -> torch.Tensor:
        """
        Performs a single training step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.

        """
        val_loss = self._calculate_loss(batch)
        return val_loss

    def validation_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) ->
        """
        Performs a single validation step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults t
```

```
        Returns:
            torch.Tensor: A tensor containing the loss for this batch.

        """
        val_loss = self._calculate_loss(batch)
        self.log("val_loss", val_loss, prog_bar=prog_bar)
        self.log("hp_metric", val_loss, prog_bar=prog_bar)
        return val_loss

    def test_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> torch.Tensor:
        """
        Performs a single test step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults to False.

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.
        """
        val_loss = self._calculate_loss(batch)
        self.log("val_loss", val_loss, prog_bar=prog_bar)
        self.log("hp_metric", val_loss, prog_bar=prog_bar)
        return val_loss

    def predict_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> torch.Tensor:
        """
        Performs a single prediction step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults to False.

        Returns:
            A tuple containing the input data, the true labels, and the predicted values.
        """
        x, y = batch
        yhat = self(x)
        y = y.view(len(y), 1)
        yhat = yhat.view(len(yhat), 1)
        print(f"Predict step x: {x}")
        print(f"Predict step y: {y}")
```

```
        print(f"Predict step y_hat: {yhat}")
        return (x, y, yhat)

    def configure_optimizers(self) -> torch.optim.Optimizer:
        """
        Configures the optimizer for the model.
        Simple examples use the following code here:
        `optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)`

        Notes:
            The default Lightning way is to define an optimizer as
            `optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)`.
            spotpython uses an optimizer handler to create the optimizer, which
            adapts the learning rate according to the lr_mult hyperparameter as
            well as other hyperparameters. See `spotpython.hyperparameters.optimizer.p

        Returns:
            torch.optim.Optimizer: The optimizer to use during training.

        """
        # optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)
        optimizer = optimizer_handler(
            optimizer_name=self.hparams.optimizer, params=self.parameters(), lr_mult=s
        )
        return optimizer
```

30.3. Calling the Neural Network With spotpython

```
PREFIX="702_lightning_user_datamodule"
```

```
import sys
sys.path.insert(0, './userModel')
import my_regressor
import my_hyper_dict
```

```
from spotpython.hyperparameters.values import add_core_model_to_fun_control
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_co
from spotpython.hyperparameters.values import set_hyperparameter
from spotpython.spot import Spot
```

```
from math import inf
fun_control = fun_control_init(
    PREFIX=PREFIX,
    fun_evals=inf,
    fun_repeats=1,
    max_time=5,
    accelerator="cpu",
    data_module=data_module,
    _L_in=dataset[0][0].shape[0],
    _L_out=1,
    noise=False,
    ocba_delta=0,
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True,
    _torchmetric="mean_squared_error",
    log_level=50,
    save_experiment=True,
    verbosity=1)

add_core_model_to_fun_control(fun_control=fun_control,
                              core_model=my_regressor.MyRegressor,
                              hyper_dict=my_hyper_dict.MyHyperDict)

set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "act_fn", [ "ReLU", "Swish", "LeakyReLU"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,5])
set_hyperparameter(fun_control, "batch_size", [1,5])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,3])
# set_hyperparameter(fun_control, "initialization", ["Default"])

design_control = design_control_init(init_size=5, repeats=1)
surrogate_control = surrogate_control_init(noise=True)

fun = HyperLight().fun

spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control, surrogate_control=
```

```
import os
from spotpython.utils.file import load_experiment
if os.path.exists("spot_" + PREFIX + "_experiment.pickle"):
    (spot_tuner, fun_control, design_control,
```

```
        surrogate_control, optimizer_control) = load_experiment(PREFIX=PREFIX)
else:
    res = spot_tuner.run()
```

30.4. Looking at the Results

30.4.1. Tuning Progress

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with `spotpython`'s method `plot_progress`. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress()
```

30.4.2. Tuned Hyperparameters and Their Importance

Results can be printed in tabular form.

```
from spotpython.utils.eda import print_res_table
print_res_table(spot_tuner)
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=1.0)
```

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

30.4.3. Get the Tuned Architecture

```
import pprint
from spotpython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(spot_tuner)
pprint.pprint(config)
```

31. HPT PyTorch Lightning: Data

In this tutorial, we will show how `spotpython` can be integrated into the `PyTorch` Lightning training workflow.

This chapter describes the data preparation and processing in `spotpython`. The Diabetes data set is used as an example. This is a PyTorch Dataset for regression. A toy data set from scikit-learn. Ten baseline variables, age, sex, body mass index, average blood pressure, and six blood serum measurements were obtained for each of n = 442 diabetes patients, as well as the response of interest, a quantitative measure of disease progression one year after baseline.

31.1. Setup

- Before we consider the detailed experimental setup, we select the parameters that affect run time, initial design size, etc.
- The parameter `WORKERS` specifies the number of workers.
- The prefix `PREFIX` is used for the experiment name and the name of the log file.
- The parameter `DEVICE` specifies the device to use for training.

```
import torch
from spotpython.utils.device import getDevice
from math import inf
WORKERS = 0
PREFIX="030"
DEVICE = getDevice()
DEVICES = 1
TEST_SIZE = 0.4
```

> **Note: Device selection**
>
> - Although there are no .cuda() or .to(device) calls required, because Lightning does these for you, see LIGHTNINGMODULE, we would like to know which device is used. Threrefore, we imitate the LightningModule behaviour which selects the highest device.
> - The method `spotpython.utils.device.getDevice()` returns the device that is used by Lightning.

31.2. Initialization of the `fun_control` Dictionary

`spotpython` uses a Python dictionary for storing the information required for the hyperparameter tuning process.

```
from spotpython.utils.init import fun_control_init
import numpy as np
fun_control = fun_control_init(
    _L_in=10,
    _L_out=1,
    _torchmetric="mean_squared_error",
    PREFIX=PREFIX,
    device=DEVICE,
    enable_progress_bar=False,
    num_workers=WORKERS,
    show_progress=True,
    test_size=TEST_SIZE,
    )
```

31.3. Loading the Diabetes Data Set

Here, we load the Diabetes data set from `spotpython`'s `data` module.

```
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
print(len(dataset))
```

```
442
```

31.3.1. Data Set and Data Loader

As shown below, a DataLoader from `torch.utils.data` can be used to check the data.

```
# Set batch size for DataLoader
batch_size = 5
# Create DataLoader
from torch.utils.data import DataLoader
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=False)

# Iterate over the data in the DataLoader
```

```
for batch in dataloader:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0381,  0.0507,  0.0617,  0.0219, -0.0442, -0.0348, -0.0434, -0.0026,
          0.0199, -0.0176],
        [-0.0019, -0.0446, -0.0515, -0.0263, -0.0084, -0.0192,  0.0744, -0.0395,
         -0.0683, -0.0922],
        [ 0.0853,  0.0507,  0.0445, -0.0057, -0.0456, -0.0342, -0.0324, -0.0026,
          0.0029, -0.0259],
        [-0.0891, -0.0446, -0.0116, -0.0367,  0.0122,  0.0250, -0.0360,  0.0343,
          0.0227, -0.0094],
        [ 0.0054, -0.0446, -0.0364,  0.0219,  0.0039,  0.0156,  0.0081, -0.0026,
         -0.0320, -0.0466]])
Targets: tensor([151.,  75., 141., 206., 135.])
```

31.3.2. Preparing Training, Validation, and Test Data

The following code shows how to split the data into training, validation, and test sets. Then a Lightning Trainer is used to train (`fit`) the model, validate it, and test it.

```
from torch.utils.data import DataLoader
from spotpython.data.diabetes import Diabetes
from spotpython.light.regression.netlightregression import NetLightRegression
from torch import nn
import lightning as L
import torch
BATCH_SIZE = 8
dataset = Diabetes(target_type=torch.float)
train1_set, test_set = torch.utils.data.random_split(dataset, [0.6, 0.4])
train_set, val_set = torch.utils.data.random_split(train1_set, [0.6, 0.4])
train_loader = DataLoader(train_set, batch_size=BATCH_SIZE, shuffle=True, drop_last=True, pin_memory
test_loader = DataLoader(test_set, batch_size=BATCH_SIZE)
```

```
val_loader = DataLoader(val_set, batch_size=BATCH_SIZE)
batch_x, batch_y = next(iter(train_loader))
print(f"batch_x.shape: {batch_x.shape}")
print(f"batch_y.shape: {batch_y.shape}")
net_light_base = NetLightRegression(l1=128,
                                    epochs=10,
                                    batch_size=BATCH_SIZE,
                                    initialization='Default',
                                    act_fn=nn.ReLU(),
                                    optimizer='Adam',
                                    dropout_prob=0.1,
                                    lr_mult=0.1,
                                    patience=5,
                                    _L_in=10,
                                    _L_out=1,
                                    _torchmetric="mean_squared_error")
trainer = L.Trainer(max_epochs=10,  enable_progress_bar=False)
trainer.fit(net_light_base, train_loader)
trainer.validate(net_light_base, val_loader)
trainer.test(net_light_base, test_loader)
```

```
batch_x.shape: torch.Size([8, 10])
batch_y.shape: torch.Size([8])
```

```
       Validate metric              DataLoader 0

          hp_metric                32421.490234375
          val_loss                 32421.490234375
```

```
         Test metric                DataLoader 0

          hp_metric                 25781.0234375
          val_loss                  25781.0234375
```

```
[{'val_loss': 25781.0234375, 'hp_metric': 25781.0234375}]
```

31.3.3. Dataset for spotpython

spotpython handles the data set, which is added to the fun_control dictionary with the key data_set as follows:

```
from spotpython.hyperparameters.values import set_control_key_value
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
set_control_key_value(control_dict=fun_control,
                        key="data_set",
                        value=dataset,
                        replace=True)
print(len(dataset))
```

```
442
```

If the data set is in the fun_control dictionary, it is used to create a LightDataModule object. This object is used to create the data loaders for the training, validation, and test sets. Therefore, the following information must be provided in the fun_control dictionary:

- data_set: the data set
- batch_size: the batch size
- num_workers: the number of workers
- test_size: the size of the test set
- test_seed: the seed for the test set

```
from spotpython.utils.init import fun_control_init
import numpy as np
fun_control = fun_control_init(
    data_set=dataset,
    device="cpu",
    enable_progress_bar=False,
    num_workers=0,
    show_progress=True,
    test_size=0.4,
    test_seed=42,
    )
```

```
from spotpython.data.lightdatamodule import LightDataModule
dm = LightDataModule(
    dataset=fun_control["data_set"],
    batch_size=8,
    num_workers=fun_control["num_workers"],
```

```
    test_size=fun_control["test_size"],
    test_seed=fun_control["test_seed"],
)
dm.setup()
print(f"train_model(): Test set size: {len(dm.data_test)}")
print(f"train_model(): Train set size: {len(dm.data_train)}")
```

```
train_model(): Test set size: 177
train_model(): Train set size: 160
```

31.4. The LightDataModule

The steps described above are handled by the `LightDataModule` class. This class is used to create the data loaders for the training, validation, and test sets. The `LightDataModule` class is part of the `spotpython` package. The `LightDataModule` class provides the following methods:

- `prepare_data()`: This method is used to prepare the data set.
- `setup()`: This method is used to create the data loaders for the training, validation, and test sets.
- `train_dataloader()`: This method is used to return the data loader for the training set.
- `val_dataloader()`: This method is used to return the data loader for the validation set.
- `test_dataloader()`: This method is used to return the data loader for the test set.
- `predict_dataloader()`: This method is used to return the data loader for the prediction set.

31.4.1. The `prepare_data()` Method

The `prepare_data()` method is used to prepare the data set. This method is called only once and on a single process. It can be used to download the data set. In our case, the data set is already available, so this method uses a simple `pass` statement.

31.4.2. The `setup()` Method

Splits the data for use in training, validation, and testing. It uses `torch.utils.data.random_split()` to split the data. Splitting is based on the `test_size` and `test_seed`. The `test_size` can be a float or an int.

31.4.2.1. Determine the Sizes of the Data Sets

```
from torch.utils.data import random_split
data_full = dataset
test_size = fun_control["test_size"]
test_seed=fun_control["test_seed"]
# if test_size is float, then train_size is 1 - test_size
if isinstance(test_size, float):
    full_train_size = round(1.0 - test_size, 2)
    val_size = round(full_train_size * test_size, 2)
    train_size = round(full_train_size - val_size, 2)
else:
    # if test_size is int, then train_size is len(data_full) - test_size
    full_train_size = len(data_full) - test_size
    val_size = int(full_train_size * test_size / len(data_full))
    train_size = full_train_size - val_size

print(f"LightDataModule setup(): full_train_size: {full_train_size}")
print(f"LightDataModule setup(): val_size: {val_size}")
print(f"LightDataModule setup(): train_size: {train_size}")
print(f"LightDataModule setup(): test_size: {test_size}")
```

```
LightDataModule setup(): full_train_size: 0.6
LightDataModule setup(): val_size: 0.24
LightDataModule setup(): train_size: 0.36
LightDataModule setup(): test_size: 0.4
```

`stage` is used to define the data set to be returned. The `stage` can be `None`, `fit`, `test`, or `predict`. If `stage` is `None`, the method returns the training (`fit`), testing (`test`) and prediction (`predict`) data sets.

31.4.2.2. Stage "fit"

```
stage = "fit"
if stage == "fit" or stage is None:
    generator_fit = torch.Generator().manual_seed(test_seed)
    data_train, data_val, _ = random_split(data_full, [train_size, val_size, test_size], generator=g
print(f"LightDataModule setup(): Train set size: {len(data_train)}")
print(f"LightDataModule setup(): Validation set size: {len(data_val)}")
```

```
LightDataModule setup(): Train set size: 160
LightDataModule setup(): Validation set size: 106
```

31.4.2.3. Stage "test"

```
stage = "test"
if stage == "test" or stage is None:
    generator_test = torch.Generator().manual_seed(test_seed)
    data_test, _ = random_split(data_full, [test_size, full_train_size], generator=gen
print(f"LightDataModule setup(): Test set size: {len(data_test)}")
# Set batch size for DataLoader
batch_size = 5
# Create DataLoader
from torch.utils.data import DataLoader
dataloader = DataLoader(data_test, batch_size=batch_size, shuffle=False)
# Iterate over the data in the DataLoader
for batch in dataloader:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
LightDataModule setup(): Test set size: 177
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0562, -0.0446, -0.0579, -0.0080,  0.0521,  0.0491,  0.0560, -0.021
         -0.0283,  0.0445],
        [ 0.0018, -0.0446, -0.0709, -0.0229, -0.0016, -0.0010,  0.0266, -0.0395,
         -0.0225,  0.0072],
        [-0.0527, -0.0446,  0.0542, -0.0263, -0.0552, -0.0339, -0.0139, -0.0395,
         -0.0741, -0.0591],
        [ 0.0054, -0.0446, -0.0482, -0.0126,  0.0012, -0.0066,  0.0634, -0.0395,
         -0.0514, -0.0591],
        [-0.0527, -0.0446, -0.0094, -0.0057,  0.0397,  0.0447,  0.0266, -0.0026,
         -0.0181, -0.0135]])
Targets: tensor([158.,  49., 142.,  96.,  59.])
```

31.4.2.4. Stage "predict"

Prediction and testing use the same data set.

```python
stage = "predict"
if stage == "predict" or stage is None:
    generator_predict = torch.Generator().manual_seed(test_seed)
    data_predict, _ = random_split(
        data_full, [test_size, full_train_size], generator=generator_predict
    )
print(f"LightDataModule setup(): Predict set size: {len(data_predict)}")
# Set batch size for DataLoader
batch_size = 5
# Create DataLoader
from torch.utils.data import DataLoader
dataloader = DataLoader(data_predict, batch_size=batch_size, shuffle=False)
# Iterate over the data in the DataLoader
for batch in dataloader:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
LightDataModule setup(): Predict set size: 177
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0562, -0.0446, -0.0579, -0.0080,  0.0521,  0.0491,  0.0560, -0.0214,
         -0.0283,  0.0445],
        [ 0.0018, -0.0446, -0.0709, -0.0229, -0.0016, -0.0010,  0.0266, -0.0395,
         -0.0225,  0.0072],
        [-0.0527, -0.0446,  0.0542, -0.0263, -0.0552, -0.0339, -0.0139, -0.0395,
         -0.0741, -0.0591],
        [ 0.0054, -0.0446, -0.0482, -0.0126,  0.0012, -0.0066,  0.0634, -0.0395,
         -0.0514, -0.0591],
        [-0.0527, -0.0446, -0.0094, -0.0057,  0.0397,  0.0447,  0.0266, -0.0026,
         -0.0181, -0.0135]])
Targets: tensor([158.,  49., 142.,  96.,  59.])
```

31.4.3. The `train_dataloader()` Method

Returns the training dataloader, i.e., a Pytorch DataLoader instance using the training dataset. It simply returns a DataLoader with the `data_train` set that was created in the `setup()` method as described in Section 31.4.2.2.

```
def train_dataloader(self) -> DataLoader:
    return DataLoader(self.data_train, batch_size=self.batch_size, num_workers=self.nu
```

The `train_dataloader()` method can be used as follows:

```
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
print(f"Training set size: {len(data_module.data_train)}")
dl = data_module.train_dataloader()
# Iterate over the data in the DataLoader
for batch in dl:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Training set size: 160
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0708,  0.0507, -0.0310,  0.0219, -0.0373, -0.0470,  0.0339, -0.039
         -0.0150, -0.0011],
        [ 0.0344,  0.0507, -0.0299,  0.0047,  0.0934,  0.0870,  0.0339, -0.0026,
          0.0241, -0.0384],
        [ 0.0090, -0.0446,  0.0552, -0.0057,  0.0576,  0.0447, -0.0029,  0.0232,
          0.0557,  0.1066],
        [ 0.0417, -0.0446, -0.0644,  0.0356,  0.0122, -0.0580,  0.1812, -0.0764,
         -0.0006, -0.0508],
        [ 0.0381,  0.0507,  0.0164,  0.0219,  0.0397,  0.0450, -0.0434,  0.0712,
          0.0498,  0.0155]])
Targets: tensor([ 66.,  69., 173., 170., 212.])
```

31.4.4. The `val_dataloader()` Method

Returns the validation dataloader, i.e., a Pytorch DataLoader instance using the validation dataset. It simply returns a DataLoader with the `data_val` set that was created in the `setup()` method as desccribed in Section 31.4.2.2.

```
def val_dataloader(self) -> DataLoader:
    return DataLoader(self.data_val, batch_size=self.batch_size, num_workers=self.num_workers)
```

The `val_dataloader()` method can be used as follows:

```
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
print(f"Validation set size: {len(data_module.data_val)}")
dl = data_module.val_dataloader()
# Iterate over the data in the DataLoader
for batch in dl:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Validation set size: 106
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0163, -0.0446,  0.0736, -0.0412, -0.0043, -0.0135, -0.0139, -0.0011,
          0.0429,  0.0445],
        [ 0.0453, -0.0446,  0.0714,  0.0012, -0.0098, -0.0010,  0.0155, -0.0395,
         -0.0412, -0.0715],
        [ 0.0308,  0.0507,  0.0326,  0.0494, -0.0401, -0.0436, -0.0692,  0.0343,
          0.0630,  0.0031],
        [ 0.0235,  0.0507, -0.0396, -0.0057, -0.0484, -0.0333,  0.0118, -0.0395,
         -0.1016, -0.0674],
        [-0.0091,  0.0507,  0.0013, -0.0022,  0.0796,  0.0701,  0.0339, -0.0026,
          0.0267,  0.0818]])
Targets: tensor([275., 141., 208.,  78., 142.])
```

31.4.5. The `test_dataloader()` Method

Returns the test dataloader, i.e., a Pytorch DataLoader instance using the test dataset. It simply returns a DataLoader with the `data_test` set that was created in the `setup()` method as described in Section 31.4.2.3.

```
def test_dataloader(self) -> DataLoader:
    return DataLoader(self.data_test, batch_size=self.batch_size, num_workers=self.nu
```

The `test_dataloader()` method can be used as follows:

```
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
print(f"Test set size: {len(data_module.data_test)}")
dl = data_module.test_dataloader()
# Iterate over the data in the DataLoader
for batch in dl:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Test set size: 177
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0562, -0.0446, -0.0579, -0.0080,  0.0521,  0.0491,  0.0560, -0.021
         -0.0283,  0.0445],
        [ 0.0018, -0.0446, -0.0709, -0.0229, -0.0016, -0.0010,  0.0266, -0.0395,
         -0.0225,  0.0072],
        [-0.0527, -0.0446,  0.0542, -0.0263, -0.0552, -0.0339, -0.0139, -0.0395,
         -0.0741, -0.0591],
        [ 0.0054, -0.0446, -0.0482, -0.0126,  0.0012, -0.0066,  0.0634, -0.0395,
         -0.0514, -0.0591],
        [-0.0527, -0.0446, -0.0094, -0.0057,  0.0397,  0.0447,  0.0266, -0.0026,
         -0.0181, -0.0135]])
Targets: tensor([158.,  49., 142.,  96.,  59.])
```

31.4.6. The `predict_dataloader()` Method

Returns the prediction dataloader, i.e., a Pytorch DataLoader instance using the prediction dataset. It simply returns a DataLoader with the `data_predict` set that was created in the `setup()` method as described in Section 31.4.2.4.

> ⚠ Warning
>
> The `batch_size` is set to the length of the `data_predict` set.

```
def predict_dataloader(self) -> DataLoader:
    return DataLoader(self.data_predict, batch_size=len(self.data_predict), num_workers=self.num_wor
```

The `predict_dataloader()` method can be used as follows:

```
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
print(f"Test set size: {len(data_module.data_predict)}")
dl = data_module.predict_dataloader()
# Iterate over the data in the DataLoader
for batch in dl:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Test set size: 177
Batch Size: 177
Inputs Shape: torch.Size([177, 10])
Targets Shape: torch.Size([177])
---------------
Inputs: tensor([[ 0.0562, -0.0446, -0.0579,  ..., -0.0214, -0.0283,  0.0445],
        [ 0.0018, -0.0446, -0.0709,  ..., -0.0395, -0.0225,  0.0072],
        [-0.0527, -0.0446,  0.0542,  ..., -0.0395, -0.0741, -0.0591],
        ...,
        [ 0.0090, -0.0446, -0.0321,  ..., -0.0764, -0.0119, -0.0384],
```

```
        [-0.0273, -0.0446, -0.0666,  ..., -0.0395, -0.0358, -0.0094],
        [ 0.0817,  0.0507,  0.0067,  ...,  0.0919,  0.0547,  0.0072]])
Targets: tensor([158.,  49., 142.,  96.,  59.,  74., 137., 136.,  39.,  66., 310., 198
        235., 116.,  55., 177.,  59., 246.,  53., 135.,  88., 198., 186., 217.,
         51., 118., 153., 180.,  51., 229.,  84.,  72., 237., 142., 185.,  91.,
         88., 148., 179., 144.,  25.,  89.,  42.,  60., 124., 170., 215., 263.,
        178., 245., 202.,  97., 321.,  71., 123., 220., 132., 243.,  61., 102.,
        187.,  70., 242., 134.,  63.,  72.,  88., 219., 127., 146., 122., 143.,
        220., 293.,  59., 317.,  60., 140.,  65., 277.,  90.,  96., 109., 190.,
         90.,  52., 160., 233., 230., 175.,  68., 272., 144.,  70.,  68., 163.,
         71.,  93., 263., 118., 220.,  90., 232., 120., 163.,  88.,  85.,  52.,
        181., 232., 212., 332.,  81., 214., 145., 268., 115.,  93.,  64., 156.,
        128., 200., 281., 103., 220.,  66.,  48., 246.,  42., 150., 125., 109.,
        129.,  97., 265.,  97., 173., 216., 237., 121.,  42., 151.,  31.,  68.,
        137., 221., 283., 124., 243., 150.,  69., 306., 182., 252., 132., 258.,
        121., 110., 292., 101., 275., 141., 208.,  78., 142., 185., 167., 258.,
        144.,  89., 225., 140., 303., 236.,  87.,  77., 131.])
```

31.5. Using the `LightDataModule` in the `train_model()` Method

First, a `LightDataModule` object is created and the `setup()` method is called.

```
dm = LightDataModule(
    dataset=fun_control["data_set"],
    batch_size=config["batch_size"],
    num_workers=fun_control["num_workers"],
    test_size=fun_control["test_size"],
    test_seed=fun_control["test_seed"],
)
dm.setup()
```

Then, the `Trainer` is initialized.

```
# Init trainer
trainer = L.Trainer(
    default_root_dir=os.path.join(fun_control["CHECKPOINT_PATH"], config_id),
    max_epochs=model.hparams.epochs,
    accelerator=fun_control["accelerator"],
    devices=fun_control["devices"],
    logger=TensorBoardLogger(
        save_dir=fun_control["TENSORBOARD_PATH"],
```

```
        version=config_id,
        default_hp_metric=True,
        log_graph=fun_control["log_graph"],
    ),
    callbacks=[
        EarlyStopping(monitor="val_loss", patience=config["patience"], mode="min", strict=False, ver
    ],
    enable_progress_bar=enable_progress_bar,
)
```

Next, the `fit()` method is called to train the model.

```
# Pass the datamodule as arg to trainer.fit to override model hooks :)
trainer.fit(model=model, datamodule=dm)
```

Finally, the `validate()` method is called to validate the model. The `validate()` method returns the validation loss.

```
# Test best model on validation and test set
# result = trainer.validate(model=model, datamodule=dm, ckpt_path="last")
result = trainer.validate(model=model, datamodule=dm)
# unlist the result (from a list of one dict)
result = result[0]
return result["val_loss"]
```

31.6. Further Information

31.6.1. Preprocessing

Preprocessing is handled by `Lightning` and `PyTorch`. It is described in the LIGHTNINGDATAMODULE documentation. Here you can find information about the `transforms` methods.

32. Hyperparameter Tuning with spotpython and PyTorch Lightning for the Diabetes Data Set

In this section, we will show how spotpython can be integrated into the PyTorch Lightning training workflow for a regression task. It demonstrates how easy it is to use spotpython to tune hyperparameters for a PyTorch Lightning model.

32.1. The Basic Setting

```
import os
from math import inf
import warnings
warnings.filterwarnings("ignore")
```

After importing the necessary libraries, the fun_control dictionary is set up via the fun_control_init function. The fun_control dictionary contains

- PREFIX: a unique identifier for the experiment
- fun_evals: the number of function evaluations
- max_time: the maximum run time in minutes
- data_set: the data set. Here we use the Diabetes data set that is provided by spotpython.
- core_model_name: the class name of the neural network model. This neural network model is provided by spotpython.
- hyperdict: the hyperparameter dictionary. This dictionary is used to define the hyperparameters of the neural network model. It is also provided by spotpython.
- _L_in: the number of input features. Since the Diabetes data set has 10 features, _L_in is set to 10.
- _L_out: the number of output features. Since we want to predict a single value, _L_out is set to 1.

The HyperLight class is used to define the objective function fun. It connects the PyTorch and the spotpython methods and is provided by spotpython.

```
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_co
from spotpython.utils.eda import print_exp_table, print_res_table
from spotpython.spot import Spot
from spotpython.utils.file import get_experiment_filename

PREFIX="601"

data_set = Diabetes()

fun_control = fun_control_init(
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1)

fun = HyperLight().fun
```

```
module_name: light
submodule_name: regression
model_name: NNLinearRegressor
```

The method `set_hyperparameter` allows the user to modify default hyperparameter settings. Here we modify some hyperparameters to keep the model small and to decrease the tuning time.

```
from spotpython.hyperparameters.values import set_hyperparameter
set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,7])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,3])

design_control = design_control_init(init_size=10)
print_exp_table(fun_control)
```

```
| name           | type   | default   |   lower |   upper | transform             |
|----------------|--------|-----------|---------|---------|-----------------------|
| l1             | int    | 3         |     3   |   4     | transform_power_2_int |
| epochs         | int    | 4         |     3   |   7     | transform_power_2_int |
| batch_size     | int    | 4         |     4   |  11     | transform_power_2_int |
| act_fn         | factor | ReLU      |     0   |   5     | None                  |
| optimizer      | factor | SGD       |     0   |   2     | None                  |
| dropout_prob   | float  | 0.01      |     0   |   0.025 | None                  |
| lr_mult        | float  | 1.0       |     0.1 |  10     | None                  |
| patience       | int    | 2         |     2   |   3     | transform_power_2_int |
| batch_norm     | factor | 0         |     0   |   1     | None                  |
| initialization | factor | Default   |     0   |   4     | None                  |
```

Finally, a `Spot` object is created. Calling the method `run()` starts the hyperparameter tuning process.

```
S = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
S.run()
```

```
train_model result: {'val_loss': 23075.09765625, 'hp_metric': 23075.09765625}

train_model result: {'val_loss': 3466.6259765625, 'hp_metric': 3466.6259765625}

train_model result: {'val_loss': 4775.9482421875, 'hp_metric': 4775.9482421875}

train_model result: {'val_loss': 23974.962890625, 'hp_metric': 23974.962890625}

train_model result: {'val_loss': 22921.740234375, 'hp_metric': 22921.740234375}

train_model result: {'val_loss': 4060.3369140625, 'hp_metric': 4060.3369140625}

train_model result: {'val_loss': 20739.09375, 'hp_metric': 20739.09375}

train_model result: {'val_loss': 4064.956298828125, 'hp_metric': 4064.956298828125}

train_model result: {'val_loss': 20911.953125, 'hp_metric': 20911.953125}

train_model result: {'val_loss': 23951.484375, 'hp_metric': 23951.484375}
train_model result: {'val_loss': 3127.83349609375, 'hp_metric': 3127.83349609375}
spotpython tuning: 3127.83349609375 [----------] 1.35%
```

```
train_model result: {'val_loss': 24038.533203125, 'hp_metric': 24038.533203125}
spotpython tuning: 3127.83349609375 [----------] 2.96%

train_model result: {'val_loss': 3949.494140625, 'hp_metric': 3949.494140625}
spotpython tuning: 3127.83349609375 [#---------] 5.07%

train_model result: {'val_loss': 3263.91748046875, 'hp_metric': 3263.91748046875}
spotpython tuning: 3127.83349609375 [#---------] 6.71%

train_model result: {'val_loss': 3726.600830078125, 'hp_metric': 3726.600830078125}
spotpython tuning: 3127.83349609375 [#---------] 8.85%

train_model result: {'val_loss': 3270.455078125, 'hp_metric': 3270.455078125}
spotpython tuning: 3127.83349609375 [#---------] 10.80%

train_model result: {'val_loss': 3280.859375, 'hp_metric': 3280.859375}
spotpython tuning: 3127.83349609375 [#---------] 12.43%

train_model result: {'val_loss': 3094.462646484375, 'hp_metric': 3094.462646484375}
spotpython tuning: 3094.462646484375 [#---------] 14.98%

train_model result: {'val_loss': 16721.701171875, 'hp_metric': 16721.701171875}
spotpython tuning: 3094.462646484375 [##--------] 16.69%

train_model result: {'val_loss': 3214.978515625, 'hp_metric': 3214.978515625}
spotpython tuning: 3094.462646484375 [##--------] 18.69%

train_model result: {'val_loss': 17011.591796875, 'hp_metric': 17011.591796875}
spotpython tuning: 3094.462646484375 [##--------] 20.61%

train_model result: {'val_loss': 3064.27783203125, 'hp_metric': 3064.27783203125}
spotpython tuning: 3064.27783203125 [##--------] 22.99%

train_model result: {'val_loss': 3280.272705078125, 'hp_metric': 3280.272705078125}
spotpython tuning: 3064.27783203125 [###-------] 26.46%

train_model result: {'val_loss': 3795.2099609375, 'hp_metric': 3795.2099609375}
spotpython tuning: 3064.27783203125 [###-------] 31.74%

train_model result: {'val_loss': 3192.682861328125, 'hp_metric': 3192.682861328125}
spotpython tuning: 3064.27783203125 [###-------] 34.73%
```

```
train_model result: {'val_loss': 4895.74658203125, 'hp_metric': 4895.74658203125}
spotpython tuning: 3064.27783203125 [####------] 37.03%

train_model result: {'val_loss': 3190.875, 'hp_metric': 3190.875}
spotpython tuning: 3064.27783203125 [####------] 40.58%

train_model result: {'val_loss': 3173.78076171875, 'hp_metric': 3173.78076171875}
spotpython tuning: 3064.27783203125 [####------] 44.03%

train_model result: {'val_loss': 3260.626708984375, 'hp_metric': 3260.626708984375}
spotpython tuning: 3064.27783203125 [#####-----] 47.51%

train_model result: {'val_loss': 3129.239990234375, 'hp_metric': 3129.239990234375}
spotpython tuning: 3064.27783203125 [#####-----] 50.62%

train_model result: {'val_loss': 3193.769775390625, 'hp_metric': 3193.769775390625}
spotpython tuning: 3064.27783203125 [#####-----] 53.86%

train_model result: {'val_loss': 2997.129638671875, 'hp_metric': 2997.129638671875}
spotpython tuning: 2997.129638671875 [######----] 57.47%

train_model result: {'val_loss': 5633.95263671875, 'hp_metric': 5633.95263671875}
spotpython tuning: 2997.129638671875 [######----] 60.19%

train_model result: {'val_loss': 3285.469970703125, 'hp_metric': 3285.469970703125}
spotpython tuning: 2997.129638671875 [######----] 63.02%

train_model result: {'val_loss': 23978.36328125, 'hp_metric': 23978.36328125}
spotpython tuning: 2997.129638671875 [#######---] 67.62%

train_model result: {'val_loss': 3132.0087890625, 'hp_metric': 3132.0087890625}
spotpython tuning: 2997.129638671875 [#######---] 70.46%

train_model result: {'val_loss': 3271.72705078125, 'hp_metric': 3271.72705078125}
spotpython tuning: 2997.129638671875 [#######---] 72.96%

train_model result: {'val_loss': 3231.15576171875, 'hp_metric': 3231.15576171875}
spotpython tuning: 2997.129638671875 [########--] 75.66%

train_model result: {'val_loss': 3098.72216796875, 'hp_metric': 3098.72216796875}
spotpython tuning: 2997.129638671875 [########--] 78.40%
```

```
train_model result: {'val_loss': 3405.714599609375, 'hp_metric': 3405.714599609375}
spotpython tuning: 2997.129638671875 [########--] 81.23%

train_model result: {'val_loss': 3313.08544921875, 'hp_metric': 3313.08544921875}
spotpython tuning: 2997.129638671875 [########--] 83.56%

train_model result: {'val_loss': 3084.812255859375, 'hp_metric': 3084.812255859375}
spotpython tuning: 2997.129638671875 [#########-] 86.03%

train_model result: {'val_loss': 3133.7744140625, 'hp_metric': 3133.7744140625}
spotpython tuning: 2997.129638671875 [#########-] 89.18%

train_model result: {'val_loss': 3248.294921875, 'hp_metric': 3248.294921875}
spotpython tuning: 2997.129638671875 [#########-] 92.26%

train_model result: {'val_loss': 3426.903076171875, 'hp_metric': 3426.903076171875}
spotpython tuning: 2997.129638671875 [#########-] 94.63%

train_model result: {'val_loss': 3099.663818359375, 'hp_metric': 3099.663818359375}
spotpython tuning: 2997.129638671875 [##########] 98.74%

train_model result: {'val_loss': 3651.17626953125, 'hp_metric': 3651.17626953125}
spotpython tuning: 2997.129638671875 [##########] 100.00% Done...

Experiment saved to 601_res.pkl


<spotpython.spot.spot.Spot at 0x1530c6810>
```

32.2. Looking at the Results

32.2.1. Tuning Progress

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with `spotpython`'s method `plot_progress`. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
S.plot_progress()
```

32.2.2. Tuned Hyperparameters and Their Importance

Results can be printed in tabular form.

```
print_res_table(S)
```

```
| name           | type   | default   |   lower |   upper | tuned               | transform
|----------------|--------|-----------|---------|---------|---------------------|---------------------
| l1             | int    | 3         |     3.0 |     4.0 | 3.0                 | transform_power_2_i
| epochs         | int    | 4         |     3.0 |     7.0 | 4.0                 | transform_power_2_i
| batch_size     | int    | 4         |     4.0 |    11.0 | 4.0                 | transform_power_2_i
| act_fn         | factor | ReLU      |     0.0 |     5.0 | ReLU                | None
| optimizer      | factor | SGD       |     0.0 |     2.0 | Adadelta            | None
| dropout_prob   | float  | 0.01      |     0.0 |   0.025 | 0.025               | None
| lr_mult        | float  | 1.0       |     0.1 |    10.0 | 0.6001693730742488  | None
| patience       | int    | 2         |     2.0 |     3.0 | 3.0                 | transform_power_2_i
| batch_norm     | factor | 0         |     0.0 |     1.0 | 0                   | None
| initialization | factor | Default   |     0.0 |     4.0 | kaiming_normal      | None
```

A histogram can be used to visualize the most important hyperparameters.

```
S.plot_importance(threshold=1.0)
```

```
S.plot_important_hyperparameter_contour(max_imp=3)
```

```
l1:  0.001
epochs:  0.1630288878588729
batch_size:  0.03917689166680782
act_fn:  5.002366265225456
optimizer:  100.0
dropout_prob:  100.0
lr_mult:  0.006697298536111381
patience:  27.017413811972087
batch_norm:  15.618570394784767
initialization:  0.001
```

32.2.3. Get the Tuned Architecture

```
import pprint
from spotpython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(S)
pprint.pprint(config)
```

```
{'act_fn': ReLU(),
 'batch_norm': False,
 'batch_size': 16,
 'dropout_prob': 0.025,
 'epochs': 16,
 'initialization': 'kaiming_normal',
 'l1': 8,
 'lr_mult': 0.6001693730742488,
 'optimizer': 'Adadelta',
 'patience': 8}
```

32.2.4. Test on the full data set

```
# set the value of the key "TENSORBOARD_CLEAN" to True in the fun_control dictionary a
fun_control.update({"TENSORBOARD_CLEAN": True})
fun_control.update({"tensorboard_log": True})
```

```
from spotpython.light.testmodel import test_model
from spotpython.utils.init import get_feature_names

test_model(config, fun_control)
get_feature_names(fun_control)
```

```
        Test metric               DataLoader 0

         hp_metric              3280.158935546875
         val_loss               3280.158935546875
```

```
test_model result: {'val_loss': 3280.158935546875, 'hp_metric': 3280.158935546875}
```

```
['age',
 'sex',
 'bmi',
 'bp',
 's1_tc',
 's2_ldl',
 's3_hdl',
 's4_tch',
 's5_ltg',
 's6_glu']
```

32.3. Cross Validation With Lightning

- The `KFold` class from `sklearn.model_selection` is used to generate the folds for cross-validation.
- These mechanism is used to generate the folds for the final evaluation of the model.
- The `CrossValidationDataModule` class [SOURCE] is used to generate the folds for the hyperparameter tuning process.

- It is called from the `cv_model` function [SOURCE].

```
config
```

```
{'l1': 8,
 'epochs': 16,
 'batch_size': 16,
 'act_fn': ReLU(),
 'optimizer': 'Adadelta',
 'dropout_prob': 0.025,
 'lr_mult': 0.6001693730742488,
 'patience': 8,
 'batch_norm': False,
 'initialization': 'kaiming_normal'}
```

```
from spotpython.light.cvmodel import cv_model
fun_control.update({"k_folds": 2})
fun_control.update({"test_size": 0.6})
cv_model(config, fun_control)
```

```
k: 0
```

```
train_model result: {'val_loss': 3209.988525390625, 'hp_metric': 3209.988525390625}
k: 1
train_model result: {'val_loss': 3011.973876953125, 'hp_metric': 3011.973876953125}
```

```
3110.981201171875
```

32.4. Extending the Basic Setup

This basic setup can be adapted to user-specific needs in many ways. For example, the user can specify a custom data set, a custom model, or a custom loss function. The following sections provide more details on how to customize the hyperparameter tuning process. Before we proceed, we will provide an overview of the basic settings of the hyperparameter tuning process and explain the parameters used so far.

32.4.1. General Experiment Setup

To keep track of the different experiments, we use a PREFIX for the experiment name. The PREFIX is used to create a unique experiment name. The PREFIX is also used to create a unique TensorBoard folder, which is used to store the TensorBoard log files.

spotpython allows the specification of two different types of stopping criteria: first, the number of function evaluations (fun_evals), and second, the maximum run time in seconds (max_time). Here, we will set the number of function evaluations to infinity and the maximum run time to one minute.

max_time is set to one minute for demonstration purposes. For real experiments, this value should be increased. Note, the total run time may exceed the specified max_time, because the initial design is always evaluated, even if this takes longer than max_time.

32.4.2. Data Setup

Here, we have provided the Diabetes data set class, which is a subclass of torch.utils.data.Dataset. Data preprocessing is handled by Lightning and PyTorch. It is described in the LIGHTNINGDATAMODULE documentation.

The data splitting, i.e., the generation of training, validation, and testing data, is handled by Lightning.

32.4.3. Objective Function fun

The objective function fun from the class HyperLight [SOURCE] is selected next. It implements an interface from PyTorch's training, validation, and testing methods to spotpython.

32.4.4. Core-Model Setup

By using core_model_name = "light.regression.NNLinearRegressor", the spotpython model class NetLightRegression [SOURCE] from the light.regression module is selected.

32.4.5. Hyperdict Setup

For a given `core_model_name`, the corresponding hyperparameters are automatically loaded from the associated dictionary, which is stored as a JSON file. The JSON file contains hyperparameter type information, names, and bounds. For `spotpython` models, the hyperparameters are stored in the `LightHyperDict`, see [SOURCE] Alternatively, you can load a local hyper_dict. The `hyperdict` uses the default hyperparameter settings. These can be modified as described in Section D.15.1.

32.4.6. Other Settings

There are several additional parameters that can be specified, e.g., since we did not specify a loss function, `mean_squared_error` is used, which is the default loss function. These will be explained in more detail in the following sections.

32.5. Tensorboard

The textual output shown in the console (or code cell) can be visualized with Tensorboard, if the argument `tensorboard_log` to `fun_control_init()` is set to `True`. The Tensorboard log files are stored in the `runs` folder. To start Tensorboard, run the following command in the terminal:

```
tensorboard --logdir="runs/"
```

Further information can be found in the PyTorch Lightning documentation for Tensorboard.

32.6. Loading the Saved Experiment and Getting the Hyperparameters of the Tuned Model

To get the tuned hyperparameters as a dictionary, the `get_tuned_architecture` function can be used.

```
from spotpython.utils.file import load_result
spot_tuner = load_result(PREFIX=PREFIX)
config = get_tuned_architecture(spot_tuner)
config
```

```
Loaded experiment from 601_res.pkl
```

```
{'l1': 8,
 'epochs': 16,
 'batch_size': 16,
 'act_fn': ReLU(),
 'optimizer': 'Adadelta',
 'dropout_prob': 0.025,
 'lr_mult': 0.6001693730742488,
 'patience': 8,
 'batch_norm': False,
 'initialization': 'kaiming_normal'}
```

32.7. Using the spotgui

The spotgui [github] provides a convenient way to interact with the hyperparameter tuning process. To obtain the settings from Section 32.1, the spotgui can be started as shown in Figure 32.1.

Figure 32.1.: spotgui

32.8. Summary

This section presented an introduction to the basic setup of hyperparameter tuning with `spotpython` and `PyTorch` Lightning.

33. Hyperparameter Tuning with PyTorch Lightning and User Data Sets

In this section, we will show how user specfied data can be used for the `PyTorch` Lightning hyperparameter tuning workflow with `spotpython`.

33.1. Loading a User Specified Data Set

Using a user-specified data set is straightforward.

The user simply needs to provide a data set and loads is as a `spotpython` `CVSDataset()` class by specifying the path, filename, and target column.

Consider the following example, where the user has a data set stored in the `userData` directory. The data set is stored in a file named `data.csv`. The target column is named `target`. To show the data, it is loaded as a `pandas` data frame and the first 5 rows are displayed. This step is not necessary for the hyperparameter tuning process, but it is useful for understanding the data.

```
# load the csv data set as a pandas dataframe and dislay the first 5 rows
import pandas as pd
data = pd.read_csv("./userData/data.csv")
print(data.head())
```

```
        age       sex       bmi        bp        s1        s2        s3  \
0  0.038076  0.050680  0.061696  0.021872 -0.044223 -0.034821 -0.043401
1 -0.001882 -0.044642 -0.051474 -0.026328 -0.008449 -0.019163  0.074412
2  0.085299  0.050680  0.044451 -0.005670 -0.045599 -0.034194 -0.032356
3 -0.089063 -0.044642 -0.011595 -0.036656  0.012191  0.024991 -0.036038
4  0.005383 -0.044642 -0.036385  0.021872  0.003935  0.015596  0.008142

         s4        s5        s6  target
0 -0.002592  0.019907 -0.017646   151.0
1 -0.039493 -0.068332 -0.092204    75.0
```

```
2 -0.002592  0.002861 -0.025930   141.0
3  0.034309  0.022688 -0.009362   206.0
4 -0.002592 -0.031988 -0.046641   135.0
```

Next, the data set is loaded as a `spotpython` `CSVDataset()` class. This step is necessary for the hyperparameter tuning process.

```
from spotpython.data.csvdataset import CSVDataset
import torch
data_set = CSVDataset(directory="./userData/",
                      filename="data.csv",
                      target_column="target",
                      feature_type=torch.float32,
                      target_type=torch.float32,
                      rmNA=True)
print(len(data_set))
```

```
442
```

The following step is not necessary for the hyperparameter tuning process, but it is useful for understanding the data. The data set is loaded as a `DataLoader` from `torch.utils.data` to check the data.

```
# Set batch size for DataLoader
batch_size = 5
# Create DataLoader
from torch.utils.data import DataLoader
dataloader = DataLoader(data_set, batch_size=batch_size, shuffle=False)

# Iterate over the data in the DataLoader
for batch in dataloader:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
```

```
---------------
Inputs: tensor([[ 0.0381,  0.0507,  0.0617,  0.0219, -0.0442, -0.0348, -0.0434, -0.0026,
          0.0199, -0.0176],
        [-0.0019, -0.0446, -0.0515, -0.0263, -0.0084, -0.0192,  0.0744, -0.0395,
         -0.0683, -0.0922],
        [ 0.0853,  0.0507,  0.0445, -0.0057, -0.0456, -0.0342, -0.0324, -0.0026,
          0.0029, -0.0259],
        [-0.0891, -0.0446, -0.0116, -0.0367,  0.0122,  0.0250, -0.0360,  0.0343,
          0.0227, -0.0094],
        [ 0.0054, -0.0446, -0.0364,  0.0219,  0.0039,  0.0156,  0.0081, -0.0026,
         -0.0320, -0.0466]])
Targets: tensor([151.,  75., 141., 206., 135.])
```

Similar to the setting from Section 32.1, the hyperparameter tuning setup is defined. Instead of using the `Diabetes` data set, the user data set is used. The `data_set` parameter is set to the user data set. The `fun_control` dictionary is set up via the `fun_control_init` function.

Note, that we have modified the `fun_evals` parameter to 12 and the `init_size` to 7 to reduce the computational time for this example.

```python
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_control_init)
from spotpython.utils.eda import print_res_table
from spotpython.hyperparameters.values import set_hyperparameter
from spotpython.spot import Spot

fun_control = fun_control_init(
    PREFIX="601",
    fun_evals=12,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1)

design_control = design_control_init(init_size=7)

set_hyperparameter(fun_control, "initialization", ["Default"])

fun = HyperLight().fun

spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
```

```
module_name: light
submodule_name: regression
model_name: NNLinearRegressor
```

```
res = spot_tuner.run()
print_res_table(spot_tuner)
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
train_model(): trainer.fit failed with exception: SparseAdam does not support dense gr
train_model result: {'val_loss': 23320.16015625, 'hp_metric': 23320.16015625}

train_model result: {'val_loss': nan, 'hp_metric': nan}

train_model result: {'val_loss': 4052.16845703125, 'hp_metric': 4052.16845703125}

train_model result: {'val_loss': 3584.89013671875, 'hp_metric': 3584.89013671875}
train_model result: {'val_loss': nan, 'hp_metric': nan}

train_model result: {'val_loss': nan, 'hp_metric': nan}

train_model result: {'val_loss': 18160.900390625, 'hp_metric': 18160.900390625}
train_model result: {'val_loss': 3988.379638671875, 'hp_metric': 3988.379638671875}
spotpython tuning: 3584.89013671875 [####------] 41.67%

train_model result: {'val_loss': 3752.748779296875, 'hp_metric': 3752.748779296875}
spotpython tuning: 3584.89013671875 [#####-----] 50.00%

train_model result: {'val_loss': 3618.64404296875, 'hp_metric': 3618.64404296875}
spotpython tuning: 3584.89013671875 [######----] 58.33%

train_model result: {'val_loss': 3731.224365234375, 'hp_metric': 3731.224365234375}
spotpython tuning: 3584.89013671875 [#######---] 66.67%

train_model result: {'val_loss': 3642.53173828125, 'hp_metric': 3642.53173828125}
spotpython tuning: 3584.89013671875 [########--] 75.00%

train_model result: {'val_loss': 3767.342529296875, 'hp_metric': 3767.342529296875}
spotpython tuning: 3584.89013671875 [########--] 83.33%

train_model result: {'val_loss': 3528.6337890625, 'hp_metric': 3528.6337890625}
spotpython tuning: 3528.6337890625 [#########-] 91.67%
```

```
train_model result: {'val_loss': 12298.3740234375, 'hp_metric': 12298.3740234375}
spotpython tuning: 3528.6337890625 [##########] 100.00% Done...

Experiment saved to 601_res.pkl
| name           | type   | default    |   lower |   upper | tuned              | transform
|----------------|--------|------------|---------|---------|--------------------|---------------------
| l1             | int    | 3          |     3.0 |     8.0 | 6.0                | transform_power_2_in
| epochs         | int    | 4          |     4.0 |     9.0 | 9.0                | transform_power_2_in
| batch_size     | int    | 4          |     1.0 |     4.0 | 3.0                | transform_power_2_in
| act_fn         | factor | ReLU       |     0.0 |     5.0 | Sigmoid            | None
| optimizer      | factor | SGD        |     0.0 |    11.0 | NAdam              | None
| dropout_prob   | float  | 0.01       |     0.0 |    0.25 | 0.0                | None
| lr_mult        | float  | 1.0        |     0.1 |    10.0 | 6.203235170441004  | None
| patience       | int    | 2          |     2.0 |     6.0 | 3.0                | transform_power_2_in
| batch_norm     | factor | 0          |     0.0 |     1.0 | 1                  | None
| initialization | factor | Default    |     0.0 |     0.0 | Default            | None
l1:  8.57605796959573
epochs:  0.004677170523997896
batch_size:  0.12125843422237481
act_fn:  0.004677170523997896
optimizer:  92.45415473418441
dropout_prob:  0.004677170523997896
lr_mult:  0.08544388767815575
patience:  100.0
batch_norm:  68.80462735958255
```

33.2. Summary

This section showed how to use user-specified data sets for the hyperparameter tuning process with `spotpython`. The user needs to provide the data set and load it as a `spotpython CSVDataset()` class.

34. Hyperparameter Tuning with PyTorch Lightning and User Models

In this section, we will show how a user defined model can be used for the `PyTorch` Lightning hyperparameter tuning workflow with `spotpython`.

34.1. Using a User Specified Model

As templates, we provide the following three files that allow the user to specify a model in the `/userModel` directory:

- `my_regressor.py`, see Section 34.2.4
- `my_hyperdict.json`, see Section 34.2.3
- `my_hyperdict.py`, see Section 34.2.2.

The `my_regressor.py` file contains the model class, which is a subclass of `nn.Module`. The `my_hyperdict.json` file contains the hyperparameter settings as a dictionary, which are loaded via the `my_hyperdict.py` file.

Note, that we have to add the path to the `userModel` directory to the `sys.path` list as shown below.

```
import sys
sys.path.insert(0, './userModel')
import my_regressor
import my_hyper_dict
from spotpython.hyperparameters.values import add_core_model_to_fun_control

from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, design_control_init)
from spotpython.utils.eda import print_res_table
from spotpython.hyperparameters.values import set_hyperparameter
from spotpython.spot import Spot
```

```
fun_control = fun_control_init(
    PREFIX="601-user-model",
    fun_evals=inf,
    max_time=1,
    data_set = Diabetes(),
    _L_in=10,
    _L_out=1)

add_core_model_to_fun_control(fun_control=fun_control,
                              core_model=my_regressor.MyRegressor,
                              hyper_dict=my_hyper_dict.MyHyperDict)

design_control = design_control_init(init_size=7)

fun = HyperLight().fun

spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
```

```
res = spot_tuner.run()
print_res_table(spot_tuner)
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
train_model(): trainer.fit failed with exception: SparseAdam does not support dense gr
train_model result: {'val_loss': 23988.361328125, 'hp_metric': 23988.361328125}

train_model result: {'val_loss': nan, 'hp_metric': nan}

train_model result: {'val_loss': 3127.356689453125, 'hp_metric': 3127.356689453125}

train_model result: {'val_loss': 4815.912109375, 'hp_metric': 4815.912109375}
train_model result: {'val_loss': nan, 'hp_metric': nan}

train_model result: {'val_loss': 2994.751953125, 'hp_metric': 2994.751953125}

train_model result: {'val_loss': 2893.5380859375, 'hp_metric': 2893.5380859375}

train_model result: {'val_loss': 4207.3974609375, 'hp_metric': 4207.3974609375}
spotpython tuning: 2893.5380859375 [----------] 4.43%

train_model result: {'val_loss': 3276.323974609375, 'hp_metric': 3276.323974609375}
spotpython tuning: 2893.5380859375 [#---------] 6.15%
```

```
train_model result: {'val_loss': 3185.3427734375, 'hp_metric': 3185.3427734375}
spotpython tuning: 2893.5380859375 [#---------] 8.50%

train_model result: {'val_loss': 3342.8623046875, 'hp_metric': 3342.8623046875}
spotpython tuning: 2893.5380859375 [#---------] 11.88%

train_model result: {'val_loss': 3275.938720703125, 'hp_metric': 3275.938720703125}
spotpython tuning: 2893.5380859375 [#---------] 14.34%

train_model result: {'val_loss': 3104.694091796875, 'hp_metric': 3104.694091796875}
spotpython tuning: 2893.5380859375 [###-------] 31.43%

train_model result: {'val_loss': 3232.864501953125, 'hp_metric': 3232.864501953125}
spotpython tuning: 2893.5380859375 [###-------] 33.87%

train_model result: {'val_loss': 3393.63671875, 'hp_metric': 3393.63671875}
spotpython tuning: 2893.5380859375 [####------] 36.10%

train_model result: {'val_loss': 3623.41015625, 'hp_metric': 3623.41015625}
spotpython tuning: 2893.5380859375 [####------] 44.35%

train_model result: {'val_loss': 3068.196044921875, 'hp_metric': 3068.196044921875}
spotpython tuning: 2893.5380859375 [######----] 64.41%

train_model result: {'val_loss': 3293.473876953125, 'hp_metric': 3293.473876953125}
spotpython tuning: 2893.5380859375 [#######---] 66.52%

train_model result: {'val_loss': 2899.44775390625, 'hp_metric': 2899.44775390625}
spotpython tuning: 2893.5380859375 [#######---] 71.59%

train_model result: {'val_loss': 2898.648681640625, 'hp_metric': 2898.648681640625}
spotpython tuning: 2893.5380859375 [########--] 77.56%

train_model result: {'val_loss': 3078.852294921875, 'hp_metric': 3078.852294921875}
spotpython tuning: 2893.5380859375 [########--] 83.74%

train_model result: {'val_loss': 2886.664306640625, 'hp_metric': 2886.664306640625}
spotpython tuning: 2886.664306640625 [#########-] 86.54%

train_model result: {'val_loss': 3959.025146484375, 'hp_metric': 3959.025146484375}
spotpython tuning: 2886.664306640625 [##########] 99.40%
```

```
train_model result: {'val_loss': 3046.339111328125, 'hp_metric': 3046.339111328125}
spotpython tuning: 2886.664306640625 [##########] 100.00% Done...

Experiment saved to 601-user-model_res.pkl
| name           | type   | default   |    lower |    upper | tuned                | tra
|----------------|--------|-----------|----------|----------|----------------------|----
| l1             | int    | 3         |      3.0 |      8.0 | 4.0                  | tra
| epochs         | int    | 4         |      4.0 |      9.0 | 6.0                  | tra
| batch_size     | int    | 4         |      1.0 |      4.0 | 4.0                  | tra
| act_fn         | factor | ReLU      |      0.0 |      5.0 | ReLU                 | None
| optimizer      | factor | SGD       |      0.0 |     11.0 | Adagrad              | None
| dropout_prob   | float  | 0.01      |      0.0 |     0.25 | 0.11448887310206422  | None
| lr_mult        | float  | 1.0       |      0.1 |     10.0 | 10.0                 | None
| patience       | int    | 2         |      2.0 |      6.0 | 3.0                  | tra
| initialization | factor | Default   |      0.0 |      4.0 | xavier_uniform       | None
l1:  0.9986910151874973
epochs:  1.2059238997636084
batch_size:  1.2516780242717707
act_fn:  1.0989614191000827
optimizer:  1.0537139135933933
dropout_prob:  1.0726143932055738
lr_mult:  62.51897882171729
patience:  100.0
initialization:  0.9986910198782283
```

34.2. Details

34.2.1. Model Setup

By using `core_model_name = "my_regressor.MyRegressor"`, the user specified model class `MyRegressor` [SOURCE] is selected. For this given `core_model_name`, the local hyper_dict is loaded using the `my_hyper_dict.py` file as shown below.

34.2.2. The `my_hyper_dict.py` File

The `my_hyper_dict.py` file must be placed in the `/userModel` directory. It provides a convenience function to load the hyperparameters from user specified the `my_hyper_dict.json` file, see Section 34.2.2. The user does not need to modify this file, if the JSON file is stored as `my_hyper_dict.json`. Alternative filenames can be specified via the `filename` argument (which is default set to `"my_hyper_dict.json"`).

34.2.3. The `my_hyper_dict.json` File

The `my_hyper_dict.json` file contains the hyperparameter settings as a dictionary, which are loaded via the `my_hyper_dict.py` file. The example below shows the content of the `my_hyper_dict.json` file.

```
{
    "MyRegressor": {
        "l1": {
            "type": "int",
            "default": 3,
            "transform": "transform_power_2_int",
            "lower": 3,
            "upper": 8
```

```
        },
        "epochs": {
            "type": "int",
            "default": 4,
            "transform": "transform_power_2_int",
            "lower": 4,
            "upper": 9
        },
        "batch_size": {
            "type": "int",
            "default": 4,
            "transform": "transform_power_2_int",
            "lower": 1,
            "upper": 4
        },
        "act_fn": {
            "levels": [
                "Sigmoid",
                "Tanh",
                "ReLU",
                "LeakyReLU",
                "ELU",
                "Swish"
            ],
            "type": "factor",
            "default": "ReLU",
            "transform": "None",
            "class_name": "spotpython.torch.activation",
            "core_model_parameter_type": "instance()",
            "lower": 0,
            "upper": 5
        },
        "optimizer": {
            "levels": [
                "Adadelta",
                "Adagrad",
                "Adam",
                "AdamW",
                "SparseAdam",
                "Adamax",
                "ASGD",
                "NAdam",
                "RAdam",
                "RMSprop",
```

```
                "Rprop",
                "SGD"
            ],
            "type": "factor",
            "default": "SGD",
            "transform": "None",
            "class_name": "torch.optim",
            "core_model_parameter_type": "str",
            "lower": 0,
            "upper": 11
        },
        "dropout_prob": {
            "type": "float",
            "default": 0.01,
            "transform": "None",
            "lower": 0.0,
            "upper": 0.25
        },
        "lr_mult": {
            "type": "float",
            "default": 1.0,
            "transform": "None",
            "lower": 0.1,
            "upper": 10.0
        },
        "patience": {
            "type": "int",
            "default": 2,
            "transform": "transform_power_2_int",
            "lower": 2,
            "upper": 6
        },
        "initialization": {
            "levels": [
                "Default",
                "Kaiming",
                "Xavier"
            ],
            "type": "factor",
            "default": "Default",
            "transform": "None",
            "core_model_parameter_type": "str",
            "lower": 0,
            "upper": 2
```

```
        }
    }
}
```

34.2.4. The `my_regressor.py` File

The `my_regressor.py` file contains [SOURCE] the model class, which is a subclass of `nn.Module`. It must implement the following methods:

- `__init__(self, **kwargs)`: The constructor of the model class. The hyperparameters are passed as keyword arguments.
- `forward(self, x: torch.Tensor) -> torch.Tensor`: The forward pass of the model. The input `x` is passed through the model and the output is returned.
- `training_step(self, batch, batch_idx) -> torch.Tensor`: The training step of the model. It takes a batch of data and the batch index as input and returns the loss.
- `validation_step(self, batch, batch_idx) -> torch.Tensor`: The validation step of the model. It takes a batch of data and the batch index as input and returns the loss.
- `test_step(self, batch, batch_idx) -> torch.Tensor`: The test step of the model. It takes a batch of data and the batch index as input and returns the loss.
- `predict(self, x: torch.Tensor) -> torch.Tensor`: The prediction method of the model. It takes an input `x` and returns the prediction.
- `configure_optimizers(self) -> torch.optim.Optimizer`: The method to configure the optimizer of the model. It returns the optimizer.

The file `my_regressor.py` must be placed in the `/userModel` directory. The user can modify the model class to implement a custom model architecture.

We will take a closer look at the methods defined in the `my_regressor.py` file in the next subsections.

34.2.4.1. The `__init__` Method

`__init__()` initializes the `MyRegressor` object. It takes the following arguments:

- `l1` (int): The number of neurons in the first hidden layer.
- `epochs` (int): The number of epochs to train the model for.
- `batch_size` (int): The batch size to use during training.
- `initialization` (str): The initialization method to use for the weights.
- `act_fn` (nn.Module): The activation function to use in the hidden layers.
- `optimizer` (str): The optimizer to use during training.
- `dropout_prob` (float): The probability of dropping out a neuron during training.

- `lr_mult` (float): The learning rate multiplier for the optimizer.
- `patience` (int): The number of epochs to wait before early stopping.
- `_L_in` (int): The number of input features. Not a hyperparameter, but needed to create the network.
- `_L_out` (int): The number of output classes. Not a hyperparameter, but needed to create the network.
- `_torchmetric` (str): The metric to use for the loss function. If `None`, then "mean_squared_error" is used.

It is implemented as follows:

```
class MyRegressor(L.LightningModule):
        def __init__(
        self,
        l1: int,
        epochs: int,
        batch_size: int,
        initialization: str,
        act_fn: nn.Module,
        optimizer: str,
        dropout_prob: float,
        lr_mult: float,
        patience: int,
        _L_in: int,
        _L_out: int,
        _torchmetric: str,
        *args,
        **kwargs,
    ):
        super().__init__()
        self._L_in = _L_in
        self._L_out = _L_out
        if _torchmetric is None:
            _torchmetric = "mean_squared_error"
        self._torchmetric = _torchmetric
        self.metric = getattr(torchmetrics.functional.regression, _torchmetric)
        # _L_in and _L_out are not hyperparameters, but are needed to create the network
        # _torchmetric is not a hyperparameter, but is needed to calculate the loss
        self.save_hyperparameters(ignore=["_L_in", "_L_out", "_torchmetric"])
        # set dummy input array for Tensorboard Graphs
        # set log_graph=True in Trainer to see the graph (in traintest.py)
        self.example_input_array = torch.zeros((batch_size, self._L_in))
        if self.hparams.l1 < 4:
            raise ValueError("l1 must be at least 4")
        hidden_sizes = [l1 * 2, l1, ceil(l1/2)]
```

```
        # Create the network based on the specified hidden sizes
        layers = []
        layer_sizes = [self._L_in] + hidden_sizes
        layer_size_last = layer_sizes[0]
        for layer_size in layer_sizes[1:]:
            layers += [
                nn.Linear(layer_size_last, layer_size),
                self.hparams.act_fn,
                nn.Dropout(self.hparams.dropout_prob),
            ]
            layer_size_last = layer_size
        layers += [nn.Linear(layer_sizes[-1], self._L_out)]
        # nn.Sequential summarizes a list of modules into a single module,
        # applying them in sequence
        self.layers = nn.Sequential(*layers)
```

34.2.4.2. The `hidden_sizes`

`__init__()` uses the `hidden_sizes` list to define the sizes of the hidden layers in the network. The hidden sizes are calculated based on the `l1` hyperparameter. The hidden sizes can be computed in different ways, depending on the problem and the desired network architecture. We recommend using a separate function to calculate the hidden sizes based on the hyperparameters.

34.2.4.3. The `forward` Method

The `forward()` method defines the forward pass of the model. It takes an input tensor `x` and passes it through the network layers to produce an output tensor. It is implemented as follows:

```
def forward(self, x: torch.Tensor) -> torch.Tensor:
    return self.layers(x)
```

34.2.4.4. The `_calculate_loss` Method

The `_calculate_loss()` method calculates the loss based on the predicted output and the target values. It uses the specified metric to calculate the loss. It takes the following arguments:

- `batch (tuple)`: A tuple containing a batch of input data and labels.

It is implemented as follows:

```
def _calculate_loss(self, batch):
    x, y = batch
    y = y.view(len(y), 1)
    y_hat = self(x)
    loss = self.metric(y_hat, y)
    return loss
```

34.2.4.5. The `training_step` Method

The `training_step()` method defines the training step of the model. It takes a batch of data and returns the loss. It is implemented as follows:

```
def training_step(self, batch: tuple) -> torch.Tensor:
    val_loss = self._calculate_loss(batch)
    return val_loss
```

34.2.4.6. The `validation_step` Method

The `validation_step()` method defines the validation step of the model. It takes a batch of data and returns the loss. It is implemented as follows:

```
def validation_step(self, batch: tuple) -> torch.Tensor:
    val_loss = self._calculate_loss(batch)
    return val_loss
```

34.2.4.7. The `test_step` Method

The `test_step()` method defines the test step of the model. It takes a batch of data and returns the loss. It is implemented as follows:

```
def test_step(self, batch: tuple) -> torch.Tensor:
    val_loss = self._calculate_loss(batch)
    return val_loss
```

34.2.4.8. The `predict` Method

The `predict()` method defines the prediction method of the model. It takes an input tensor `x` and returns a tuple with the input tensor `x`, the target tensor `y`, and the predicted tensor `y_hat`.

It is implemented as follows:

```
def predict(self, x: torch.Tensor) -> torch.Tensor:
    x, y = batch
    yhat = self(x)
    y = y.view(len(y), 1)
    yhat = yhat.view(len(yhat), 1)
    return (x, y, yhat)
```

34.2.4.9. The `configure_optimizers` Method

The `configure_optimizers()` method defines the optimizer to use during training. It uses the `optimizer_handler` from `spotpython.hyperparameter.optimizer` to create the optimizer based on the specified optimizer name, parameters, and learning rate multiplier. It is implemented as follows:

```
def configure_optimizers(self) -> torch.optim.Optimizer:
    optimizer = optimizer_handler(
        optimizer_name=self.hparams.optimizer, params=self.parameters(), lr_mult=self
    )
    return optimizer
```

Note, the default Lightning way is to define an optimizer as `optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)`. `spotpython` uses an optimizer handler to create the optimizer, which adapts the learning rate according to the `lr_mult` hyperparameter as well as other hyperparameters. See `spotpython.hyperparameters.optimizer.py` [SOURCE] for details.

34.3. Connection with the LightDataModule

The steps described in Section 34.2.4 are connected to the `LightDataModule` class [DOC]. This class is used to create the data loaders for the training, validation, and test sets. The `LightDataModule` class is part of the `spotpython` package and class provides the following methods [SOURCE]:

- `prepare_data()`: This method is used to prepare the data set.

- `setup()`: This method is used to create the data loaders for the training, validation, and test sets.
- `train_dataloader()`: This method is used to return the data loader for the training set.
- `val_dataloader()`: This method is used to return the data loader for the validation set.
- `test_dataloader()`: This method is used to return the data loader for the test set.
- `predict_dataloader()`: This method is used to return the data loader for the prediction set.

34.3.1. The `prepare_data()` Method

The `prepare_data()` method is used to prepare the data set. This method is called only once and on a single process. It can be used to download the data set. In our case, the data set is already available, so this method uses a simple `pass` statement.

34.3.2. The `setup()` Method

The `stage` is used to define the data set to be returned. It can be `None`, `fit`, `test`, or `predict`. If `stage` is `None`, the method returns the training (`fit`), testing (`test`), and prediction (`predict`) data sets.

The `setup` methods splits the data based on the `stage` setting for use in training, validation, and testing. It uses `torch.utils.data.random_split()` to split the data.

Splitting is based on the `test_size` and `test_seed`. The `test_size` can be a float or an int.

First, the data set sizes are determined as described in Section 34.3.2.1. Then, the data sets are split based on the `stage` setting. `spotpython`'s `LightDataModule` class uses the following sizes:

- `full_train_size`: The size of the full training data set. This data set is splitted into the final training data set and a validation data set.
- `val_size`: The size of the validation data set. The validation data set is used to validate the model during training.
- `train_size`: The size of the training data set. The training data set is used to train the model.
- `test_size`: The size of the test data set. The test data set is used to evaluate the model after training. It is not used during training ("hyperparameter tuning"). Only after everything is finished, the model is evaluated on the test data set.

34.3.2.1. Determine the Sizes of the Data Sets

```
import torch
from torch.utils.data import random_split
data_full = Diabetes()
test_size = fun_control["test_size"]
test_seed=fun_control["test_seed"]
# if test_size is float, then train_size is 1 - test_size
if isinstance(test_size, float):
    full_train_size = round(1.0 - test_size, 2)
    val_size = round(full_train_size * test_size, 2)
    train_size = round(full_train_size - val_size, 2)
else:
    # if test_size is int, then train_size is len(data_full) - test_size
    full_train_size = len(data_full) - test_size
    val_size = int(full_train_size * test_size / len(data_full))
    train_size = full_train_size - val_size

print(f"LightDataModule setup(): full_train_size: {full_train_size}")
print(f"LightDataModule setup(): val_size: {val_size}")
print(f"LightDataModule setup(): train_size: {train_size}")
print(f"LightDataModule setup(): test_size: {test_size}")
```

```
LightDataModule setup(): full_train_size: 0.6
LightDataModule setup(): val_size: 0.24
LightDataModule setup(): train_size: 0.36
LightDataModule setup(): test_size: 0.4
```

34.3.2.2. The "setup" Method: Stage "fit"

Here, `train_size` and `val_size` are used to split the data into training and validation sets.

```
stage = "fit"
scaler = None
# Assign train/val datasets for use in dataloaders
if stage == "fit" or stage is None:
    print(f"train_size: {train_size}, val_size: {val_size} used for train & val data.'
    generator_fit = torch.Generator().manual_seed(test_seed)
    data_train, data_val, _ = random_split(
        data_full, [train_size, val_size, test_size], generator=generator_fit
    )
```

```
    if scaler is not None:
        # Fit the scaler on training data and transform both train and val data
        scaler_train_data = torch.stack([data_train[i][0] for i in range(len(data_train))]).squeeze(
        # train_val_data = data_train[:,0]
        print(scaler_train_data.shape)
        scaler.fit(scaler_train_data)
        data_train = [(scaler.transform(data), target) for data, target in data_train]
        data_tensors_train = [data.clone().detach() for data, target in data_train]
        target_tensors_train = [target.clone().detach() for data, target in data_train]
        data_train = TensorDataset(
            torch.stack(data_tensors_train).squeeze(1), torch.stack(target_tensors_train)
        )
        # print(data_train)
        data_val = [(scaler.transform(data), target) for data, target in data_val]
        data_tensors_val = [data.clone().detach() for data, target in data_val]
        target_tensors_val = [target.clone().detach() for data, target in data_val]
        data_val = TensorDataset(torch.stack(data_tensors_val).squeeze(1), torch.stack(target_tensor
```

```
train_size: 0.36, val_size: 0.24 used for train & val data.
```

The `data_train` and `data_val` data sets are further used to create the training and validation data loaders as described in Section 34.3.3 and Section 34.3.4, respectively.

34.3.2.3. The "setup" Method: Stage "test"

Here, the test data set, which is based on the `test_size`, is created.

```
stage = "test"
# Assign test dataset for use in dataloader(s)
if stage == "test" or stage is None:
    print(f"test_size: {test_size} used for test dataset.")
    # get test data set as test_abs percent of the full dataset
    generator_test = torch.Generator().manual_seed(test_seed)
    data_test, _ = random_split(data_full, [test_size, full_train_size], generator=generator_test)
    if scaler is not None:
        data_test = [(scaler.transform(data), target) for data, target in data_test]
        data_tensors_test = [data.clone().detach() for data, target in data_test]
        target_tensors_test = [target.clone().detach() for data, target in data_test]
        data_test = TensorDataset(
            torch.stack(data_tensors_test).squeeze(1), torch.stack(target_tensors_test)
        )
print(f"LightDataModule setup(): Test set size: {len(data_test)}")
```

```
# Set batch size for DataLoader
batch_size = 5
# Create DataLoader
from torch.utils.data import DataLoader
dataloader = DataLoader(data_test, batch_size=batch_size, shuffle=False)
# Iterate over the data in the DataLoader
for batch in dataloader:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
test_size: 0.4 used for test dataset.
LightDataModule setup(): Test set size: 177
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0490, -0.0446, -0.0418,  0.1045,  0.0356, -0.0257,  0.1775, -0.076
         -0.0129,  0.0155],
        [-0.0273,  0.0507, -0.0159, -0.0298,  0.0039, -0.0007,  0.0413, -0.0395,
         -0.0236,  0.0113],
        [ 0.0708,  0.0507, -0.0170,  0.0219,  0.0438,  0.0563,  0.0376, -0.0026,
         -0.0702, -0.0176],
        [-0.0382,  0.0507,  0.0714, -0.0573,  0.1539,  0.1559,  0.0008,  0.0719,
          0.0503,  0.0693],
        [ 0.0453, -0.0446,  0.0391,  0.0460,  0.0067, -0.0242,  0.0081, -0.0126,
          0.0643,  0.0569]])
Targets: tensor([103.,  53.,  80., 220., 246.])
```

34.3.2.4. The "setup" Method: Stage "predict"

Prediction and testing use the same data set. The prediction data set is created based on the `test_size`.

```
stage = "predict"
if stage == "predict" or stage is None:
    print(f"test_size: {test_size} used for predict dataset.")
    # get test data set as test_abs percent of the full dataset
```

```
    generator_predict = torch.Generator().manual_seed(test_seed)
    data_predict, _ = random_split(
        data_full, [test_size, full_train_size], generator=generator_predict
    )
    if scaler is not None:
        data_predict = [(scaler.transform(data), target) for data, target in data_predict]
        data_tensors_predict = [data.clone().detach() for data, target in data_predict]
        target_tensors_predict = [target.clone().detach() for data, target in data_predict]
        data_predict = TensorDataset(
            torch.stack(data_tensors_predict).squeeze(1), torch.stack(target_tensors_predict)
        )
print(f"LightDataModule setup(): Predict set size: {len(data_predict)}")
# Set batch size for DataLoader
batch_size = 5
# Create DataLoader
from torch.utils.data import DataLoader
dataloader = DataLoader(data_predict, batch_size=batch_size, shuffle=False)
# Iterate over the data in the DataLoader
for batch in dataloader:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
test_size: 0.4 used for predict dataset.
LightDataModule setup(): Predict set size: 177
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0490, -0.0446, -0.0418,  0.1045,  0.0356, -0.0257,  0.1775, -0.0764,
         -0.0129,  0.0155],
        [-0.0273,  0.0507, -0.0159, -0.0298,  0.0039, -0.0007,  0.0413, -0.0395,
         -0.0236,  0.0113],
        [ 0.0708,  0.0507, -0.0170,  0.0219,  0.0438,  0.0563,  0.0376, -0.0026,
         -0.0702, -0.0176],
        [-0.0382,  0.0507,  0.0714, -0.0573,  0.1539,  0.1559,  0.0008,  0.0719,
          0.0503,  0.0693],
        [ 0.0453, -0.0446,  0.0391,  0.0460,  0.0067, -0.0242,  0.0081, -0.0126,
          0.0643,  0.0569]])
```

```
Targets: tensor([103.,  53.,  80., 220., 246.])
```

34.3.3. The `train_dataloader()` Method

The method '`train_dataloader` returns the training dataloader, i.e., a Pytorch DataLoader instance using the training dataset. It simply returns a DataLoader with the `data_train` set that was created in the `setup()` method as described in Section 34.3.2.2.

```
def train_dataloader(self) -> DataLoader:
    return DataLoader(data_train, batch_size=batch_size, num_workers=num_workers)
```

Using the `train_dataloader()` Method

The `train_dataloader()` method can be used as follows:

```
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
print(f"Training set size: {len(data_module.data_train)}")
dl = data_module.train_dataloader()
# Iterate over the data in the DataLoader
for batch in dl:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Training set size: 160
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[-0.0237, -0.0446, -0.0655, -0.0814, -0.0387, -0.0536,  0.0597, -0.0
         -0.0371, -0.0425],
        [ 0.0417,  0.0507, -0.0148, -0.0171, -0.0057,  0.0084, -0.0139, -0.0019,
         -0.0119,  0.0031],
```

```
        [ 0.0708,  0.0507,  0.0121,  0.0563,  0.0342,  0.0494, -0.0397,  0.0343,
          0.0274, -0.0011],
        [-0.0527,  0.0507, -0.0407, -0.0676, -0.0318, -0.0370,  0.0376, -0.0395,
         -0.0345,  0.0693],
        [ 0.0018, -0.0446, -0.0655, -0.0057, -0.0071, -0.0195,  0.0413, -0.0395,
         -0.0033,  0.0072]])
Targets: tensor([ 59.,  97., 144.,  42., 153.])
```

34.3.4. The `val_dataloader()` Method

Returns the validation dataloader, i.e., a Pytorch DataLoader instance using the validation dataset. It simply returns a DataLoader with the `data_val` set that was created in the `setup()` method as desccribed in Section 34.3.2.2.

```python
def val_dataloader(self) -> DataLoader:
    return DataLoader(data_val, batch_size=batch_size, num_workers=num_workers)
```

Using the `val_dataloader()` Method

The `val_dataloader()` method can be used as follows:

```python
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
print(f"Validation set size: {len(data_module.data_val)}")
dl = data_module.val_dataloader()
# Iterate over the data in the DataLoader
for batch in dl:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Validation set size: 106
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
```

```
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0163, -0.0446,  0.0736, -0.0412, -0.0043, -0.0135, -0.0139, -0.0
          0.0429,  0.0445],
        [ 0.0453, -0.0446,  0.0714,  0.0012, -0.0098, -0.0010,  0.0155, -0.0395,
         -0.0412, -0.0715],
        [ 0.0308,  0.0507,  0.0326,  0.0494, -0.0401, -0.0436, -0.0692,  0.0343,
          0.0630,  0.0031],
        [ 0.0235,  0.0507, -0.0396, -0.0057, -0.0484, -0.0333,  0.0118, -0.0395,
         -0.1016, -0.0674],
        [-0.0091,  0.0507,  0.0013, -0.0022,  0.0796,  0.0701,  0.0339, -0.0026,
          0.0267,  0.0818]])
Targets: tensor([275., 141., 208.,  78., 142.])
```

34.3.5. The `test_dataloader()` Method

Returns the test dataloader, i.e., a Pytorch DataLoader instance using the test dataset. It simply returns a DataLoader with the `data_test` set that was created in the `setup()` method as described in Section 31.4.2.3.

```
def test_dataloader(self) -> DataLoader:
    return DataLoader(data_test, batch_size=batch_size, num_workers=num_workers)
```

Using the `test_dataloader()` Method

The `test_dataloader()` method can be used as follows:

```python
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
print(f"Test set size: {len(data_module.data_test)}")
dl = data_module.test_dataloader()
# Iterate over the data in the DataLoader
for batch in dl:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Test set size: 177
Batch Size: 5
Inputs Shape: torch.Size([5, 10])
Targets Shape: torch.Size([5])
---------------
Inputs: tensor([[ 0.0562, -0.0446, -0.0579, -0.0080,  0.0521,  0.0491,  0.0560, -0.0214,
         -0.0283,  0.0445],
        [ 0.0018, -0.0446, -0.0709, -0.0229, -0.0016, -0.0010,  0.0266, -0.0395,
         -0.0225,  0.0072],
        [-0.0527, -0.0446,  0.0542, -0.0263, -0.0552, -0.0339, -0.0139, -0.0395,
         -0.0741, -0.0591],
        [ 0.0054, -0.0446, -0.0482, -0.0126,  0.0012, -0.0066,  0.0634, -0.0395,
         -0.0514, -0.0591],
        [-0.0527, -0.0446, -0.0094, -0.0057,  0.0397,  0.0447,  0.0266, -0.0026,
         -0.0181, -0.0135]])
Targets: tensor([158.,  49., 142.,  96.,  59.])
```

34.3.6. The `predict_dataloader()` Method

Returns the prediction dataloader, i.e., a Pytorch DataLoader instance using the prediction dataset. It simply returns a DataLoader with the `data_predict` set that was created in the `setup()` method as described in Section 31.4.2.4.

Warning

The `batch_size` is set to the length of the `data_predict` set.

```
def predict_dataloader(self) -> DataLoader:
    return DataLoader(data_predict, batch_size=len(data_predict), num_workers=num_work
```

Using the `predict_dataloader()` Method

The `predict_dataloader()` method can be used as follows:

```
from spotpython.data.lightdatamodule import LightDataModule
from spotpython.data.diabetes import Diabetes
dataset = Diabetes(target_type=torch.float)
data_module = LightDataModule(dataset=dataset, batch_size=5, test_size=0.4)
data_module.setup()
print(f"Test set size: {len(data_module.data_predict)}")
dl = data_module.predict_dataloader()
# Iterate over the data in the DataLoader
for batch in dl:
    inputs, targets = batch
    print(f"Batch Size: {inputs.size(0)}")
    print(f"Inputs Shape: {inputs.shape}")
    print(f"Targets Shape: {targets.shape}")
    print("---------------")
    print(f"Inputs: {inputs}")
    print(f"Targets: {targets}")
    break
```

```
Test set size: 177
Batch Size: 177
Inputs Shape: torch.Size([177, 10])
Targets Shape: torch.Size([177])
---------------
Inputs: tensor([[ 0.0562, -0.0446, -0.0579,  ..., -0.0214, -0.0283,  0.0445],
        [ 0.0018, -0.0446, -0.0709,  ..., -0.0395, -0.0225,  0.0072],
        [-0.0527, -0.0446,  0.0542,  ..., -0.0395, -0.0741, -0.0591],
        ...,
        [ 0.0090, -0.0446, -0.0321,  ..., -0.0764, -0.0119, -0.0384],
        [-0.0273, -0.0446, -0.0666,  ..., -0.0395, -0.0358, -0.0094],
        [ 0.0817,  0.0507,  0.0067,  ...,  0.0919,  0.0547,  0.0072]])
Targets: tensor([158.,  49., 142.,  96.,  59.,  74., 137., 136.,  39.,  66., 310.,
        235., 116.,  55., 177.,  59., 246.,  53., 135.,  88., 198., 186., 217.,
```

```
        51., 118., 153., 180.,  51., 229.,  84.,  72., 237., 142., 185.,  91.,
        88., 148., 179., 144.,  25.,  89.,  42.,  60., 124., 170., 215., 263.,
       178., 245., 202.,  97., 321.,  71., 123., 220., 132., 243.,  61., 102.,
       187.,  70., 242., 134.,  63.,  72.,  88., 219., 127., 146., 122., 143.,
       220., 293.,  59., 317.,  60., 140.,  65., 277.,  90.,  96., 109., 190.,
        90.,  52., 160., 233., 230., 175.,  68., 272., 144.,  70.,  68., 163.,
        71.,  93., 263., 118., 220.,  90., 232., 120., 163.,  88.,  85.,  52.,
       181., 232., 212., 332.,  81., 214., 145., 268., 115.,  93.,  64., 156.,
       128., 200., 281., 103., 220.,  66.,  48., 246.,  42., 150., 125., 109.,
       129.,  97., 265.,  97., 173., 216., 237., 121.,  42., 151.,  31.,  68.,
       137., 221., 283., 124., 243., 150.,  69., 306., 182., 252., 132., 258.,
       121., 110., 292., 101., 275., 141., 208.,  78., 142., 185., 167., 258.,
       144.,  89., 225., 140., 303., 236.,  87.,  77., 131.])
```

34.4. Using the LightDataModule in the train_model() Method

The methods discussed so far are used in spotpython's train_model() method [DOC] to train the model. It is implemented as follows [SOURCE].

First, a LightDataModule object is created and the setup() method is called.

```
dm = LightDataModule(
    dataset=fun_control["data_set"],
    batch_size=config["batch_size"],
    num_workers=fun_control["num_workers"],
    test_size=fun_control["test_size"],
    test_seed=fun_control["test_seed"],
)
dm.setup()
```

Then, the Trainer is initialized.

```
# Init trainer
trainer = L.Trainer(
    default_root_dir=os.path.join(fun_control["CHECKPOINT_PATH"], config_id),
    max_epochs=model.hparams.epochs,
    accelerator=fun_control["accelerator"],
    devices=fun_control["devices"],
    logger=TensorBoardLogger(
        save_dir=fun_control["TENSORBOARD_PATH"],
```

```
        version=config_id,
        default_hp_metric=True,
        log_graph=fun_control["log_graph"],
    ),
    callbacks=[
        EarlyStopping(monitor="val_loss", patience=config["patience"], mode="min", str
    ],
    enable_progress_bar=enable_progress_bar,
)
```

Next, the `fit()` method is called to train the model.

```
# Pass the datamodule as arg to trainer.fit to override model hooks :)
trainer.fit(model=model, datamodule=dm)
```

Finally, the `validate()` method is called to validate the model. The `validate()` method returns the validation loss.

```
# Test best model on validation and test set
result = trainer.validate(model=model, datamodule=dm)
# unlist the result (from a list of one dict)
result = result[0]
return result["val_loss"]
```

34.5. The Last Connection: The `HyperLight` Class

The method `train_model()` is part of the `HyperLight` class [DOC]. It is called from `spotpython` as an objective function to train the model and return the validation loss.

The `HyperLight` class is implemented as follows [SOURCE].

```
class HyperLight:
    def fun(self, X: np.ndarray, fun_control: dict = None) -> np.ndarray:
        z_res = np.array([], dtype=float)
        self.check_X_shape(X=X, fun_control=fun_control)
        var_dict = assign_values(X, get_var_name(fun_control))
        for config in generate_one_config_from_var_dict(var_dict, fun_control):
            df_eval = train_model(config, fun_control)
            z_val = fun_control["weights"] * df_eval
            z_res = np.append(z_res, z_val)
        return z_res
```

34.6. Further Information

34.6.1. Preprocessing

Preprocessing is handled by `Lightning` and `PyTorch`. It is described in the LIGHTNINGDATAMODULE documentation. Here you can find information about the `transforms` methods.

35. Hyperparameter Tuning with PyTorch Lightning: ResNets

Neural ODEs are related to Residual Neural Networks (ResNets). We consider ResNets in Section 35.1.

35.1. Residual Neural Networks

He et al. (2015) introduced Residual Neural Networks (ResNets).

35.1.1. Residual Connections

Residual connections are a key component of ResNets. They are used to stabilize the training of very deep networks. The idea is to learn a residual mapping instead of the full mapping. The residual mapping is defined as:

Definition 35.1 (Residual Connection)**.** Let F denote a non-linear mapping (usually a sequence of NN modules likes convolutions, activation functions, and normalizations).

Instead of modeling

$$x_{l+1} = F(x_l),$$

residual connections model

$$x_{l+1} = x_l + F(x_l). \tag{35.1}$$

This is illustrated in Figure 35.1.

Applying backpropagation to the residual mapping results in the following gradient calculation:

$$\frac{\partial x_{l+1}}{\partial x_l} = \mathbf{I} + \frac{\partial F(x_l)}{\partial x_l}, \tag{35.2}$$

where $\mathbf{I}$ is the identity matrix. The identity matrix is added to the gradient, which helps to stabilize the training of very deep networks. The identity matrix ensures that the gradient is not too small, which can happen if the gradient of F is close to zero.

Figure 35.1.: Residual Connection. Figure credit He et al. (2015)

This is especially important for very deep networks, where the gradient can vanish quickly.

The bias towards the identity matrix guarantees a stable gradient propagation being less effected by F itself.

There have been many variants of ResNet proposed, which mostly concern the function F, or operations applied on the sum. Figure 35.2 shows two different ResNet blocks:

- the original ResNet block, which applies a non-linear activation function, usually ReLU, after the skip connection. and
- the pre-activation ResNet block, which applies the non-linearity at the beginning of F.

For very deep network the pre-activation ResNet has shown to perform better as the gradient flow is guaranteed to have the identity matrix as shown in Equation 35.2, and is not harmed by any non-linear activation applied to it.

35.1.2. Implementation of the Original ResNet Block

One special case we have to handle is when we want to reduce the image dimensions in terms of width and height. The basic ResNet block requires $F(x_l)$ to be of the same shape as x_l. Thus, we need to change the dimensionality of x_l as well before adding to $F(x_l)$. The original implementation used an identity mapping with stride 2 and padded additional feature dimensions with 0. However, the more common implementation is to use a 1x1 convolution with stride 2 as it allows us to change the feature dimensionality while being efficient in parameter and computation cost. The code for the ResNet block is relatively simple, and shown below:

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

Figure 35.2.: ResNet Block. Left: original Residual block in He et al. (2015). Right: pre-activation block. BN describes batch-normalization. Figure credit He et al. (2016)

```
class ResNetBlock(nn.Module):
    def __init__(self, c_in, act_fn, subsample=False, c_out=-1):
        """
        Inputs:
            c_in - Number of input features
            act_fn - Activation class constructor (e.g. nn.ReLU)
            subsample - If True, we need to apply a transformation inside the block to change the fe
            c_out - Number of output features. Note that this is only relevant if subsample is True,
        """
        super().__init__()
        if not subsample:
            c_out = c_in

        # Network representing F
        self.net = nn.Sequential(
            nn.Linear(c_in, c_out, bias=False),  # Linear layer for feature transformation
            nn.BatchNorm1d(c_out),               # Batch normalization for stable learning
            act_fn(),                            # Activation function
            nn.Linear(c_out, c_out, bias=False), # Second linear layer
            nn.BatchNorm1d(c_out)                # Batch normalization
```

```
        )

        # If subsampling, adjust the input feature dimensionality using a linear layer
        self.downsample = nn.Linear(c_in, c_out) if subsample else None
        self.act_fn = act_fn()

    def forward(self, x):
        z = self.net(x)  # Apply the main network
        if self.downsample is not None:
            x = self.downsample(x)  # Adjust dimensionality if necessary
        out = z + x  # Residual connection
        out = self.act_fn(out)  # Apply activation function
        return out

class ResNetRegression(nn.Module):
    def __init__(self, input_dim, output_dim, block, num_blocks=1, hidden_dim=64, act_
        super().__init__()
        self.input_layer = nn.Linear(input_dim, hidden_dim)  # Input layer transformat
        self.blocks = nn.ModuleList([block(hidden_dim, act_fn) for _ in range(num_bloc
        self.output_layer = nn.Linear(hidden_dim, output_dim)  # Output layer for regr

    def forward(self, x):
        x = self.input_layer(x)  # Apply input layer
        for block in self.blocks:
            x = block(x)  # Apply each block
        x = self.output_layer(x)  # Get final output
        return x
```

```
input_dim = 10
output_dim = 1
hidden_dim = 64
model = ResNetRegression(input_dim, output_dim, ResNetBlock, num_blocks=2, hidden_dim=
model
```

```
ResNetRegression(
  (input_layer): Linear(in_features=10, out_features=64, bias=True)
  (blocks): ModuleList(
    (0-1): 2 x ResNetBlock(
      (net): Sequential(
        (0): Linear(in_features=64, out_features=64, bias=False)
        (1): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (2): ReLU()
        (3): Linear(in_features=64, out_features=64, bias=False)
        (4): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
```

```
      )
      (act_fn): ReLU()
    )
  )
  (output_layer): Linear(in_features=64, out_features=1, bias=True)
)
```

```python
# Create a sample input tensor with a batch size of 2
from torchviz import make_dot
sample_input = torch.randn(2, input_dim)

# Generate the visualization
output = model(sample_input)
dot = make_dot(output, params=dict(model.named_parameters()))

# Save and render the visualization
dot.format = 'png'
dot.render('./figures_static/resnet_regression')
```

```
'figures_static/resnet_regression.png'
```

Figure 35.3.: ResNet Regression

35.1.3. Implementation of the Pre-Activation ResNet Block

The second block we implement is the pre-activation ResNet block. For this, we have to change the order of layer in `self.net`, and do not apply an activation function on the output. Additionally, the downsampling operation has to apply a non-linearity as well as the input, x_l, has not been processed by a non-linearity yet. Hence, the block looks as follows:

```python
import torch
import torch.nn as nn

class PreActResNetBlock(nn.Module):
    def __init__(self, c_in, act_fn, subsample=False, c_out=-1):
        super().__init__()
        if not subsample:
            c_out = c_in
        self.net = nn.Sequential(
            nn.LayerNorm(c_in),  # Replacing BatchNorm1d with LayerNorm
            act_fn(),
            nn.Linear(c_in, c_out, bias=False),
            nn.LayerNorm(c_out),
            act_fn(),
            nn.Linear(c_out, c_out, bias=False)
        )
        self.downsample = nn.Sequential(
            nn.LayerNorm(c_in),
            act_fn(),
            nn.Linear(c_in, c_out, bias=False)
        ) if subsample else None

    def forward(self, x):
        z = self.net(x)
        if self.downsample is not None:
            x = self.downsample(x)
        out = z + x
        return out

class PreActResNetRegression(nn.Module):
    def __init__(self, input_dim, output_dim, block, num_blocks=1, hidden_dim=64, act_fn=nn.ReLU):
        super().__init__()
        self.input_layer = nn.Linear(input_dim, hidden_dim)
        self.blocks = nn.ModuleList([block(hidden_dim, act_fn) for _ in range(num_blocks)])
        self.output_layer = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.input_layer(x)
        for block in self.blocks:
            x = block(x)
        x = self.output_layer(x)
        return x
```

```
input_dim = 10
output_dim = 1
hidden_dim = 64
model = PreActResNetRegression(input_dim, output_dim, PreActResNetBlock, num_blocks=2
model
```

```
PreActResNetRegression(
  (input_layer): Linear(in_features=10, out_features=64, bias=True)
  (blocks): ModuleList(
    (0-1): 2 x PreActResNetBlock(
      (net): Sequential(
        (0): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
        (1): ReLU()
        (2): Linear(in_features=64, out_features=64, bias=False)
        (3): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
        (4): ReLU()
        (5): Linear(in_features=64, out_features=64, bias=False)
      )
    )
  )
  (output_layer): Linear(in_features=64, out_features=1, bias=True)
)
```

```
from torchviz import make_dot
# Create a sample input tensor
sample_input = torch.randn(1, input_dim)

# Generate the visualization
output = model(sample_input)
dot = make_dot(output, params=dict(model.named_parameters()))

# Save and render the visualization
dot.format = 'png'
dot.render('./figures_static/preact_resnet_regression')
```

```
'figures_static/preact_resnet_regression.png'
```

Figure 35.4.: Pre-Activation ResNet Regression

35.1.4. The Overall ResNet Architecture

The overall ResNet architecture for regression consists of stacking multiple ResNet blocks, of which some are downsampling the input. When discussing ResNet blocks within the entire network, they are usually grouped by output shape. If we describe

the ResNet as having `[3,3,3]` blocks, it means there are three groups of ResNet blocks, each containing three blocks, with downsampling occurring in the first block of the second and third groups. The final layer produces continuous outputs suitable for regression tasks.

Figure 35.5.: ResNet Notation. Figure credit Lippe (2022)

The `output_dim` parameter is used to determine the number of outputs for regression. This is set to 1 for a single regression target by default, but can be adjusted for multiple targets. Note, a final layer without a softmax or similar classification layer has to be added for regression tasks. A similar notation is used by many other implementations such as in the torchvision library from PyTorch.

Example 35.1 (Example ResNet Model)**.**

```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import make_regression
from types import SimpleNamespace

def get_resnet_blocks_by_name():
    return {"ResNetBlock": ResNetBlock}

def get_act_fn_by_name():
    return {"relu": nn.ReLU}

# Define a simple ResNetBlock for fully connected layers
class ResNetBlock(nn.Module):
    def __init__(self, c_in, act_fn, subsample=False, c_out=-1):
        super().__init__()
        if not subsample:
            c_out = c_in
```

```
        self.net = nn.Sequential(
            nn.Linear(c_in, c_out, bias=False),
            nn.BatchNorm1d(c_out),
            act_fn(),
            nn.Linear(c_out, c_out, bias=False),
            nn.BatchNorm1d(c_out)
        )

        self.downsample = nn.Linear(c_in, c_out) if subsample else None
        self.act_fn = act_fn()

    def forward(self, x):
        z = self.net(x)
        if self.downsample is not None:
            x = self.downsample(x)
        out = z + x
        out = self.act_fn(out)
        return out

# Generate a simple random dataset for regression
num_samples = 100
num_features = 20  # Number of features, typical in a regression dataset
X, y = make_regression(n_samples=num_samples, n_features=num_features, noise=0.1)

# Convert to PyTorch tensors
X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.float32).unsqueeze(1)  # Add a dimension for compatibility

# Define the ResNet model for regression
class ResNet(nn.Module):
    def __init__(self, input_dim, output_dim, num_blocks=[3, 3, 3], c_hidden=[64, 64, 64], act_fn_na
        super().__init__()
        resnet_blocks_by_name = get_resnet_blocks_by_name()
        act_fn_by_name = get_act_fn_by_name()
        assert block_name in resnet_blocks_by_name
        self.hparams = SimpleNamespace(output_dim=output_dim,
                                       c_hidden=c_hidden,
                                       num_blocks=num_blocks,
                                       act_fn_name=act_fn_name,
                                       act_fn=act_fn_by_name[act_fn_name],
                                       block_class=resnet_blocks_by_name[block_name])
        self._create_network(input_dim)
        self._init_params()
```

```
    def _create_network(self, input_dim):
        c_hidden = self.hparams.c_hidden
        self.input_net = nn.Sequential(
            nn.Linear(input_dim, c_hidden[0]),
            self.hparams.act_fn()
        )

        blocks = []
        for block_idx, block_count in enumerate(self.hparams.num_blocks):
            for bc in range(block_count):
                subsample = (bc == 0 and block_idx > 0)
                blocks.append(
                    self.hparams.block_class(c_in=c_hidden[block_idx if not subsample
                                             act_fn=self.hparams.act_fn,
                                             subsample=subsample,
                                             c_out=c_hidden[block_idx])
                )
        self.blocks = nn.Sequential(*blocks)

        self.output_net = nn.Linear(c_hidden[-1], self.hparams.output_dim)

    def _init_params(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            elif isinstance(m, nn.BatchNorm1d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        x = self.input_net(x)
        x = self.blocks(x)
        x = self.output_net(x)
        return x

# Instantiate the model
model = ResNet(input_dim=num_features, output_dim=1)

# Define a loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)

# Example training loop
num_epochs = 10
```

```python
for epoch in range(num_epochs):
    model.train()
    optimizer.zero_grad()

    # Forward pass
    output = model(X_tensor)

    # Compute loss
    loss = criterion(output, y_tensor)

    # Backward pass and optimization
    loss.backward()
    optimizer.step()

    print(f'Epoch {epoch+1}/{num_epochs}, Loss: {loss.item()}')
```

```
Epoch 1/10, Loss: 23964.037109375
Epoch 2/10, Loss: 19316.23828125
Epoch 3/10, Loss: 16658.470703125
Epoch 4/10, Loss: 14944.6103515625
Epoch 5/10, Loss: 13596.298828125
Epoch 6/10, Loss: 12424.3466796875
Epoch 7/10, Loss: 11291.4072265625
Epoch 8/10, Loss: 10234.8203125
Epoch 9/10, Loss: 9260.505859375
Epoch 10/10, Loss: 8365.5498046875
```

36. Neural ODEs

Neural ODEs are related to Residual Neural Networks (ResNets). We consider ResNets in Section 35.1.

36.1. Neural Ordinary Differential Equations

Neural Ordinary Differential Equations (Neural ODEs) are a class of models that are based on ordinary differential equations (ODEs). They are a generalization of ResNets, where the depth of the network is treated as a continuous parameter. Neural ODEs have been introduced by R. T. Q. Chen et al. (2018). We will consider dynamical systems first.

Definition 36.1. A dynamical system is a triple

$$(\mathcal{S}, \mathcal{T}, \Phi)$$

where

- $\mathcal{S}$ is the *state space*
- $\mathcal{T}$ is the *parameter space*, and
- $\Phi : (\mathcal{T} \times \mathcal{S}) \longrightarrow \mathcal{S}$ is the evolution.

Definition 36.1 is a very general definition that includes all sort of dynamical systems. We deal with ODEs where Φ plays the role of the *general solution*: indeed a 1-parameter family of transformations of the state space. $\mathcal{T} = \mathbb{R}_+$ is the time, and usually, $\mathcal{S} = \mathbb{R}^n$ is the state space. The evolution takes a point in space (initial value), a point in time, and returns the a point in space. A general solution to an ODE is a function $y : I \times \mathbb{R}^n \longrightarrow \mathbb{R}^n$: a 1-parameter (usually time is the parameter) family of transformations of the state space. A 1-parameter family of transformations is often called a *flow*.

First-order Ordinary Differential Equations (ODEs) can be defined as follows:

Definition 36.2 (First-Order Ordinary Differential Equation (ODE))**.**

$$\dot{\mathbf{y}}(t) = f(t, \mathbf{y}(t)), \quad \mathbf{y}(t_0) = y_0, \quad f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$$

The solution of the ODE is the function $\mathbf{y}(t)$ that satisfies the ODE and the initial condition, which can be stated as an initial value problems (IVP), i.e. predict $\mathbf{y}(t_1)$ given $\mathbf{y}(t_0)$.

Definition 36.3 (Initial Value Problem (IVP))**.**

$$\mathbf{y}(t_1) = \mathbf{y}(t_0) + \int_{t_0}^{t_1} f(\mathbf{y}(t), t)\mathrm{d}t = \mathrm{ODESolve}(\mathbf{y}(t_0), f, t_0, t_1) \tag{36.1}$$

The existence and uniqueness of solutions to an IVP is ensured by the Picard-Lindelöf theorem, provided the RHS of the ODE is *Lipschitz continuous*. Lipschitz continuity is a property that pops up quite often in ODE-related results in ML.

Definition 36.4 (Lipschitz Continuity)**.** A function $f : X \subset \mathbb{R}^n \longrightarrow \mathbb{R}^n$ is called *Lipschitz continuous* (with constant λ) if

$$||f(x_1) - f(x_2)|| \leq \lambda ||x_1 - x_2|| \quad \forall x_1, x_2 \in X.$$

Note that Lipschitz continuity is a stronger condition than just continuity.

Numerical solvers can be used to perform the forward pass and solve the IVP. If we use, for example, Euler's method, we have the following update rule:

$$\mathbf{y}(t + h) = \mathbf{y}(t) + hf(\mathbf{y}(t), t) \tag{36.2}$$

where h is the step size. The update rule is applied iteratively to solve the IVP. The solution is a discrete approximation of the continuous function $\mathbf{y}(t)$.

Equation 36.2 looks almost identical to a ResNet block (see Equation 35.1). This was one of the main motivations for Neural ODEs (R. T. Q. Chen et al. 2018).

ResNets update hidden states by employing residual connections:

$$\mathbf{y}_{l+1} = \mathbf{y}_l + f(\mathbf{y}_l, \theta_l)$$

where f is a neural network with parameters θ_l, and $\mathbf{y}_l$ and $\mathbf{y}_{l+1}$ are the hidden states at subsequent layers, $l \in \{0, \dots, L\}$.

These updates can be seen as Euler discretizations of continuous transformations.

$$\dot{\mathbf{y}} = f(\mathbf{y}, t, \theta) \tag{36.3}$$

$$\Big\downarrow \text{ Euler Discretization} \tag{36.4}$$

$$\mathbf{y}_{n+1} = \mathbf{y}_n + hf(\mathbf{y}_n, t_n, \theta) \tag{36.5}$$

What happens in a residual network (with step sizes h) if we consider the continuous limit of each discrete layer in the network? What happens as we add more layers and take smaller steps? The answer seems rather astounding: instead of having a discrete number of layers between the input and output domains, we allow the evolution of the hidden states to become continuous.

Figure 36.1.: A residual network defines a discrete sequence of finite transformations. Circles represent evaluation locations. Figure credit R. T. Q. Chen et al. (2018).

The main technical difficulty in training continuous-depth networks is performing back-propagation through the ODE solver. Differentiating through the operations of the forward pass is straightforward, but incurs a high memory cost and introduces additional numerical error.

Figure 36.2.: An ODE network defines a vector field, which continuously transforms the state. Circles represent evaluation locations. Figure credit R. T. Q. Chen et al. (2018).

Pontryagin (1987) treated the ODE solver as a black box, and computed gradients using the adjoint sensitivity method. This approach computes gradients by solving a second, augmented ODE backwards in time, and is applicable to all ODE solvers. It scales linearly with problem size, has low memory cost, and explicitly controls numerical error.

Consider optimizing a scalar-valued loss function $L()$, whose input is the result of an ODE solver:

$$L(y(t_1) = L\left(y(t_0) + \int_{t_0}^{t_1} f(y(t), t, \theta)dt\right) = L\left(\text{ODESolve}(y(t_0), f, t_0, t_1, \theta)\right) \quad (36.6)$$

Equation 36.6 is related to {Equation 36.1}. To optimize L, we require gradients with respect to θ.

Similar to standard neural networks, we start with determining how the gradient of the loss depends on the hidden state $y(t)$ at each instant. This quantity is called the adjoint $a(t) = \frac{\partial L}{\partial y(t)}$. It satisfies the following IVP:

$$\dot{\mathbf{a}}(t) = -\mathbf{a}(t)^\top \frac{\partial f(\mathbf{x}(t), t, \theta)}{\partial \mathbf{x}}, \quad \mathbf{a}(t_1) = \frac{\partial L}{\partial \mathbf{x}(t_1)}.$$

Its dynamics are given by another ODE, which can be thought of as the instantaneous analog of the chain rule:

$$\frac{da(t)}{dt} = -a(t)^T \frac{\partial f(y(t), t, \theta)}{\partial y}.$$

Thus, starting from the initial (remember we are running backwards) value $\mathbf{a}(t_1) = \frac{\partial L}{\partial \mathbf{x}(t_1)}$, we can compute $\mathbf{a}(t_0) = \frac{\partial L}{\partial \mathbf{x}(t_0)}$ by another call to an ODE solver.

Finally, computing the gradients with respect to the parameters θ requires evaluating a third integral, which depends on both $\mathbf{x}(t)$ and $\mathbf{a}(t)$:

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = -\int_{t_1}^{t_0} \mathbf{a}(t)^\top \frac{\partial f}{\partial \theta} \mathrm{d}t,$$

So this method trades off computation for memory—in fact the memory requirement for this gradient calculation is only $\mathcal{O}(1)$ with respect to the number of layers. The corresponding algorithm is described in R. T. Q. Chen et al. (2018), see also Figure 36.3.

Here you can find a very good explanation of the following result based on Lagrange multipliers.

Figure 36.3.: Reverse-mode differentiation of an ODE solution. The adjoint sensitivity method solves an augmented ODE backwards in time. The augmented system contains both the original state and the sensitivity of the loss with respect to the state. If the loss depends directly on the state at multiple observation times, the adjoint state must be updated in the direction of the partial derivative of the loss with respect to each observation. Figure credit R. T. Q. Chen et al. (2018).

36.2. Regression Example

To illustrate this concept, we will consider a simple regression example. This example is based on the Neural-ODEs tutorial from Neural Ordinary Differential Equations, which is provided by R. T. Q. Chen et al. (2018). We will use the ODE solvers from Torchdiffeq.

Neural ODEs, or ODE-Nets, build complex models by chaining together simple building blocks, similar to residual networks. Here, our base layer will define the dynamics of an ODE, which will be interconnected using an ODE solver to form the complete neural network model.

36.2.1. Specifying the Dynamics Layer

The dynamics of an ODE can be captured by the equation:

$$\dot{y}(t) = f(y(t), t, \theta), \qquad y(0) = y_0,$$

where the initial value $y_0 \in \mathbb{R}^n$. The θ parameters were added to the dynamics, so the dynamics function has the dimensions $f : \mathbb{R}^n \times \mathbb{R} \times \mathbb{R}^{|\theta|} \to \mathbb{R}^n$, where $|\theta|$ is the number of parameters we've added to f.

We need the dynamics function to take in the current state $y(t)$ of the ODE, the current time t, and some parameters θ, and output $\frac{\partial y(t)}{\partial t}$, which has the same shape as $y(t)$. They are passed as input to a multi-layer perceptron (MLP). Multiple evaluations of this dynamics layer can be combined using any suitable ODE solver, such as the adaptive-step Dormand-Price solver implemented in the `torchdiffeq` library's `odeint` function.

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
import torchdiffeq
```

Let's start by defining an MLP class to serve as the building block of our models.

```
class MLP(nn.Module):
    def __init__(self, layer_sizes):
        super(MLP, self).__init__()
        layers = []
        for i in range(len(layer_sizes) - 1):
            layers.append(nn.Linear(layer_sizes[i], layer_sizes[i+1]))
```

```
            if i < len(layer_sizes) - 2:
                layers.append(nn.Tanh())
        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)
```

Next, we'll define a ResNet class that uses the MLP as its inner component.

```
class ResNet(nn.Module):
    def __init__(self, layer_sizes, depth):
        super(ResNet, self).__init__()
        self.mlp = MLP(layer_sizes)
        self.depth = depth

    def forward(self, x):
        for _ in range(self.depth):
            x = self.mlp(x) + x
        return x
```

- `ODEFunc` defines how the system evolves over time using the MLP to approximate derivatives $\dot{y}(t)$.
- `ODEBlock` specifies the network structure. It uses `torchdiffeq.odeint` to integrate these dynamics over time.

```
class ODEFunc(nn.Module):
    def __init__(self, layer_sizes):
        super(ODEFunc, self).__init__()
        self.mlp = MLP(layer_sizes)

    def forward(self, t, y):
        t_expanded = t.expand_as(y)
        state_and_time = torch.cat([y, t_expanded], dim=1)
        return self.mlp(state_and_time)

class ODEBlock(nn.Module):
    def __init__(self, odefunc):
        super(ODEBlock, self).__init__()
        self.odefunc = odefunc

    def forward(self, x):
        t = torch.tensor([0.0, 1.0])
        out = torchdiffeq.odeint(self.odefunc, x, t, atol=1e-3, rtol=1e-3)
        return out[1]
```

Generate a toy 1D dataset.

```
inputs = torch.linspace(-2.0, 2.0, 10).reshape(10, 1)
targets = inputs**3 +  0.1 * inputs
```

We specify the hyperparameters for the ResNet and ODE-Net.

```
layer_sizes = [1, 25, 1]
param_scale = 1.0
step_size = 0.01
train_iters = 1000
resnet_depth = 3
```

Initialize and train the ResNet.

```
resnet = ResNet(layer_sizes, resnet_depth)
criterion = nn.MSELoss()
optimizer = optim.SGD(resnet.parameters(), lr=step_size)

for _ in range(train_iters):
    optimizer.zero_grad()
    outputs = resnet(inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    optimizer.step()
```

```
# We need to change the input dimension to 2, to allow time-dependent dynamics.
odenet_layer_sizes = [2, 25, 1]

# Initialize and train ODE-Net.
odefunc = ODEFunc(odenet_layer_sizes)
odenet = ODEBlock(odefunc)
optimizer = optim.SGD(odenet.parameters(), lr=step_size)

for _ in range(train_iters):
    optimizer.zero_grad()
    outputs = odenet(inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    optimizer.step()
```

Finally, plot the predictions of both models.

```
fine_inputs = np.linspace(-3.0, 3.0, 100).reshape(-1, 1)
fine_inputs_tensor = torch.from_numpy(fine_inputs).float()
plt.figure(figsize=(6, 4), dpi=150)
plt.scatter(inputs, targets, color='green', label='Targets')
plt.plot(fine_inputs, resnet(fine_inputs_tensor).detach().numpy(), color='blue', label

plt.plot(fine_inputs, odenet(fine_inputs_tensor).detach().numpy(), color='red', label=
plt.xlabel('Input')
plt.ylabel('Output')
plt.legend()
plt.show()
```

36.3. Further Reading

Neural ODEs have received a lot of attention in the past few years, ever since their introduction in Neurips 2018. Some of many many work in this field include:

- Neural Stochastic Differential Equations (Neural SDEs),
- Neural Controlled Differential Equations (Neural CDEs),
- Graph ODEs,

- Hamiltonial Neural Networks, and
- Lagrangian Neural Networks.

Michael Poli maintains the excellent Awesome Neural ODE, a collection of resources regarding the interplay between neural differential equations, dynamical systems, deep learning, control, numerical methods and scientific machine learning.

Torchdyn is an excellent library for Neural Differential Equations.

Implicit Layers is a list of tutorials on implicit functions and automatic differentiation, Neural ODEs, and Deep Equilibrium Models.

Understanding Neural ODE's is an excellent blogpost on ODEs and Neural ODEs.

Patrick Kidger's doctoral dissertation is an excellent textbook on Neural Differential Equations, see Kidger (2022).

37. Neural ODE Example

37.1. Implementation of a Neural ODE

The following example is based on the "UvA Deep Learning Tutorials" (Lippe 2022).

Example 37.1 (Example: Neural ODE)**.**

```
%matplotlib inline
import time
import logging
import statistics
from typing import Optional, List

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.animation import FuncAnimation
from IPython.display import HTML

import torch
import torch.nn as nn
import torch.utils.data as data
import torch.nn.functional as F
from torch.utils.data import Dataset

try:
    import torchdiffeq
except ModuleNotFoundError:
    !pip install --quiet torchdiffeq
    import torchdiffeq

try:
    import rich
except ModuleNotFoundError:
    !pip install --quiet rich
```

```
    import rich

try:
    # import pytorch_lightning as pl
    import lightning as pl
except ModuleNotFoundError:
    !pip install --quiet pytorch-lightning>=1.4
    # import pytorch_lightning as pl
    import lightning as pl
from torchmetrics.classification import Accuracy

pl.seed_everything(42)

torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Device: {device}")

from torchmetrics.functional import accuracy
```

```
Device: cpu
```

First, we define the core of our Neural ODE model.

```
class _ODEFunc(nn.Module):
    def __init__(self, module, autonomous=True):
        super().__init__()
        self.module = module
        self.autonomous = autonomous

    def forward(self, t, x):
        if not self.autonomous:
            x = torch.cat([torch.ones_like(x[:, [0]]) * t, x], 1)
        return self.module(x)


class ODEBlock(nn.Module):
    def __init__(self, odefunc: nn.Module, solver: str = 'dopri5',
                 rtol: float = 1e-4, atol: float = 1e-4, adjoint: bool = True,
                 autonomous: bool = True):
        super().__init__()
        self.odefunc = _ODEFunc(odefunc, autonomous=autonomous)
```

```
        self.rtol = rtol
        self.atol = atol
        self.solver = solver
        self.use_adjoint = adjoint
        self.integration_time = torch.tensor([0, 1], dtype=torch.float32)

    @property
    def ode_method(self):
        return torchdiffeq.odeint_adjoint if self.use_adjoint else torchdiffeq.odeint

    def forward(self, x: torch.Tensor, adjoint: bool = True, integration_time=None):
        integration_time = self.integration_time if integration_time is None else integration_time
        integration_time = integration_time.to(x.device)
        ode_method =  torchdiffeq.odeint_adjoint if adjoint else torchdiffeq.odeint
        out = ode_method(
            self.odefunc, x, integration_time, rtol=self.rtol,
            atol=self.atol, method=self.solver)
        return out
```

Next, we will wrap everything together in a LightningModule.

```
class Learner(pl.LightningModule):
    def __init__(self, model:nn.Module, t_span:torch.Tensor, learning_rate:float=5e-3):
        super().__init__()
        self.model = model
        self.t_span = t_span
        self.learning_rate = learning_rate
        # self.accuracy = Accuracy(num_classes=2)
        self.accuracy = accuracy

    def forward(self, x):
        return self.model(x)

    def inference(self, x, time_span):
        return self.model(x, adjoint=False, integration_time=time_span)

    def inference_no_projection(self, x, time_span):
        return self.model.forward_no_projection(x, adjoint=False, integration_time=time_span)

    def training_step(self, batch, batch_idx):
        x, y = batch
        y_pred = self(x)
        y_pred = y_pred[-1]  # select last point of solution trajectory
        loss = nn.CrossEntropyLoss()(y_pred, y)
```

```
        self.log('train_loss', loss, prog_bar=True, logger=True)
        return loss

    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_pred = self(x)
        y_pred = y_pred[-1]  # select last point of solution trajectory
        loss = nn.CrossEntropyLoss()(y_pred, y)
        self.log('val_loss', loss, prog_bar=True, logger=True)
        acc = self.accuracy(y_pred.softmax(dim=-1), y, num_classes=2, task="MULTICLASS
        self.log('val_accuracy', acc, prog_bar=True, logger=True)
        return loss

    def test_step(self, batch, batch_idx):
        x, y = batch
        y_pred = self(x)
        y_pred = y_pred[-1]  # select last point of solution trajectory
        loss = nn.CrossEntropyLoss()(y_pred, y)
        self.log('test_loss', loss, prog_bar=True, logger=True)
        acc = self.accuracy(y_pred.softmax(dim=-1), y, num_classes=2, task="MULTICLASS
        self.log('test_accuracy', acc, prog_bar=True, logger=True)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.model.parameters(), lr=self.learning_rate)
        return optimizer
```

We will be working will Half Moons Dataset, a non-linearly separable, binary classification dataset. The code is based on the excellent TorchDyn tutorials (https://github.com/DiffEqML/torchdyn), as well as the original TorchDiffEq examples (https://github.com/rtqichen/torchdiffeq).

```
class MoonsDataset(Dataset):
    """Half Moons Classification Dataset

    Adapted from https://github.com/DiffEqML/torchdyn
    """
    def __init__(self, num_samples=100, noise_std=1e-4):
        self.num_samples = num_samples
        self.noise_std = noise_std
        self.X, self.y = self.generate_moons(num_samples, noise_std)

    @staticmethod
    def generate_moons(num_samples=100, noise_std=1e-4):
```

```
        """Creates a *moons* dataset of `num_samples` data points.
        :param num_samples: number of data points in the generated dataset
        :type num_samples: int
        :param noise_std: standard deviation of noise magnitude added to each data point
        :type noise_std: float
        """
        num_samples_out = num_samples // 2
        num_samples_in = num_samples - num_samples_out
        theta_out = np.linspace(0, np.pi, num_samples_out)
        theta_in = np.linspace(0, np.pi, num_samples_in)
        outer_circ_x = np.cos(theta_out)
        outer_circ_y = np.sin(theta_out)
        inner_circ_x = 1 - np.cos(theta_in)
        inner_circ_y = 1 - np.sin(theta_in) - 0.5

        X = np.vstack([np.append(outer_circ_x, inner_circ_x),
                       np.append(outer_circ_y, inner_circ_y)]).T
        y = np.hstack([np.zeros(num_samples_out), np.ones(num_samples_in)])

        if noise_std is not None:
            X += noise_std * np.random.rand(num_samples, 2)

        X = torch.Tensor(X)
        y = torch.LongTensor(y)
        return X, y

    def __len__(self):
        return self.num_samples

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

```
def plot_binary_classification_dataset(X, y, title=None):
    CLASS_COLORS = ['coral', 'darkviolet']
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.scatter(X[:, 0], X[:, 1], color=[CLASS_COLORS[yi.int()] for yi in y], alpha=0.6)
    ax.set_aspect('equal')
    if title is not None:
        ax.set_title(title)

    return fig, ax
```

Let's create a sample dataset and visualize it.

```
sample_dataset = MoonsDataset(num_samples=400, noise_std=1e-1)
fig, ax = plot_binary_classification_dataset(sample_dataset.X, sample_dataset.y, title
```

Let's now create the train, validation, and test sets, with their corresponding data loaders. We will create a single big dataset and randomly split it in train, val, and test sets.

```
def split_dataset(dataset_size:int, split_percentages:List[float]) -> List[int]:
    split_sizes = [int(pi * dataset_size) for pi in split_percentages]
    split_sizes[0] += dataset_size - sum(split_sizes)
    return split_sizes


class ToyDataModule(pl.LightningDataModule):
    def __init__(self, dataset_size:int, split_percentages:Optional[float]=None):
        super().__init__()
        self.dataset_size = dataset_size
        if split_percentages is None:
            split_percentages = [0.8, 0.1, 0.1]
        self.split_sizes = split_dataset(self.dataset_size, split_percentages)

    def prepare_data(self):
        pass

    def setup(self, stage: Optional[str] = None):
        pass
```

```
    def train_dataloader(self):
        train_loader = torch.utils.data.DataLoader(self.train_set, batch_size=len(self.train_set), s
        return train_loader

    def val_dataloader(self):
        val_loader = torch.utils.data.DataLoader(self.val_set, batch_size=len(self.val_set), shuffle
        return val_loader

    def test_dataloader(self):
        test_loader = torch.utils.data.DataLoader(self.test_set, batch_size=len(self.test_set), shuf
        return test_loader


class HalfMoonsDataModule(ToyDataModule):
    def __init__(self, dataset_size:int, split_percentages:Optional[float]=None):
        super().__init__(dataset_size, split_percentages=split_percentages)

    def setup(self, stage: Optional[str] = None):
        dataset = MoonsDataset(num_samples=self.dataset_size, noise_std=1e-1)
        self.train_set, self.val_set, self.test_set = torch.utils.data.random_split(dataset, self.sp
```

We define a Neural ODE and train it. We will use a simple 2-layer MLP with a *tanh* activation and 64 hidden dimensions. We will train the model using the adjoint method for backpropagation.

A quick note on the architectural choices for our model. The **Picard-Lindelöf theorem** (Coddington and Levinson, 1955) states that the solution to an initial value problem **exists and is unique** if the differential equation is *uniformly Lipschitz continuous* in $\mathbf{z}$ and *continuous* in t. It turns out that this theorem holds for our model if the neural network has finite weights and uses Lipschitz nonlinearities, such as tanh or relu. However, not all tools are our deep learning arsenal is c. For example, as shown in **The Lipschitz Constant of Self-Attention** by Hyunjik Kim et al., standard self-attention is ***not*** Lipschitz. The authors propose alternative forms of self-attention that are Lipschitz.

```
import torch
from lightning.pytorch import Trainer
from lightning.pytorch.callbacks import ModelCheckpoint, RichProgressBar


adjoint = True
data_module = HalfMoonsDataModule(1000)
t_span = torch.linspace(0, 1, 2)
f = nn.Sequential(
```

```
    nn.Linear(2, 64),
    nn.Tanh(),
    nn.Linear(64, 2))
model = ODEBlock(f, adjoint=adjoint)
learner = Learner(model, t_span)

trainer = Trainer(
    max_epochs=200,
    accelerator="gpu" if torch.cuda.is_available() else "cpu",
    devices=1,
    callbacks=[
        ModelCheckpoint(mode="max", monitor="val_accuracy"),
        RichProgressBar(),
    ],
    log_every_n_steps=1,
)
trainer.fit(learner, datamodule=data_module)
val_result = trainer.validate(learner, datamodule=data_module, verbose=True)
test_result = trainer.test(learner, datamodule=data_module, verbose=True)
```

```
     Name    Type       Params   Mode

  0  model   ODEBlock      322   train


Trainable params: 322
Non-trainable params: 0
Total params: 322
Total estimated model params size (MB): 0
Modules in train mode: 6
Modules in eval mode: 0


Output()


/Users/bartz/miniforge3/envs/spot312/lib/python3.12/site-packages/lightning/pytorch/tr
or.py:424: The 'val_dataloader' does not have many workers which may be a bottleneck.
of the `num_workers` argument` to `num_workers=23` in the `DataLoader` to improve perf
```

```
/Users/bartz/miniforge3/envs/spot312/lib/python3.12/site-packages/lightning/pytorch/trainer/connecto
or.py:424: The 'train_dataloader' does not have many workers which may be a bottleneck. Consider inc
value of the `num_workers` argument` to `num_workers=23` in the `DataLoader` to improve performance.
```

```
Output()
```

```
       Validate metric             DataLoader 0

        val_accuracy                    1.0
          val_loss             0.002127678133547306
```

```
Output()
```

```
         Test metric               DataLoader 0

        test_accuracy                   1.0
          test_loss            0.0018281706143170595
```

It seems that in less that 200 epochs we have achieved perfect validation accuracy. Let's now use the trained model to run inference and visualize the trajectories using a dense time span of 100 timesteps.

```
@torch.no_grad()
def run_inference(learner, data_loader, time_span):
    learner.to(device)
    trajectories = []
    classes = []
    time_span = torch.from_numpy(time_span).to(device)
    for data, target in data_loader:
        data = data.to(device)
        traj = learner.inference(data, time_span).cpu().numpy()
        trajectories.append(traj)
        classes.extend(target.numpy())
    trajectories = np.concatenate(trajectories, 1)
    return trajectories, classes

time_span = np.linspace(0.0, 1.0, 100)
trajectories, classes = run_inference(learner, data_module.train_dataloader(), time_sp

colors = ['coral', 'darkviolet']
class_colors = [colors[ci] for ci in classes]
```

We will now define a few functions to visualize the learned trajectories, the state-space, and the learned vector field.

Before we visualize the trajectories, let's plot the (training) data once again:

```
fig, ax = plot_binary_classification_dataset(*data_module.train_set[:], title='Half Mo
```

Below we visualize the evolution for each of the 2 inputs dimensions as a function of time (depth):

```
plot_trajectories(time_span, trajectories, class_colors)
```

And the same evolution combined in a single plot:

```
plot_trajectories_3d(time_span, trajectories, class_colors)
```

The 3D plot can be somewhat complicated to decipher. Thus, we also plot an animated version of the evolution. Each timestep of the animation is a slice on the temporal axis of the figure above.

```
anim = plot_trajectories_animation(time_span, trajectories, colors, classes, lim=8.0)
HTML(anim.to_html5_video())
```

```
<IPython.core.display.HTML object>
```

Finally, we can visualize the state-space diagram and the learned vector field:

```
fig, ax = plt.subplots(2, 1, figsize=(16, 8))
plot_state_space(trajectories, class_colors, ax=ax[0])
plot_static_vector_field(model, trajectories, ax=ax[1], device=device)
```

38. Physics Informed Neural Networks

38.1. PINNs

```
import torch
import numpy as np
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as thdat
import functools
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_theme()
torch.manual_seed(42)

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# boundaries for the frequency range
a = 0
b = 500
```

38.2. Generation and Visualization of the Training Data and the Ground Truth (Function)

- Definition of the (unknown) differential equation:

```
def ode(frequency, loc, sigma, R):
    """Computes the amplitude. Defining equation, used
    to generate data and train models.
    The equation itself is not known to the model.

    Args:
        frequency: (N,) array-like
```

```
        loc: float
        sigma: float
        R: float

    Returns:
        (N,) array-like

    Examples:
        >>> ode(0, 25, 100, 0.005)
        100.0
    """
    A = np.exp(-R * (frequency - loc)**2/sigma**2)
    return A
```

- Setting the parameters for the ode

```
np.random.seed(10)
loc = 250
sigma = 100
R = 0.5
```

- Generating the data

```
frequencies = np.linspace(a, b, 1000)
eq = functools.partial(ode, loc=loc, sigma=sigma, R=R)
amplitudes = eq(frequencies)
```

- Now we have the ground truth for the full frequency range and can take a look at the first 10 values:

```
import pandas as pd
df = pd.DataFrame({'Frequency': frequencies[:10], 'Amplitude': amplitudes[:10]})
print(df)
```

```
   Frequency  Amplitude
0   0.000000   0.043937
1   0.500501   0.044490
2   1.001001   0.045048
3   1.501502   0.045612
4   2.002002   0.046183
5   2.502503   0.046759
6   3.003003   0.047341
7   3.503504   0.047929
8   4.004004   0.048524
9   4.504505   0.049124
```

- We generate the training data as a subset of the full frequency range and add some noise:

```
t = np.linspace(a, 2*b/3, 10)
A = eq(t) +  0.2 * np.random.randn(10)
```

- Plot of the training data and the ground truth:

```
plt.plot(frequencies, amplitudes)
plt.plot(t, A, 'o')
plt.legend(['Equation (ground truth)', 'Training data'])
plt.ylabel('Amplitude')
plt.xlabel('Frequency')
```

```
Text(0.5, 0, 'Frequency')
```

(a)

(b)

Figure 38.1.

38.3. Gradient With Autograd

```
def grad(outputs, inputs):
    """Computes the partial derivative of
    an output with respect to an input.

    Args:
        outputs: (N, 1) tensor
        inputs: (N, D) tensor

    Returns:
        (N, D) tensor

    Examples:
        >>> x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
        >>> y = x**2
        >>> grad(y, x)
        tensor([2., 4., 6.])
    """
    return torch.autograd.grad(
        outputs, inputs, grad_outputs=torch.ones_like(outputs), create_graph=True
    )
```

- Autograd example:

```
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = x**2
grad(y, x)
```

```
(tensor([2., 4., 6.], grad_fn=<MulBackward0>),)
```

38.4. Network

```
def numpy2torch(x):
    """Converts a numpy array to a pytorch tensor.

    Args:
        x: (N, D) array-like
```

```
    Returns:
        (N, D) tensor

    Examples:
        >>> numpy2torch(np.array([1,2,3]))
        tensor([1., 2., 3.])
    """
    n_samples = len(x)
    return torch.from_numpy(x).to(torch.float).to(DEVICE).reshape(n_samples, -1)
```

```
class Net(nn.Module):
    def __init__(
        self,
        input_dim,
        output_dim,
        n_units=100,
        epochs=1000,
        loss=nn.MSELoss(),
        lr=1e-3,
        loss2=None,
        loss2_weight=0.1,
    ) -> None:
        super().__init__()

        self.epochs = epochs
        self.loss = loss
        self.loss2 = loss2
        self.loss2_weight = loss2_weight
        self.lr = lr
        self.n_units = n_units

        self.layers = nn.Sequential(
            nn.Linear(input_dim, self.n_units),
            nn.ReLU(),
            nn.Linear(self.n_units, self.n_units),
            nn.ReLU(),
            nn.Linear(self.n_units, self.n_units),
            nn.ReLU(),
            nn.Linear(self.n_units, self.n_units),
            nn.ReLU(),
        )
        self.out = nn.Linear(self.n_units, output_dim)
```

```
    def forward(self, x):
        h = self.layers(x)
        out = self.out(h)
        return out

    def fit(self, X, y):
        Xt = numpy2torch(X)
        yt = numpy2torch(y)

        optimiser = optim.Adam(self.parameters(), lr=self.lr)
        self.train()
        losses = []
        for ep in range(self.epochs):
            optimiser.zero_grad()
            outputs = self.forward(Xt)
            loss = self.loss(yt, outputs)
            if self.loss2:
                loss += self.loss2_weight + self.loss2_weight * self.loss2(self)
            loss.backward()
            optimiser.step()
            losses.append(loss.item())
            if ep % int(self.epochs / 10) == 0:
                print(f"Epoch {ep}/{self.epochs}, loss: {losses[-1]:.2f}")
        return losses

    def predict(self, X):
        self.eval()
        out = self.forward(numpy2torch(X))
        return out.detach().cpu().numpy()
```

- Extended network for parameter estimation of parameter `r`:

```
class PINNParam(Net):
    def __init__(
        self,
        input_dim,
        output_dim,
        n_units=100,
        epochs=1000,
        loss=nn.MSELoss(),
        lr=0.001,
        loss2=None,
        loss2_weight=0.1,
    ) -> None:
```

```
        super().__init__(
            input_dim, output_dim, n_units, epochs, loss, lr, loss2, loss2_weight
        )

        self.r = nn.Parameter(data=torch.tensor([1.]))
        self.sigma = nn.Parameter(data=torch.tensor([100.]))
        self.loc = nn.Parameter(data=torch.tensor([100.]))
```

38.5. Basic Neutral Network

- Network without regularization:

```
net = Net(1,1, loss2=None, epochs=2000, lr=1e-5).to(DEVICE)

losses = net.fit(t, A)

plt.plot(losses)
plt.yscale('log')
```

```
Epoch 0/2000, loss: 6.59
Epoch 200/2000, loss: 0.06
Epoch 400/2000, loss: 0.05
Epoch 600/2000, loss: 0.05
Epoch 800/2000, loss: 0.05
Epoch 1000/2000, loss: 0.05
Epoch 1200/2000, loss: 0.05
Epoch 1400/2000, loss: 0.05
Epoch 1600/2000, loss: 0.05
Epoch 1800/2000, loss: 0.05
```

Figure 38.2.

- Adding L2 regularization:

```
def l2_reg(model: torch.nn.Module):
    """L2 regularization for the model parameters.

    Args:
        model: torch.nn.Module

    Returns:
        torch.Tensor

    Examples:
        >>> l2_reg(Net(1,1))
        tensor(0.0001, grad_fn=<SumBackward0>)
    """
    return torch.sum(sum([p.pow(2.) for p in model.parameters()]))
```

```
netreg = Net(1,1, loss2=l2_reg, epochs=20000, lr=1e-5, loss2_weight=.1).to(DEVICE)
losses = netreg.fit(t, A)
plt.plot(losses)
plt.yscale('log')
```

```
Epoch 0/20000, loss: 662.07
Epoch 2000/20000, loss: 612.81
Epoch 4000/20000, loss: 571.31
Epoch 6000/20000, loss: 533.74
Epoch 8000/20000, loss: 499.32
Epoch 10000/20000, loss: 467.44
Epoch 12000/20000, loss: 437.53
Epoch 14000/20000, loss: 409.15
Epoch 16000/20000, loss: 382.10
Epoch 18000/20000, loss: 356.29
```

Figure 38.3.

```
predsreg = netreg.predict(frequencies)
preds = net.predict(frequencies)
plt.plot(frequencies, amplitudes, alpha=0.8)
plt.plot(t, A, 'o')
plt.plot(frequencies, preds, alpha=0.8)
plt.plot(frequencies, predsreg, alpha=0.8)

plt.legend(labels=['Equation','Training data', 'Network', 'L2 Regularization Network'])
plt.ylabel('Amplitude')
plt.xlabel('Frequency')
```

```
Text(0.5, 0, 'Frequency')
```

(a)

(b)

Figure 38.4.

38.6. PINNs

- Calculate the physics-informed loss (similar to the L2 regularization):

```
def physics_loss(model: torch.nn.Module):
    """Computes the physics-informed loss for the model.

    Args:
        model: torch.nn.Module

    Returns:
        torch.Tensor

    Examples:
        >>> physics_loss(Net(1,1))
```

```
        tensor(0.0001, grad_fn=<MeanBackward0>)
    """
    ts = torch.linspace(a, b, steps=1000).view(-1,1).requires_grad_(True).to(DEVICE)
    amplitudes = model(ts)
    dT = grad(amplitudes, ts)[0]
    ode = -2*R*(ts-loc)/ sigma**2 * amplitudes - dT
    return torch.mean(ode**2)
```

- Train the network with the physics-informed loss and plot the training error:

```
net_pinn = Net(1,1, loss2=physics_loss, epochs=2000, loss2_weight=1, lr=1e-5).to(DEVICE)
losses = net_pinn.fit(t, A)
plt.plot(losses)
plt.yscale('log')
```

```
Epoch 0/2000, loss: 12.23
Epoch 200/2000, loss: 1.05
Epoch 400/2000, loss: 1.05
Epoch 600/2000, loss: 1.05
Epoch 800/2000, loss: 1.05
Epoch 1000/2000, loss: 1.05
Epoch 1200/2000, loss: 1.05
Epoch 1400/2000, loss: 1.05
Epoch 1600/2000, loss: 1.05
Epoch 1800/2000, loss: 1.05
```

Figure 38.5.

- Predict the amplitude and plot the results:

```
preds_pinn = net_pinn.predict(frequencies)
plt.plot(frequencies, amplitudes, alpha=0.8)
plt.plot(t, A, 'o')
plt.plot(frequencies, preds, alpha=0.8)
plt.plot(frequencies, predsreg, alpha=0.8)
plt.plot(frequencies, preds_pinn, alpha=0.8)
plt.legend(labels=['Equation','Training data', 'NN', "R2", 'PINN'])
plt.ylabel('Amplitude')
plt.xlabel('Frequency')
```

```
Text(0.5, 0, 'Frequency')
```

(a)

(b)

Figure 38.6.

38.6.1. PINNs: Parameter Estimation

```
def physics_loss_estimation(model: torch.nn.Module):
    ts = torch.linspace(a, b, steps=1000,).view(-1,1).requires_grad_(True).to(DEVICE)
    amplitudes = model(ts)
    dT = grad(amplitudes, ts)[0]
    ode = -2*model.r*(ts-model.loc)/ (model.sigma)**2 * amplitudes - dT
    return torch.mean(ode**2)
```

```
pinn_param = PINNParam(1, 1, loss2=physics_loss_estimation, loss2_weight=1, epochs=4000, lr= 5e-6).t
losses = pinn_param.fit(t, A)
plt.plot(losses)
plt.yscale('log')
```

```
Epoch 0/4000, loss: 15.06
```

```
Epoch 400/4000, loss: 1.06
Epoch 800/4000, loss: 1.06
Epoch 1200/4000, loss: 1.06
Epoch 1600/4000, loss: 1.05
Epoch 2000/4000, loss: 1.05
Epoch 2400/4000, loss: 1.05
Epoch 2800/4000, loss: 1.05
Epoch 3200/4000, loss: 1.05
Epoch 3600/4000, loss: 1.05
```

Figure 38.7.

```python
preds_disc = pinn_param.predict(frequencies)
print(f"Estimated r: {pinn_param.r}")
print(f"Estimated sigma: {pinn_param.sigma}")
print(f"Estimated loc: {pinn_param.loc}")
```

```
Estimated r: Parameter containing:
tensor([0.9893], requires_grad=True)
Estimated sigma: Parameter containing:
tensor([100.0067], requires_grad=True)
Estimated loc: Parameter containing:
tensor([100.0065], requires_grad=True)
```

```
plt.plot(frequencies, amplitudes, alpha=0.8)
plt.plot(t, A, 'o')
plt.plot(frequencies, preds_disc, alpha=0.8)
plt.legend(labels=['Equation','Training data', 'estimation PINN'])
plt.ylabel('Amplitude')
plt.xlabel('Frequency')
```

```
Text(0.5, 0, 'Frequency')
```

(a)

(b)

Figure 38.8.

```
plt.plot(frequencies, amplitudes, alpha=0.8)
plt.plot(t, A, 'o')
plt.plot(frequencies, preds, alpha=0.8)
plt.plot(frequencies, predsreg, alpha=0.8)
plt.plot(frequencies, preds_pinn, alpha=0.8)
plt.plot(frequencies, preds_disc, alpha=0.8)
plt.legend(labels=['Equation','Training data', 'NN', "R2", 'PINN', 'paramPINN'])
plt.ylabel('Amplitude')
```

```
plt.xlabel('Frequency')
```

```
Text(0.5, 0, 'Frequency')
```

(a)

(b)

Figure 38.9.

```
plt.plot(frequencies, amplitudes, alpha=0.8)
plt.plot(t, A, 'o')
plt.plot(frequencies, preds, alpha=0.8)
plt.plot(frequencies, predsreg, alpha=0.8)
plt.plot(frequencies, preds_disc, alpha=0.8)
plt.legend(labels=['Equation','Training data', 'NN', "R2", 'paramPINN'])
plt.ylabel('Amplitude')
plt.xlabel('Frequency')
```

```
Text(0.5, 0, 'Frequency')
```

(a)

(b)

Figure 38.10.

```
plt.plot(frequencies, amplitudes, alpha=0.8)
plt.plot(t, A, 'o')
plt.plot(frequencies, preds, alpha=0.8)
plt.plot(frequencies, predsreg, alpha=0.8)
plt.plot(frequencies, preds_disc, alpha=0.8)
plt.legend(labels=['Grundwahrheit','Trainingsdaten', 'NN', "NN+R2", 'PINN'])
plt.ylabel('Amplitude')
plt.xlabel('Frequenz')
# save the plot as a pdf
plt.savefig('pinns.pdf')
plt.savefig('pinns.png')
```

Figure 38.11.

38.7. Summary

- Results strongly depend on the parametrization(s)
- PINN parameter estimation not robust
- Hyperparameter tuning is crucial
- Use SPOT before further analysis is done

39. Hyperparameter Tuning with PyTorch Lightning: Physics Informed Neural Networks

39.1. PINNs

In this section, we will show how to set up PINN hyperparameter tuner from scratch based on the `spotpython` programs from Chapter 34.

39.1.1. The Ground Truth Model

Definition of the (unknown) differential equation:

```
import torch
import numpy as np
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as thdat
import functools
import matplotlib.pyplot as plt
import seaborn as sns
# boundaries for the frequency range
a = 0
b = 500

def ode(frequency, loc, sigma, R):
    """Computes the amplitude. Defining equation, used
    to generate data and train models.
    The equation itself is not known to the model.

    Args:
        frequency: (N,) array-like
        loc: float
        sigma: float
```

```
        R: float

    Returns:
        (N,) array-like

    Examples:
        >>> ode(0, 25, 100, 0.005)
        100.0
    """
    A = np.exp(-R * (frequency - loc)**2/sigma**2)
    return A
```

Setting the parameters for the ode

```
np.random.seed(10)
loc = 250
sigma = 100
R = 0.5
```

- Generating the data

```
frequencies = np.linspace(a, b, 1000)
eq = functools.partial(ode, loc=loc, sigma=sigma, R=R)
amplitudes = eq(frequencies)
```

- Now we have the ground truth for the full frequency range and can take a look at the first 10 values:

```
df = pd.DataFrame({'Frequency': frequencies[:10], 'Amplitude': amplitudes[:10]})
print(df)
```

```
   Frequency  Amplitude
0   0.000000   0.043937
1   0.500501   0.044490
2   1.001001   0.045048
3   1.501502   0.045612
4   2.002002   0.046183
5   2.502503   0.046759
6   3.003003   0.047341
7   3.503504   0.047929
8   4.004004   0.048524
9   4.504505   0.049124
```

- We generate the training data as a subset of the full frequency range and add some noise:

```
# Make training data
t = np.linspace(a, 2*b/3, 10)
A = eq(t) +  0.2 * np.random.randn(10)
```

- Plot of the training data and the ground truth:

```
plt.plot(frequencies, amplitudes)
plt.plot(t, A, 'o')
plt.legend(['Equation (ground truth)', 'Training data'])
plt.ylabel('Amplitude')
plt.xlabel('Frequency')
```

```
Text(0.5, 0, 'Frequency')
```

39.1.2. Required Files

We use the files from the `/userModel` directory as templates. They are renamed as follows:

- `my_regressor.py` ⇒ `pinn_regressor.py`, see Section 39.1.4
- `my_hyperdict.json` ⇒ `pinn_hyperdict.py`, see Section 39.1.5
- `my_hyperdict.py` ⇒ `pinn_hyperdict.py`, see Section 39.1.3.

39.1.3. The New `pinn_hyperdict.py` File

Modifying the `pin_hyperdict.py` file is very easy. We simply have to change the class-name `MyHyperDict` to `PINNHyperDict` and the `filename` from `"my_hyper_dict.json"` to `"pinn_hyper_dict.json"`. The file is shown below.

```
import json
from spotpython.data import base
import pathlib

class PINNHyperDict(base.FileConfig):
    def __init__(
        self,
        filename: str = "pinn_hyper_dict.json",
        directory: None = None,
    ) -> None:
        super().__init__(filename=filename, directory=directory)
        self.filename = filename
        self.directory = directory
        self.hyper_dict = self.load()

    @property
    def path(self):
        if self.directory:
            return pathlib.Path(self.directory).joinpath(self.filename)
        return pathlib.Path(__file__).parent.joinpath(self.filename)

    def load(self) -> dict:
        with open(self.path, "r") as f:
            d = json.load(f)
        return d
```

39.1.4. The New `pinn_regressor.py` File

> **Warning**
>
> The document is not complete. The code below is a template and needs to be modified to work with the PINN model.

```
import lightning as L
import torch
from torch import nn
from spotpython.hyperparameters.optimizer import optimizer_handler
import torchmetrics.functional.regression

class PINNRegressor(L.LightningModule):
    """
    A LightningModule class for a regression neural network model.

    Attributes:
        l1 (int):
            The number of neurons in the first hidden layer.
        epochs (int):
            The number of epochs to train the model for.
        batch_size (int):
            The batch size to use during training.
        initialization (str):
            The initialization method to use for the weights.
        act_fn (nn.Module):
            The activation function to use in the hidden layers.
        optimizer (str):
            The optimizer to use during training.
        dropout_prob (float):
            The probability of dropping out a neuron during training.
        lr_mult (float):
            The learning rate multiplier for the optimizer.
        patience (int):
            The number of epochs to wait before early stopping.
        _L_in (int):
            The number of input features.
        _L_out (int):
            The number of output classes.
        _torchmetric (str):
            The metric to use for the loss function. If `None`,
            then "mean_squared_error" is used.
        layers (nn.Sequential):
```

```
            The neural network model.

    """

    def __init__(
        self,
        l1: int,
        epochs: int,
        batch_size: int,
        initialization: str,
        act_fn: nn.Module,
        optimizer: str,
        dropout_prob: float,
        lr_mult: float,
        patience: int,
        _L_in: int,
        _L_out: int,
        _torchmetric: str,
    ):
        """
        Initializes the MyRegressor object.

        Args:
            l1 (int):
                The number of neurons in the first hidden layer.
            epochs (int):
                The number of epochs to train the model for.
            batch_size (int):
                The batch size to use during training.
            initialization (str):
                The initialization method to use for the weights.
            act_fn (nn.Module):
                The activation function to use in the hidden layers.
            optimizer (str):
                The optimizer to use during training.
            dropout_prob (float):
                The probability of dropping out a neuron during training.
            lr_mult (float):
                The learning rate multiplier for the optimizer.
            patience (int):
                The number of epochs to wait before early stopping.
            _L_in (int):
                The number of input features. Not a hyperparameter, but needed to crea
            _L_out (int):
```

```
            The number of output classes. Not a hyperparameter, but needed to create the network
        _torchmetric (str):
            The metric to use for the loss function. If `None`,
            then "mean_squared_error" is used.

    Returns:
        (NoneType): None

    Raises:
        ValueError: If l1 is less than 4.

    """
    super().__init__()
    # Attribute 'act_fn' is an instance of `nn.Module` and is already saved during
    # checkpointing. It is recommended to ignore them
    # using `self.save_hyperparameters(ignore=['act_fn'])`
    # self.save_hyperparameters(ignore=["act_fn"])
    #
    self._L_in = _L_in
    self._L_out = _L_out
    if _torchmetric is None:
        _torchmetric = "mean_squared_error"
    self._torchmetric = _torchmetric
    self.metric = getattr(torchmetrics.functional.regression, _torchmetric)
    # _L_in and _L_out are not hyperparameters, but are needed to create the network
    # _torchmetric is not a hyperparameter, but is needed to calculate the loss
    self.save_hyperparameters(ignore=["_L_in", "_L_out", "_torchmetric"])
    # set dummy input array for Tensorboard Graphs
    # set log_graph=True in Trainer to see the graph (in traintest.py)
    self.example_input_array = torch.zeros((batch_size, self._L_in))
    if self.hparams.l1 < 4:
        raise ValueError("l1 must be at least 4")
    hidden_sizes = self._get_hidden_sizes()
    # Create the network based on the specified hidden sizes
    layers = []
    layer_sizes = [self._L_in] + hidden_sizes
    layer_size_last = layer_sizes[0]
    for layer_size in layer_sizes[1:]:
        layers += [
            nn.Linear(layer_size_last, layer_size),
            self.hparams.act_fn,
            nn.Dropout(self.hparams.dropout_prob),
        ]
        layer_size_last = layer_size
```

```
        layers += [nn.Linear(layer_sizes[-1], self._L_out)]
        # nn.Sequential summarizes a list of modules into a single module, applying th
        self.layers = nn.Sequential(*layers)

    def _generate_div2_list(self, n, n_min) -> list:
        """
        Generate a list of numbers from n to n_min (inclusive) by dividing n by 2
        until the result is less than n_min.
        This function starts with n and keeps dividing it by 2 until n_min is reached
        The number of times each value is added to the list is determined by n // curr
        No more than 4 repeats of the same value (`max_repeats` below) are added to th

        Args:
            n (int): The number to start with.
            n_min (int): The minimum number to stop at.

        Returns:
            list: A list of numbers from n to n_min (inclusive).

        Examples:
            _generate_div2_list(10, 1)
            [10, 5, 5, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
            _ generate_div2_list(10, 2)
            [10, 5, 5, 2, 2, 2, 2, 2]
        """
        result = []
        current = n
        repeats = 1
        max_repeats = 4
        while current >= n_min:
            result.extend([current] * min(repeats, max_repeats))
            current = current // 2
            repeats = repeats + 1
        return result

    def _get_hidden_sizes(self):
        """
        Generate the hidden layer sizes for the network.

        Returns:
            list: A list of hidden layer sizes.

        """
        n_low = self._L_in // 4
```

```
        n_high = max(self.hparams.l1, 2 * n_low)
        hidden_sizes = self._generate_div2_list(n_high, n_low)
        return hidden_sizes

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Performs a forward pass through the model.

        Args:
            x (torch.Tensor): A tensor containing a batch of input data.

        Returns:
            torch.Tensor: A tensor containing the output of the model.

        """
        x = self.layers(x)
        return x

    def _calculate_loss(self, batch):
        """
        Calculate the loss for the given batch.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.

        """
        x, y = batch
        y = y.view(len(y), 1)
        y_hat = self(x)
        loss = self.metric(y_hat, y)
        return loss

    def training_step(self, batch: tuple) -> torch.Tensor:
        """
        Performs a single training step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.
```

```
        """
        val_loss = self._calculate_loss(batch)
        # self.log("train_loss", val_loss, on_step=True, on_epoch=True, prog_bar=True)
        # self.log("train_mae_loss", mae_loss, on_step=True, on_epoch=True, prog_bar=T
        return val_loss

    def validation_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) ->
        """
        Performs a single validation step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults t

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.

        """
        val_loss = self._calculate_loss(batch)
        # self.log("val_loss", val_loss, on_step=False, on_epoch=True, prog_bar=prog_b
        self.log("val_loss", val_loss, prog_bar=prog_bar)
        self.log("hp_metric", val_loss, prog_bar=prog_bar)
        return val_loss

    def test_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> torch
        """
        Performs a single test step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults t

        Returns:
            torch.Tensor: A tensor containing the loss for this batch.
        """
        val_loss = self._calculate_loss(batch)
        self.log("val_loss", val_loss, prog_bar=prog_bar)
        self.log("hp_metric", val_loss, prog_bar=prog_bar)
        return val_loss

    def predict_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> to
        """
```

```
        Performs a single prediction step.

        Args:
            batch (tuple): A tuple containing a batch of input data and labels.
            batch_idx (int): The index of the current batch.
            prog_bar (bool, optional): Whether to display the progress bar. Defaults to False.

        Returns:
            A tuple containing the input data, the true labels, and the predicted values.
        """
        x, y = batch
        yhat = self(x)
        y = y.view(len(y), 1)
        yhat = yhat.view(len(yhat), 1)
        print(f"Predict step x: {x}")
        print(f"Predict step y: {y}")
        print(f"Predict step y_hat: {yhat}")
        # pred_loss = F.mse_loss(y_hat, y)
        # pred loss not registered
        # self.log("pred_loss", pred_loss, prog_bar=prog_bar)
        # self.log("hp_metric", pred_loss, prog_bar=prog_bar)
        # MisconfigurationException: You are trying to `self.log()`
        # but the loop's result collection is not registered yet.
        # This is most likely because you are trying to log in a `predict` hook, but it doesn't supp
        # If you want to manually log, please consider using `self.log_dict({'pred_loss': pred_loss}
        return (x, y, yhat)

    def configure_optimizers(self) -> torch.optim.Optimizer:
        """
        Configures the optimizer for the model.

        Notes:
            The default Lightning way is to define an optimizer as
            `optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)`.
            spotpython uses an optimizer handler to create the optimizer, which
            adapts the learning rate according to the lr_mult hyperparameter as
            well as other hyperparameters. See `spotpython.hyperparameters.optimizer.py` for details

        Returns:
            torch.optim.Optimizer: The optimizer to use during training.

        """
        # optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)
        optimizer = optimizer_handler(
```

```
            optimizer_name=self.hparams.optimizer, params=self.parameters(), lr_mult=s
        )
        return optimizer
```

39.1.5. The New `pinn_hyperdict.json` File

40. Explainable AI with SpotPython and Pytorch

```
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_control_init)
from spotpython.spot import Spot
from spotpython.utils.file import get_experiment_filename
from spotpython.hyperparameters.values import set_hyperparameter
from math import inf

PREFIX="602_12_1"

data_set = Diabetes()

fun_control = fun_control_init(
    save_experiment=True,
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1)

fun = HyperLight().fun


set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,7])
set_hyperparameter(fun_control, "epochs", [10,12])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,9])
```

```
design_control = design_control_init(init_size=7)

S = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
```

```
module_name: light
submodule_name: regression
model_name: NNLinearRegressor
Experiment saved to 602_12_1_exp.pkl
```

40.1. Running the Hyperparameter Tuning or Loading the Existing Model

```
S.run()
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}

train_model result: {'val_loss': 3785.54541015625, 'hp_metric': 3785.54541015625}

train_model result: {'val_loss': 3005.0224609375, 'hp_metric': 3005.0224609375}

train_model result: {'val_loss': 4960.9326171875, 'hp_metric': 4960.9326171875}

train_model result: {'val_loss': 3578.1474609375, 'hp_metric': 3578.1474609375}

train_model result: {'val_loss': 5508.482421875, 'hp_metric': 5508.482421875}
train_model result: {'val_loss': 4394.2734375, 'hp_metric': 4394.2734375}

train_model result: {'val_loss': 4790.76904296875, 'hp_metric': 4790.76904296875}
spotpython tuning: 3005.0224609375 [#---------] 14.38%

train_model result: {'val_loss': 487075.875, 'hp_metric': 487075.875}
spotpython tuning: 3005.0224609375 [##########] 100.00% Done...

Experiment saved to 602_12_1_res.pkl


<spotpython.spot.spot.Spot at 0x14eb7f980>
```

40.2. Results from the Hyperparameter Tuning Experiment

- After the hyperparameter tuning is finished, the following information is available:
  - the `S` object and the associated
  - `fun_control` dictionary

```
S.print_results(print_screen=True)
```

```
min y: 3005.0224609375
l1: 4.0
epochs: 12.0
batch_size: 5.0
act_fn: 2.0
optimizer: 1.0
dropout_prob: 0.0035182324401420283
lr_mult: 5.090825861997777
patience: 2.0
batch_norm: 0.0
initialization: 1.0
```

```
[['l1', np.float64(4.0)],
 ['epochs', np.float64(12.0)],
 ['batch_size', np.float64(5.0)],
 ['act_fn', np.float64(2.0)],
 ['optimizer', np.float64(1.0)],
 ['dropout_prob', np.float64(0.0035182324401420283)],
 ['lr_mult', np.float64(5.090825861997777)],
 ['patience', np.float64(2.0)],
 ['batch_norm', np.float64(0.0)],
 ['initialization', np.float64(1.0)]]
```

```
S.plot_progress()
```

40.2.1. Getting the Best Model, i.e, the Tuned Architecture

- The method `get_tuned_architecture` [DOC] returns the best model architecture found during the hyperparameter tuning.
- It returns the transformed values, i.e., `batch_size = 2^x` if the hyperparameter `batch_size` was transformed with the `transform_power_2_int` function.

```
from spotpython.hyperparameters.values import get_tuned_architecture
import pprint
config = get_tuned_architecture(S)
pprint.pprint(config)
```

```
{'act_fn': ReLU(),
 'batch_norm': False,
 'batch_size': 32,
 'dropout_prob': 0.0035182324401420283,
 'epochs': 4096,
 'initialization': 'kaiming_uniform',
 'l1': 16,
 'lr_mult': 5.090825861997777,
 'optimizer': 'Adam',
 'patience': 4}
```

- Note: `get_tuned_architecture` has the option `force_minX` which does not have any effect in this case.

```
from spotpython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(S, force_minX=True)
pprint.pprint(config)
```

```
{'act_fn': ReLU(),
 'batch_norm': False,
 'batch_size': 32,
 'dropout_prob': 0.0035182324401420283,
 'epochs': 4096,
 'initialization': 'kaiming_uniform',
 'l1': 16,
 'lr_mult': 5.090825861997777,
 'optimizer': 'Adam',
 'patience': 4}
```

40.3. Training the Tuned Architecture on the Test Data

- Since we are interested in the explainability of the model, we will train the tuned architecture on the test data.
- `spotpythons`'s `test_model` function [DOC] is used to train the model on the test data.
- Note: Until now, we do not use any information about the NN's weights and biases. Only the architecture, which is available as the `config`, is used.
- `spotpython` used the TensorBoard logger to save the training process in the `./runs` directory. Therefore, we have to enable the TensorBoard logger in the `fun_control` dictionary. To get a clean start, we remove an existing `runs` folder.

```
from spotpython.light.testmodel import test_model
from spotpython.light.loadmodel import load_light_from_checkpoint
fun_control.update({"tensorboard_log": True})
test_model(config, fun_control)
```

```
        Test metric               DataLoader 0

         hp_metric               3726.458984375
         val_loss                3726.458984375


test_model result: {'val_loss': 3726.458984375, 'hp_metric': 3726.458984375}


(3726.458984375, 3726.458984375)
```

```
model = load_light_from_checkpoint(config, fun_control)
```

```
config: {'l1': 16, 'epochs': 4096, 'batch_size': 32, 'act_fn': ReLU(), 'optimizer': 'A
Loading model with 16_4096_32_ReLU_Adam_0.0035_5.0908_4_False_kaiming_uniform_TEST fro
Model: NNLinearRegressor(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=320, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.0035182324401420283, inplace=False)
    (3): Linear(in_features=320, out_features=160, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.0035182324401420283, inplace=False)
    (6): Linear(in_features=160, out_features=320, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.0035182324401420283, inplace=False)
    (9): Linear(in_features=320, out_features=160, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.0035182324401420283, inplace=False)
    (12): Linear(in_features=160, out_features=160, bias=True)
    (13): ReLU()
    (14): Dropout(p=0.0035182324401420283, inplace=False)
    (15): Linear(in_features=160, out_features=80, bias=True)
    (16): ReLU()
    (17): Dropout(p=0.0035182324401420283, inplace=False)
    (18): Linear(in_features=80, out_features=80, bias=True)
    (19): ReLU()
    (20): Dropout(p=0.0035182324401420283, inplace=False)
    (21): Linear(in_features=80, out_features=1, bias=True)
  )
)
```

40.3.0.1. Details of the Training Process on the Test Data

- The `test_model` method initializes the model with the tuned architecture as follows:

```
model = fun_control["core_model"](**config, _L_in=_L_in, _L_out=_L_out, _torchmetric=_
```

- Then, the Lightning Trainer is initialized with the `fun_control` dictionary and the model as follows:

```
    trainer = L.Trainer(
    default_root_dir=os.path.join(fun_control["CHECKPOINT_PATH"], config_id),
    max_epochs=model.hparams.epochs,
    accelerator=fun_control["accelerator"],
    devices=fun_control["devices"],
    logger=TensorBoardLogger(
        save_dir=fun_control["TENSORBOARD_PATH"],
        version=config_id,
        default_hp_metric=True,
        log_graph=fun_control["log_graph"],
    ),
    callbacks=[
        EarlyStopping(monitor="val_loss", patience=config["patience"], mode="min", strict=False
        ModelCheckpoint(
            dirpath=os.path.join(fun_control["CHECKPOINT_PATH"], config_id), save_last=True
        ),
    ],
    enable_progress_bar=enable_progress_bar,
)
trainer.fit(model=model, datamodule=dm)
test_result = trainer.test(datamodule=dm, ckpt_path="last")
```

- As shown in the code above, the last checkpoint ist saved.
- `spotpython`'s method `load_light_from_checkpoint` is used to load the last checkpoint and to get the model's weights and biases. It requires the `fun_control` dictionary and the `config_id` as input to find the correct checkpoint.
- Now, the model is trained and the weights and biases are available.

40.4. Visualizing the Neural Network Architecture

```
# get the device
from spotpython.utils.device import getDevice
device = getDevice()
```

```
from spotpython.plot.xai import viz_net
viz_net(model, device=device)
```

Figure 40.1.: architecture

40.5. XAI Methods

- `spotpython` provides methods to explain the model's predictions. The following neural network elements can be analyzed:

40.5.1. Weights

- Weights are the parameters of the neural network that are learned from the data during training. They connect neurons between layers and determine the

strength and direction of the signal sent from one neuron to another. The network adjusts the weights during training to minimize the error between the predicted output and the actual output.
- Interpretation of the weights: A high weight value indicates a strong influence of the input neuron on the output. Positive weights suggest a positive correlation, whereas negative weights suggest an inverse relationship between neurons.

40.5.2. Activations

- Activations are the outputs produced by neurons after applying an activation function to the weighted sum of inputs. The activation function (e.g., ReLU, sigmoid, tanh) adds non-linearity to the model, allowing it to learn more complex relationships.
- Interpretation of the activations: The value of activations indicates the intensity of the signal passed to the next layer. Certain activation patterns can highlight which features or parts of the data the network is focusing on.

40.5.3. Gradients

- Gradients are the partial derivatives of the loss function with respect to different parameters (weights) of the network. During backpropagation, gradients are used to update the weights in the direction that reduces the loss by methods like gradient descent.
- Interpretation of the gradients: The magnitude of the gradient indicates how much a parameter should change to reduce the error. A large gradient implies a steeper slope and a bigger update, while a small gradient suggests that the parameter is near an optimal point. If gradients are too small (vanishing gradient problem), the network may learn slowly or stop learning. If they are too large (exploding gradient problem), the updates may be unstable.
- `sptpython` provides the method `get_gradients` to get the gradients of the model.

```
from spotpython.plot.xai import (get_activations, get_gradients, get_weights, visualize_weights, vis
batch_size = config["batch_size"]
```

40.5.4. Getting the Weights

```
from spotpython.plot.xai import sort_layers
weights, _ = get_weights(model)
# sort_layers(weights)
```

```
visualize_weights(model, absolute=True, cmap="GreenYellowRed", figsize=(6, 6))
```

```
3200 values in Layer Layer 0. Geometry: (320, 10)
```

```
51200 values in Layer Layer 3. Geometry: (160, 320)
```

```
51200 values in Layer Layer 6. Geometry: (320, 160)
```

```
51200 values in Layer Layer 9. Geometry: (160, 320)
```

```
25600 values in Layer Layer 12. Geometry: (160, 160)
```

```
12800 values in Layer Layer 15. Geometry: (80, 160)
```

```
6400 values in Layer Layer 18. Geometry: (80, 80)
```

```
80 values in Layer Layer 21. Geometry: (1, 80)
```

```
visualize_weights_distributions(model, color=f"C{0}", columns=4)
```

```
n:8
```

40.5.5. Getting the Activations

```
from spotpython.plot.xai import get_activations
activations, mean_activations, layer_sizes = get_activations(net=model, fun_control=fun_control, bat
```

```
train_size: 0.36, val_size: 0.24, test_sie: 0.4 for splitting train & val data.
train samples: 160, val samples: 106 generated for train & val data.
LightDataModule.train_dataloader(). data_train size: 160
```

```
visualize_mean_activations(mean_activations, layer_sizes=layer_sizes, absolute=True, cmap="GreenYell
```

```
320 values in Layer 0. Geometry: (1, 320)
```

```
160 values in Layer 3. Geometry: (1, 160)
```

```
320 values in Layer 6. Geometry: (1, 320)
```

```
160 values in Layer 9. Geometry: (1, 160)
```

```
160 values in Layer 12. Geometry: (1, 160)
```

```
80 values in Layer 15. Geometry: (1, 80)
```

```
80 values in Layer 18. Geometry: (1, 80)
```

```
visualize_activations_distributions(activations=activations,
                                    net=model, color="C0", columns=4)
```

40.5.6. Getting the Gradients

```
gradients, _ = get_gradients(net=model, fun_control=fun_control, batch_size=batch_size
```

```
train_size: 0.36, val_size: 0.24, test_sie: 0.4 for splitting train & val data.
train samples: 160, val samples: 106 generated for train & val data.
LightDataModule.train_dataloader(). data_train size: 160
```

```
visualize_gradients(model, fun_control, batch_size, absolute=True, cmap="GreenYellowRed", figsize=(6
```

```
train_size: 0.36, val_size: 0.24, test_sie: 0.4 for splitting train & val data.
train samples: 160, val samples: 106 generated for train & val data.
LightDataModule.train_dataloader(). data_train size: 160
3200 values in Layer layers.0.weight. Geometry: (320, 10)
```

```
51200 values in Layer layers.3.weight. Geometry: (160, 320)
```

```
51200 values in Layer layers.6.weight. Geometry: (320, 160)
```

```
51200 values in Layer layers.9.weight. Geometry: (160, 320)
```

```
25600 values in Layer layers.12.weight. Geometry: (160, 160)
```

```
12800 values in Layer layers.15.weight. Geometry: (80, 160)
```

```
6400 values in Layer layers.18.weight. Geometry: (80, 80)
```

```
80 values in Layer layers.21.weight. Geometry: (1, 80)
```

```
visualize_gradient_distributions(model, fun_control, batch_size=batch_size, color=f"C-
```

```
train_size: 0.36, val_size: 0.24, test_sie: 0.4 for splitting train & val data.
train samples: 160, val samples: 106 generated for train & val data.
LightDataModule.train_dataloader(). data_train size: 160
n:8
```

40.6. Feature Attributions

40.6.1. Integrated Gradients

```
from spotpython.plot.xai import get_attributions, plot_attributions
df_att = get_attributions(S, fun_control, attr_method="IntegratedGradients", n_rel=10)
plot_attributions(df_att, attr_method="IntegratedGradients")
```

```
train_model result: {'val_loss': 2979.10546875, 'hp_metric': 2979.10546875}
config: {'l1': 16, 'epochs': 4096, 'batch_size': 32, 'act_fn': ReLU(), 'optimizer': 'Adam', 'dropout
Loading model with 16_4096_32_ReLU_Adam_0.0035_5.0908_4_False_kaiming_uniform_TRAIN from runs/saved_
Model: NNLinearRegressor(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=320, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.0035182324401420283, inplace=False)
    (3): Linear(in_features=320, out_features=160, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.0035182324401420283, inplace=False)
    (6): Linear(in_features=160, out_features=320, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.0035182324401420283, inplace=False)
    (9): Linear(in_features=320, out_features=160, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.0035182324401420283, inplace=False)
    (12): Linear(in_features=160, out_features=160, bias=True)
    (13): ReLU()
    (14): Dropout(p=0.0035182324401420283, inplace=False)
    (15): Linear(in_features=160, out_features=80, bias=True)
    (16): ReLU()
    (17): Dropout(p=0.0035182324401420283, inplace=False)
    (18): Linear(in_features=80, out_features=80, bias=True)
    (19): ReLU()
    (20): Dropout(p=0.0035182324401420283, inplace=False)
    (21): Linear(in_features=80, out_features=1, bias=True)
  )
)
train_size: 0.36, val_size: 0.24, test_sie: 0.4 for splitting test data.
test samples: 177 generated for test data.
LightDataModule.test_dataloader(). Test set size: 177
```

40.6.2. Deep Lift

```
df_lift = get_attributions(S, fun_control, attr_method="DeepLift",n_rel=10)
print(df_lift)
plot_attributions(df_lift,  attr_method="DeepLift")
```

```
train_model result: {'val_loss': 3620.978515625, 'hp_metric': 3620.978515625}
config: {'l1': 16, 'epochs': 4096, 'batch_size': 32, 'act_fn': ReLU(), 'optimizer': 'A
Loading model with 16_4096_32_ReLU_Adam_0.0035_5.0908_4_False_kaiming_uniform_TRAIN fr
Model: NNLinearRegressor(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=320, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.0035182324401420283, inplace=False)
    (3): Linear(in_features=320, out_features=160, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.0035182324401420283, inplace=False)
    (6): Linear(in_features=160, out_features=320, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.0035182324401420283, inplace=False)
    (9): Linear(in_features=320, out_features=160, bias=True)
```

```
    (10): ReLU()
    (11): Dropout(p=0.0035182324401420283, inplace=False)
    (12): Linear(in_features=160, out_features=160, bias=True)
    (13): ReLU()
    (14): Dropout(p=0.0035182324401420283, inplace=False)
    (15): Linear(in_features=160, out_features=80, bias=True)
    (16): ReLU()
    (17): Dropout(p=0.0035182324401420283, inplace=False)
    (18): Linear(in_features=80, out_features=80, bias=True)
    (19): ReLU()
    (20): Dropout(p=0.0035182324401420283, inplace=False)
    (21): Linear(in_features=80, out_features=1, bias=True)
  )
)
train_size: 0.36, val_size: 0.24, test_sie: 0.4 for splitting test data.
test samples: 177 generated for test data.
LightDataModule.test_dataloader(). Test set size: 177
   Feature Index Feature  DeepLiftAttribution
0              3      bp           967.263550
1              1     sex           944.123291
2              0     age           942.902954
3              2     bmi           840.350220
4              9  s6_glu           683.232361
5              8  s5_ltg           631.670593
6              6  s3_hdl           566.313477
7              4   s1_tc           517.500244
8              7  s4_tch           404.392334
9              5  s2_ldl           324.468750
```

40.6.3. Feature Ablation

```
df_fl = get_attributions(S, fun_control, attr_method="FeatureAblation",n_rel=10)
```

```
train_model result: {'val_loss': 3190.41455078125, 'hp_metric': 3190.41455078125}
config: {'l1': 16, 'epochs': 4096, 'batch_size': 32, 'act_fn': ReLU(), 'optimizer': 'A
Loading model with 16_4096_32_ReLU_Adam_0.0035_5.0908_4_False_kaiming_uniform_TRAIN fr
Model: NNLinearRegressor(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=320, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.0035182324401420283, inplace=False)
    (3): Linear(in_features=320, out_features=160, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.0035182324401420283, inplace=False)
    (6): Linear(in_features=160, out_features=320, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.0035182324401420283, inplace=False)
    (9): Linear(in_features=320, out_features=160, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.0035182324401420283, inplace=False)
    (12): Linear(in_features=160, out_features=160, bias=True)
```

```
    (13): ReLU()
    (14): Dropout(p=0.0035182324401420283, inplace=False)
    (15): Linear(in_features=160, out_features=80, bias=True)
    (16): ReLU()
    (17): Dropout(p=0.0035182324401420283, inplace=False)
    (18): Linear(in_features=80, out_features=80, bias=True)
    (19): ReLU()
    (20): Dropout(p=0.0035182324401420283, inplace=False)
    (21): Linear(in_features=80, out_features=1, bias=True)
  )
)
train_size: 0.36, val_size: 0.24, test_sie: 0.4 for splitting test data.
test samples: 177 generated for test data.
LightDataModule.test_dataloader(). Test set size: 177
```

```
print(df_fl)
plot_attributions(df_fl, attr_method="FeatureAblation")
```

```
   Feature Index Feature  FeatureAblationAttribution
0              3      bp                  601.256042
1              0     age                  566.902100
2              1     sex                  475.893738
3              2     bmi                  446.341858
4              9  s6_glu                  372.990051
5              6  s3_hdl                  285.713867
6              8  s5_ltg                  281.033508
7              4   s1_tc                  173.454834
8              7  s4_tch                   74.899757
9              5  s2_ldl                   65.023361
```

40.7. Conductance

```
from spotpython.plot.xai import plot_conductance_last_layer, get_weights_conductance_l
weights_last, layer_conductance_last = get_weights_conductance_last_layer(S, fun_contr
plot_conductance_last_layer(weights_last, layer_conductance_last, figsize=(6, 6))
```

```
train_model result: {'val_loss': 5097.568359375, 'hp_metric': 5097.568359375}
config: {'l1': 16, 'epochs': 4096, 'batch_size': 32, 'act_fn': ReLU(), 'optimizer': 'A
Loading model with 16_4096_32_ReLU_Adam_0.0035_5.0908_4_False_kaiming_uniform_TRAIN fr
Model: NNLinearRegressor(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=320, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.0035182324401420283, inplace=False)
    (3): Linear(in_features=320, out_features=160, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.0035182324401420283, inplace=False)
    (6): Linear(in_features=160, out_features=320, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.0035182324401420283, inplace=False)
    (9): Linear(in_features=320, out_features=160, bias=True)
```

```
    (10): ReLU()
    (11): Dropout(p=0.0035182324401420283, inplace=False)
    (12): Linear(in_features=160, out_features=160, bias=True)
    (13): ReLU()
    (14): Dropout(p=0.0035182324401420283, inplace=False)
    (15): Linear(in_features=160, out_features=80, bias=True)
    (16): ReLU()
    (17): Dropout(p=0.0035182324401420283, inplace=False)
    (18): Linear(in_features=80, out_features=80, bias=True)
    (19): ReLU()
    (20): Dropout(p=0.0035182324401420283, inplace=False)
    (21): Linear(in_features=80, out_features=1, bias=True)
  )
)
train_model result: {'val_loss': 3062.875, 'hp_metric': 3062.875}
config: {'l1': 16, 'epochs': 4096, 'batch_size': 32, 'act_fn': ReLU(), 'optimizer': 'Adam', 'dropout
Loading model with 16_4096_32_ReLU_Adam_0.0035_5.0908_4_False_kaiming_uniform_TRAIN from runs/saved_
Model: NNLinearRegressor(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=320, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.0035182324401420283, inplace=False)
    (3): Linear(in_features=320, out_features=160, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.0035182324401420283, inplace=False)
    (6): Linear(in_features=160, out_features=320, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.0035182324401420283, inplace=False)
    (9): Linear(in_features=320, out_features=160, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.0035182324401420283, inplace=False)
    (12): Linear(in_features=160, out_features=160, bias=True)
    (13): ReLU()
    (14): Dropout(p=0.0035182324401420283, inplace=False)
    (15): Linear(in_features=160, out_features=80, bias=True)
    (16): ReLU()
    (17): Dropout(p=0.0035182324401420283, inplace=False)
    (18): Linear(in_features=80, out_features=80, bias=True)
    (19): ReLU()
    (20): Dropout(p=0.0035182324401420283, inplace=False)
    (21): Linear(in_features=80, out_features=1, bias=True)
  )
)
Conductance analysis for layer:  Linear(in_features=80, out_features=1, bias=True)
```

Layer Conductance vs. Weights
Normalized Value
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
10
20
30
40
50
60
70
80
Weight Index
Weights
Layer Conductance

41. HPT PyTorch Lightning Transformer: Introduction

In this chapter, we will introduce transformer. The transformer architecture is a neural network architecture that is based on the attention mechanism (Vaswani et al. 2017). It is particularly well suited for sequence-to-sequence tasks, such as machine translation, text summarization, and more. The transformer architecture has been a breakthrough in the field of natural language processing (NLP) and has been the basis for many state-of-the-art models in the field.

We start with a description of the transformer basics in Section 41.1. Section 41.2 provides a detailed description of the implementation of the transformer architecture. Finally, an example of a transformer implemented in PyTorch Lightning in presented in Section 41.3.

41.1. Transformer Basics

41.1.1. Embedding

Word embedding is a technique where words or phrases (so-called tokens) from the vocabulary are mapped to vectors of real numbers. These vectors capture the semantic properties of the words. Words that are similar in meaning are mapped to vectors that are close to each other in the vector space, and words that are dissimilar are mapped to vectors that are far apart. Word embeddings are needed for transformers for several reasons:

- Dimensionality Reduction: Word embeddings reduce the dimensionality of the data. Instead of dealing with high-dimensional sparse vectors (like one-hot encoded vectors), we deal with dense vectors of much lower dimensionality.
- Capturing Semantic Similarities: Word embeddings capture semantic similarities between words. This is crucial for tasks like text classification, sentiment analysis, etc., where the meaning of the words is important.
- Handling Unknown Words: If a word is not present in the training data but appears in the test data, one-hot encoding cannot handle it. But word embeddings can handle such situations by mapping the unknown word to a vector that is similar to known words.

- Input to Neural Networks: Transformers, like other neural networks, work with numerical data. Word embeddings provide a way to convert text data into numerical form that can be fed into these networks.

In the context of transformers, word embeddings are used as the initial input representation. The transformer then learns more complex representations by considering the context in which each token appears.

41.1.1.1. Neural Network for Embeddings

Idea for word embeddings: use a relatively simple NN that has one input for every token (word, symbol) in the vocabulary. The output of the NN is a vector of a fixed size, which is the word embedding. The network that is used in this chapter is visualized in Figure 41.1. For simplicity, a 2-dimensional output vector is used in this visualization. The weights of the NN are randomly initialized, and are learned during training.

All tokens are embedded in this way. For each token there are two numerical values, the embedding vector. The same network is used for embedding all tokens. If a longer input is added, it can be embedded with the same net.

41.1.1.2. Positional Encoding for the Embeddings

Positional encoding is added to the input embeddings to give the model some information about the relative or absolute position of the tokens in the sequence. The positional encodings have the same dimension as the embeddings so that the two can be summed.

If a token occurs several times, it is embedded several times and receives different embedding vectors, as the position is taken into account by the positional encoding.

41.1.2. Attention

Attention describes how similar is each token to itself and to all other tokens in the input, e.g., in a sentence. The attention mechanism can be implemented as a set of layers in neural networks. There are a lot of different possible definitions of "attention" in the literature, but the one we will use here is the following: *the attention mechanism describes a weighted average of (sequence) elements with the weights dynamically computed based on an input query and elements' keys* (Lippe 2022).

The goal is to take an average over the features of multiple elements. However, instead of weighting each element equally, we want to weight them depending on their actual values. In other words, we want to dynamically decide on which inputs we want to "attend" more than others.

Calculation of the self-attention:

Figure 41.1.: Transformer. Computation of the self attention. In this example, we consider two inputs, i.e., (1,0) and (0,1). For each input, there are two values, which results in a 2×2 matrix. In general, when there are T inputs, a $T \times T$ matrix will be generated. Figure credits: Starmer, Josh: Decoder-Only Transformers, ChatGPTs specific Transformer, Clearly Explained.

1. Queries: Calculate two new values from the (two) values of the embedding vector using an NN, which are referred to as query values.
2. Keys: Calculate two new values, called key values, from the (two) values of the embedding vector using an NN.
3. Dot product: Calculate the dot product of the query values and the key values. This is a measure of the similarity of the query and key values.
4. Softmax: Apply the softmax function to the outputs from the dot product. This is a measure of the attention that a token pays to other tokens.
5. Values: Calculate two new values from the (two) values of the embedding vector using an NN, which are referred to as value values.
6. The values are multiplied (weighted) by the values of the softmax function.
7. The weighted values are summed. Now we have the self attention value for the token.

41.1.3. Self-Attention

Most attention mechanisms differ in terms of what queries they use, how the key and value vectors are defined, and what score function is used. The attention applied inside the Transformer architecture is called "self-attention". In self-attention, each sequence element provides a key, value, and query. For each element, we perform an attention layer where based on its query, we check the similarity of the all sequence elements' keys, and returned a different, averaged value vector for each element.

41.1.4. Masked Self-Attention

Masked self-attention is a variant of the self-attention method described in Section 41.1.3. It asks the question: How similar is each token to itself and to all preceding tokens in the input (sentence)? Masked self-attention is an autoregressive mechanism, which means that the attention mechanism is only allowed to look at the tokens that have already been processed. Calculation of the mask self-attention is identical to the self-attention, but the attention is only calculated for the tokens that have already been processed. If the masked self-attention method is applied to the first token, the masked self-attention value is exactly the value of the first token, as it only takes itself into account. For the other tokens, the masked self-attention value is a weighted sum of the values of the previous tokens. The weighting is determined by the similarity of the query values and the key values (dot product and softmax).

41.1.5. Generation of Outputs

To calculate the output, we use a residual connector that adds the output of the neural network and the output of the masked self-attention method. We thus obtain the residual connection values. The residual connector is used to facilitate training.

To generate the next token, we use another neural network that calculates the output from the (two) residual connection values. The input layer of the neural network has the size of the residual connection values, the output layer has the number of tokens in the vocabulary as a dimension.

If we now enter the residual connection value of the first token, we receive the token (or the probabilities using Softmax) that is to come next as the output of the neural network. This makes sense even if we already know the second token (as with the first token): We can use it to calculate the error of the neural network and train the network. In addition, the decoder-transformer uses the masked self-attention method to calculate the output, i.e. the encoding and generation of new tokens is done with exactly the same elements of the network.

Note: ChatGPT does not use a new neural network, but the same network that was already used to calculate the embedding. The network is therefore used for embedding, masked self-attention and calculating the output. In the last calculation, the network is inverted, i.e. it is run in the opposite direction to obtain the tokens and not the embeddings as in the original run.

41.1.6. End-Of-Sequence-Token

The end-of-sequence token is used to signal the end of the input and also to start generating new tokens after the input. The EOS token recognizes all other tokens, as it comes after all tokens. When generating tokens, it is important to consider the relationships between the input tokens and the generation of new tokens.

41.2. Details of the Implementation

We will now go into a bit more detail by first looking at the specific implementation of the attention mechanism which is in the Transformer case the (scaled) dot product attention. The variables shown in Table 41.1 are used in the Transformer architecture.

Table 41.1.: Variables used in the Transformer architecture.

Symbol	Variable	Description
Q	`query`	The query vectors.
K	`key`	The key vectors.
V	`value`	The value vectors.
d_{model}	`d_model`	The dimensionality of the input and output features of the Transformer.

Symbol	Variable	Description
d_k	`d_k`	The hidden dimensionality of the key and query vectors.
d_v	`d_v`	The hidden dimensionality of the value vectors.
h	`num_heads`	The number of heads in the Multi-Head Attention layer.
B	`batch_size`	The batch size.
T	`seq_length`	The sequence length.
X	`x`	The input features (input elements in the sequence).
W^Q	`qkv_proj`	The weight matrix to transform the input to the query vectors.
W^K	`qkv_proj`	The weight matrix to transform the input to the key vectors.
W^V	`qkv_proj`	The weight matrix to transform the input to the value vectors.
W^O	`o_proj`	The weight matrix to transform the concatenated output of the Multi-Head Attention layer to the final output.
N	`num_layers`	The number of layers in the Transformer.
$PE_{(pos,i)}$	`positional_encoding`	The positional encoding for position pos and hidden dimensionality i.

Summarizing the ideas from Section 41.1, an attention mechanism has usually four parts we need to specify (Lippe 2022):

- *Query*: The query is a feature vector that describes what we are looking for in the sequence, i.e., what would we maybe want to pay attention to.
- *Keys*: For each input element, we have a key which is again a feature vector. This feature vector roughly describes what the element is "offering", or when it might be important. The keys should be designed such that we can identify the elements we want to pay attention to based on the query.
- *Score function*: To rate which elements we want to pay attention to, we need to specify a score function f_{attn}. The score function takes the query and a key as input, and output the score/attention weight of the query-key pair. It is usually implemented by simple similarity metrics like a dot product, or a small MLP.
- *Values*: For each input element, we also have a value vector. This feature vector is the one we want to average over.

The weights of the average are calculated by a softmax over all score function outputs. Hence, we assign those value vectors a higher weight whose corresponding key is most similar to the query. If we try to describe it with pseudo-math, we can write:

$$\alpha_i = \frac{\exp\left(f_{attn}\left(\text{key}_i, \text{query}\right)\right)}{\sum_j \exp\left(f_{attn}\left(\text{key}_j, \text{query}\right)\right)}, \qquad \text{out} = \sum_i \alpha_i \cdot \text{value}_i$$

Visually, we can show the attention over a sequence of words as follows:

Figure 41.2.: Attention over a sequence of words. For every word, we have one key and one value vector. The query is compared to all keys with a score function (in this case the dot product) to determine the weights. The softmax is not visualized for simplicity. Finally, the value vectors of all words are averaged using the attention weights. Figure taken from Lippe (2022)

41.2.1. Dot Product Attention

Our goal is to have an attention mechanism with which any element in a sequence can attend to any other while still being efficient to compute. The dot product attention takes as input a set of queries $Q \in \mathbb{R}^{T \times d_k}$, keys $K \in \mathbb{R}^{T \times d_k}$ and values $V \in \mathbb{R}^{T \times d_v}$ where T is the sequence length, and d_k and d_v are the hidden dimensionality for queries/keys and values respectively. For simplicity, we neglect the batch dimension for now. The attention value from element i to j is based on its similarity of the query Q_i and key K_j, using the dot product as the similarity metric (in Figure 41.1, we considered Q_2 and K_1 as well as Q_2 and K_2). The dot product attention is calculated as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(QK^T\right)V \tag{41.1}$$

The matrix multiplication QK^T performs the dot product for every possible pair of queries and keys, resulting in a matrix of the shape $T \times T$. Each row represents the attention logits for a specific element i to all other elements in the sequence. On these, we apply a softmax and multiply with the value vector to obtain a weighted mean (the weights being determined by the attention).

41.2.2. Scaled Dot Product Attention

An additional aspect is the scaling of the dot product using a scaling factor of $1/\sqrt{d_k}$. This scaling factor is crucial to maintain an appropriate variance of attention values after initialization. We initialize our layers with the intention of having equal variance throughout the model, and hence, Q and K might also have a variance close to 1. However, performing a dot product over two vectors with a variance σ^2 results in a scalar having d_k-times higher variance:

$$q_i \sim \mathcal{N}(0, \sigma^2), k_i \sim \mathcal{N}(0, \sigma^2) \rightarrow \text{Var}\left(\sum_{i=1}^{d_k} q_i \cdot k_i\right) = \sigma^4 \cdot d_k$$

If we do not scale down the variance back to $\sim \sigma^2$, the softmax over the logits will already saturate to 1 for one random element and 0 for all others. The gradients through the softmax will be close to zero so that we can't learn the parameters appropriately. Note that the extra factor of σ^2, i.e., having σ^4 instead of σ^2, is usually not an issue, since we keep the original variance σ^2 close to 1 anyways. Equation 41.1 can be modified as follows to calculate the dot product attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V.$$

Another perspective on this scaled dot product attention mechanism offers the computation graph which is visualized in Figure 41.3.

Figure 41.3.: Scaled dot product attention. Figure credit Vaswani et al. (2017)

The block `Mask (opt.)` in the diagram above represents the optional masking of specific entries in the attention matrix. This is for instance used if we stack multiple

sequences with different lengths into a batch. To still benefit from parallelization in PyTorch, we pad the sentences to the same length and mask out the padding tokens during the calculation of the attention values. This is usually done by setting the respective attention logits to a very low value.

After we have discussed the details of the scaled dot product attention block, we can write a function below which computes the output features given the triple of queries, keys, and values:

41.3. Example: Transformer in Lightning

The following code is based on https://github.com/phlippe/uvadlc_notebooks/tree/master (Author: Phillip Lippe)

First, we import the necessary libraries and download the pretrained models.

```
import os
import numpy as np
import random
import math
import json
from functools import partial
import matplotlib.pyplot as plt
from matplotlib.colors import to_rgb
import matplotlib
import seaborn as sns

## tqdm for loading bars
from tqdm.notebook import tqdm

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim

# PyTorch Lightning
import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint
```

```
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial6"
```

```
# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

```
from spotpython.utils.device import getDevice
device = getDevice()
print("Device:", device)
```

```
Device: mps
```

```
# Setting the seed
pl.seed_everything(42)
```

```
42
```

Two pre-trained models are downloaded below. Make sure to have adjusted your `CHECKPOINT_PATH` before running this code if not already done.

```
import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial6/"
# Files to download
pretrained_files = ["ReverseTask.ckpt", "SetAnomalyTask.ckpt"]

# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)
```

41.3.1. Downloading the Pretrained Models

```
# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "/" in file_name:
        os.makedirs(file_path.rsplit("/",1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Error:\n", e)
```

41.3.2. The Transformer Architecture

We will implement the Transformer architecture by hand. As the architecture is so popular, there already exists a Pytorch module `nn.Transformer` (documentation) and a tutorial on how to use it for next token prediction. However, we will implement it here ourselves, to get through to the smallest details.

41.3.3. Attention Mechanism

```
def scaled_dot_product(q, k, v, mask=None):
    """
    Compute scaled dot product attention.
    Args:
        q: Queries
        k: Keys
        v: Values
        mask: Mask to apply to the attention logits

    Returns:
        Tuple of (Values, Attention weights)

    Examples:
    >>> seq_len, d_k = 1, 2
        pl.seed_everything(42)
        q = torch.randn(seq_len, d_k)
        k = torch.randn(seq_len, d_k)
        v = torch.randn(seq_len, d_k)
        values, attention = scaled_dot_product(q, k, v)
        print("Q\n", q)
        print("K\n", k)
        print("V\n", v)
        print("Values\n", values)
        print("Attention\n", attention)
    """
    d_k = q.size()[-1]
    attn_logits = torch.matmul(q, k.transpose(-2, -1))
    attn_logits = attn_logits / math.sqrt(d_k)
    if mask is not None:
        attn_logits = attn_logits.masked_fill(mask == 0, -9e15)
    attention = F.softmax(attn_logits, dim=-1)
    values = torch.matmul(attention, v)
    return values, attention
```

Note that our code above supports any additional dimensionality in front of the sequence length so that we can also use it for batches. However, for a better understanding, let's generate a few random queries, keys, and value vectors, and calculate the attention outputs:

```
seq_len, d_k = 1, 2
pl.seed_everything(42)
q = torch.randn(seq_len, d_k)
k = torch.randn(seq_len, d_k)
v = torch.randn(seq_len, d_k)
values, attention = scaled_dot_product(q, k, v)
print("Q\n", q)
print("K\n", k)
print("V\n", v)
print("Values\n", values)
print("Attention\n", attention)
```

```
Q
 tensor([[0.3367, 0.1288]])
K
 tensor([[0.2345, 0.2303]])
V
 tensor([[-1.1229, -0.1863]])
Values
 tensor([[-1.1229, -0.1863]])
Attention
 tensor([[1.]])
```

41.3.4. Multi-Head Attention

The scaled dot product attention allows a network to attend over a sequence. However, often there are multiple different aspects a sequence element wants to attend to, and a single weighted average is not a good option for it. This is why we extend the attention mechanisms to multiple heads, i.e. multiple different query-key-value triplets on the same features. Specifically, given a query, key, and value matrix, we transform those into h sub-queries, sub-keys, and sub-values, which we pass through the scaled dot product attention independently. Afterward, we concatenate the heads and combine them with a final weight matrix. Mathematically, we can express this operation as:

$$\begin{aligned}\text{Multihead}(Q, K, V) &= \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O \\ \text{where head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)\end{aligned}$$

We refer to this as Multi-Head Attention layer with the learnable parameters $W^Q_{1\ldots h} \in \mathbb{R}^{D\times d_k}$, $W^K_{1\ldots h} \in \mathbb{R}^{D\times d_k}$, $W^V_{1\ldots h} \in \mathbb{R}^{D\times d_v}$, and $W^O \in \mathbb{R}^{h\cdot d_v\times d_{out}}$ (D being the input dimensionality). Expressed in a computational graph, we can visualize it as in Figure 41.4.

Figure 41.4.: Multi-Head Attention. Figure taken from Vaswani et al. (2017)

How are we applying a Multi-Head Attention layer in a neural network, where we do not have an arbitrary query, key, and value vector as input? Looking at the computation graph in Figure 41.4, a simple but effective implementation is to set the current feature map in a NN, $X \in \mathbb{R}^{B\times T\times d_{\text{model}}}$, as Q, K and V (B being the batch size, T the sequence length, d_{model} the hidden dimensionality of X). The consecutive weight matrices W^Q, W^K, and W^V can transform X to the corresponding feature vectors that represent the queries, keys, and values of the input. Using this approach, we can implement the Multi-Head Attention module below.

As a consequence, if the embedding dimension is 4, then 1, 2 or 4 heads can be used, but not 3. If 4 heads are used, then the dimension of the query, key and value vectors is 1. If 2 heads are used, then the dimension of the query, key and value vectors is $D = 2$. If 1 head is used, then the dimension of the query, key and value vectors is $D = 4$. The number of heads is a hyperparameter that can be adjusted. The number of heads is usually 8 or 16.

```
# Helper function to support different mask shapes.
# Output shape supports (batch_size, number of heads, seq length, seq length)
# If 2D: broadcasted over batch size and number of heads
# If 3D: broadcasted over number of heads
# If 4D: leave as is
def expand_mask(mask):
    assert mask.ndim >= 2, "Mask must be >= 2-dim. with seq_length x seq_length"
```

```
    if mask.ndim == 3:
        mask = mask.unsqueeze(1)
    while mask.ndim < 4:
        mask = mask.unsqueeze(0)
    return mask
```

```
class MultiheadAttention(nn.Module):

    def __init__(self, input_dim, embed_dim, num_heads):
        super().__init__()
        assert embed_dim % num_heads == 0, "Embedding dim. must be 0 modulo number of

        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads

        # Stack all weight matrices 1...h together for efficiency
        # Note that in many implementations you see "bias=False" which is optional
        self.qkv_proj = nn.Linear(input_dim, 3*embed_dim)
        self.o_proj = nn.Linear(embed_dim, embed_dim)

        self._reset_parameters()

    def _reset_parameters(self):
        # Original Transformer initialization, see PyTorch documentation
        nn.init.xavier_uniform_(self.qkv_proj.weight)
        self.qkv_proj.bias.data.fill_(0)
        nn.init.xavier_uniform_(self.o_proj.weight)
        self.o_proj.bias.data.fill_(0)

    def forward(self, x, mask=None, return_attention=False):
        batch_size, seq_length, _ = x.size()
        if mask is not None:
            mask = expand_mask(mask)
        qkv = self.qkv_proj(x)

        # Separate Q, K, V from linear output
        qkv = qkv.reshape(batch_size, seq_length, self.num_heads, 3*self.head_dim)
        qkv = qkv.permute(0, 2, 1, 3) # [Batch, Head, SeqLen, Dims]
        q, k, v = qkv.chunk(3, dim=-1)

        # Determine value outputs
        values, attention = scaled_dot_product(q, k, v, mask=mask)
        values = values.permute(0, 2, 1, 3) # [Batch, SeqLen, Head, Dims]
```

```
        values = values.reshape(batch_size, seq_length, self.embed_dim)
        o = self.o_proj(values)

        if return_attention:
            return o, attention
        else:
            return o
```

41.3.5. Permutation Equivariance

One crucial characteristic of the multi-head attention is that it is permutation-equivariant with respect to its inputs. This means that if we switch two input elements in the sequence, e.g. $X_1 \leftrightarrow X_2$ (neglecting the batch dimension for now), the output is exactly the same besides the elements 1 and 2 switched. Hence, the multi-head attention is actually looking at the input not as a sequence, but as a set of elements. This property makes the multi-head attention block and the Transformer architecture so powerful and widely applicable! But what if the order of the input is actually important for solving the task, like language modeling? The answer is to encode the position in the input features, which we will take a closer look in Section 41.3.8.

41.3.6. Transformer Encoder

Next, we will look at how to apply the multi-head attention block inside the Transformer architecture. Originally, the Transformer model was designed for machine translation. Hence, it got an encoder-decoder structure where the encoder takes as input the sentence in the original language and generates an attention-based representation. On the other hand, the decoder attends over the encoded information and generates the translated sentence in an autoregressive manner, as in a standard RNN. While this structure is extremely useful for Sequence-to-Sequence tasks with the necessity of autoregressive decoding, we will focus here on the encoder part. Many advances in NLP have been made using pure encoder-based Transformer models (if interested, models include the BERT-family (Devlin et al. 2018), the Vision Transformer (Dosovitskiy et al. 2020), and more). We will also mainly focus on the encoder part. If you have understood the encoder architecture, the decoder is a very small step to implement as well. The full Transformer architecture looks as shown in Figure 41.5.

The encoder consists of N identical blocks that are applied in sequence. Taking as input x, it is first passed through a Multi-Head Attention block as we have implemented above. The output is added to the original input using a residual connection, and we apply a consecutive Layer Normalization on the sum. Overall, it calculates

$$\text{LayerNorm}(x + \text{Multihead}(x, x, x))$$

Figure 41.5.: Transformer architecture. Figure credit: Vaswani et al. (2017)

(x being Q, K and V input to the attention layer). The residual connection is crucial in the Transformer architecture for two reasons:

1. Similar to ResNets, Transformers are designed to be very deep. Some models contain more than 24 blocks in the encoder. Hence, the residual connections are crucial for enabling a smooth gradient flow through the model.
2. Without the residual connection, the information about the original sequence is lost. Remember that the Multi-Head Attention layer ignores the position of elements in a sequence, and can only learn it based on the input features. Removing the residual connections would mean that this information is lost after the first attention layer (after initialization), and with a randomly initialized query and key vector, the output vectors for position i has no relation to its original input. All outputs of the attention are likely to represent similar/same information, and there is no chance for the model to distinguish which information came from which input element. An alternative option to residual connection would be to fix at least one head to focus on its original input, but this is very inefficient and does not have the benefit of the improved gradient flow.

41.3.7. Layer Normalization and Feed-Forward Network

The Layer Normalization also plays an important role in the Transformer architecture as it enables faster training and provides small regularization. Additionally, it ensures that the features are in a similar magnitude among the elements in the sequence.

We are not using Batch Normalization because it depends on the batch size which is often small with Transformers (they require a lot of GPU memory), and BatchNorm has shown to perform particularly bad in language as the features of words tend to have a much higher variance (there are many, very rare words which need to be considered for a good distribution estimate).

Additionally to the Multi-Head Attention, a small fully connected feed-forward network is added to the model, which is applied to each position separately and identically. Specifically, the model uses a Linear→ReLU→Linear MLP. The full transformation including the residual connection can be expressed as:

$$\begin{aligned} \mathrm{FFN}(x) &= \max(0, xW_1 + b_1)W_2 + b_2 \\ x &= \mathrm{LayerNorm}(x + \mathrm{FFN}(x)) \end{aligned}$$

This MLP adds extra complexity to the model and allows transformations on each sequence element separately. You can imagine as this allows the model to "post-process" the new information added by the previous Multi-Head Attention, and prepare it for the next attention block. Usually, the inner dimensionality of the MLP is 2-8× larger than d_{model}, i.e. the dimensionality of the original input x. The general advantage of

a wider layer instead of a narrow, multi-layer MLP is the faster, parallelizable execution.

Finally, after looking at all parts of the encoder architecture, we can start implementing it below. We first start by implementing a single encoder block. Additionally to the layers described above, we will add dropout layers in the MLP and on the output of the MLP and Multi-Head Attention for regularization.

```
class EncoderBlock(nn.Module):

    def __init__(self, input_dim, num_heads, dim_feedforward, dropout=0.0):
        """
        Inputs:
            input_dim - Dimensionality of the input
            num_heads - Number of heads to use in the attention block
            dim_feedforward - Dimensionality of the hidden layer in the MLP
            dropout - Dropout probability to use in the dropout layers
        """
        super().__init__()

        # Attention layer
        self.self_attn = MultiheadAttention(input_dim, input_dim, num_heads)

        # Two-layer MLP
        self.linear_net = nn.Sequential(
            nn.Linear(input_dim, dim_feedforward),
            nn.Dropout(dropout),
            nn.ReLU(inplace=True),
            nn.Linear(dim_feedforward, input_dim)
        )

        # Layers to apply in between the main layers
        self.norm1 = nn.LayerNorm(input_dim)
        self.norm2 = nn.LayerNorm(input_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Attention part
        attn_out = self.self_attn(x, mask=mask)
        x = x + self.dropout(attn_out)
        x = self.norm1(x)

        # MLP part
        linear_out = self.linear_net(x)
        x = x + self.dropout(linear_out)
```

```
        x = self.norm2(x)

        return x
```

Based on this block, we can implement a module for the full Transformer encoder. Additionally to a forward function that iterates through the sequence of encoder blocks, we also provide a function called `get_attention_maps`. The idea of this function is to return the attention probabilities for all Multi-Head Attention blocks in the encoder. This helps us in understanding, and in a sense, explaining the model. However, the attention probabilities should be interpreted with a grain of salt as it does not necessarily reflect the true interpretation of the model (there is a series of papers about this, including Jain and Wallace (2019) and Wiegreffe and Pinter (2019)).

```
class TransformerEncoder(nn.Module):

    def __init__(self, num_layers, **block_args):
        super().__init__()
        self.layers = nn.ModuleList(
            [EncoderBlock(**block_args) for _ in range(num_layers)])

    def forward(self, x, mask=None):
        for l in self.layers:
            x = l(x, mask=mask)
        return x

    def get_attention_maps(self, x, mask=None):
        attention_maps = []
        for l in self.layers:
            _, attn_map = l.self_attn(x, mask=mask, return_attention=True)
            attention_maps.append(attn_map)
            x = l(x)
        return attention_maps
```

41.3.8. Positional Encoding

We have discussed before that the Multi-Head Attention block is permutation-equivariant, and cannot distinguish whether an input comes before another one in the sequence or not. In tasks like language understanding, however, the position is important for interpreting the input words. The position information can therefore be added via the input features. We could learn a embedding for every possible position, but this would not generalize to a dynamical input sequence length. Hence, the better option is to use feature patterns that the network can identify from the features and

potentially generalize to larger sequences. The specific pattern chosen by Vaswani et al. (2017) are sine and cosine functions of different frequencies, as follows:

$$PE_{(pos,i)} = \begin{cases} \sin\left(\frac{pos}{10000^{i/d_{\text{model}}}}\right) & \text{if} \quad i \bmod 2 = 0 \\ \cos\left(\frac{pos}{10000^{(i-1)/d_{\text{model}}}}\right) & \text{otherwise} \end{cases}$$

$PE_{(pos,i)}$ represents the position encoding at position pos in the sequence, and hidden dimensionality i. These values, concatenated for all hidden dimensions, are added to the original input features (in the Transformer visualization above, see "Positional encoding"), and constitute the position information. We distinguish between even ($i \bmod 2 = 0$) and uneven ($i \bmod 2 = 1$) hidden dimensionalities where we apply a sine/cosine respectively. The intuition behind this encoding is that you can represent $PE_{(pos+k,:)}$ as a linear function of $PE_{(pos,:)}$, which might allow the model to easily attend to relative positions. The wavelengths in different dimensions range from 2π to $10000 \cdot 2\pi$.

The positional encoding is implemented below. The code is taken from the PyTorch tutorial https://pytorch.org/tutorials/beginner/transformer_tutorial.html#define-the-model about Transformers on NLP and adjusted for our purposes.

```
class PositionalEncoding(nn.Module):

    def __init__(self, d_model, max_len=5000):
        """
        Inputs
            d_model - Hidden dimensionality of the input.
            max_len - Maximum length of a sequence to expect.
        """
        super().__init__()

        # Create matrix of [SeqLen, HiddenDim] representing
        # the positional encoding for max_len inputs
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0)
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)

        # register_buffer => Tensor which is not a parameter,
        # but should be part of the modules state.
        # Used for tensors that need to be on the same device as the module.
        # persistent=False tells PyTorch to not add the buffer to the
        # state dict (e.g. when we save the model)
```

```
        self.register_buffer('pe', pe, persistent=False)

    def forward(self, x):
        x = x + self.pe[:, :x.size(1)]
        return x
```

To understand the positional encoding, we can visualize it below. We will generate an image of the positional encoding over hidden dimensionality and position in a sequence. Each pixel, therefore, represents the change of the input feature we perform to encode the specific position. Let's do it below.

```
matplotlib.rcParams['lines.linewidth'] = 2.0
plt.set_cmap('cividis')
encod_block = PositionalEncoding(d_model=48, max_len=96)
pe = encod_block.pe.squeeze().T.cpu().numpy()

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(8,3))
pos = ax.imshow(pe, cmap="RdGy", extent=(1,pe.shape[1]+1,pe.shape[0]+1,1))
fig.colorbar(pos, ax=ax)
ax.set_xlabel("Position in sequence")
ax.set_ylabel("Hidden dimension")
ax.set_title("Positional encoding over hidden dimensions")
ax.set_xticks([1]+[i*10 for i in range(1,1+pe.shape[1]//10)])
ax.set_yticks([1]+[i*10 for i in range(1,1+pe.shape[0]//10)])
plt.show()
```

```
<Figure size 1650x1050 with 0 Axes>
```

You can clearly see the sine and cosine waves with different wavelengths that encode the position in the hidden dimensions. Specifically, we can look at the sine/cosine wave for each hidden dimension separately, to get a better intuition of the pattern. Below we visualize the positional encoding for the hidden dimensions 1, 2, 3 and 4.

```
sns.set_theme()
fig, ax = plt.subplots(2, 2, figsize=(12,4))
ax = [a for a_list in ax for a in a_list]
for i in range(len(ax)):
    ax[i].plot(np.arange(1,17), pe[i,:16], color=f'C{i}', marker="o",
               markersize=6, markeredgecolor="black")
    ax[i].set_title(f"Encoding in hidden dimension {i+1}")
    ax[i].set_xlabel("Position in sequence", fontsize=10)
    ax[i].set_ylabel("Positional encoding", fontsize=10)
    ax[i].set_xticks(np.arange(1,17))
    ax[i].tick_params(axis='both', which='major', labelsize=10)
    ax[i].tick_params(axis='both', which='minor', labelsize=8)
    ax[i].set_ylim(-1.2, 1.2)
fig.subplots_adjust(hspace=0.8)
sns.reset_orig()
plt.show()
```

As we can see, the patterns between the hidden dimension 1 and 2 only differ in the starting angle. The wavelength is 2π, hence the repetition after position 6. The hidden dimensions 2 and 3 have about twice the wavelength.

41.3.9. Learning Rate Warm-up

One commonly used technique for training a Transformer is learning rate warm-up. This means that we gradually increase the learning rate from 0 on to our originally

specified learning rate in the first few iterations. Thus, we slowly start learning instead of taking very large steps from the beginning. In fact, training a deep Transformer without learning rate warm-up can make the model diverge and achieve a much worse performance on training and testing. Take for instance the following plot by Liu et al. (2019) comparing Adam-vanilla (i.e. Adam without warm-up) vs Adam with a warm-up:

Figure 41.6.: Warm-up comparison. Figure taken from Liu et al. (2019)

Clearly, the warm-up is a crucial hyperparameter in the Transformer architecture. Why is it so important? There are currently two common explanations. Firstly, Adam uses the bias correction factors which however can lead to a higher variance in the adaptive learning rate during the first iterations. Improved optimizers like RAdam have been shown to overcome this issue, not requiring warm-up for training Transformers. Secondly, the iteratively applied Layer Normalization across layers can lead to very high gradients during the first iterations, which can be solved by using Pre-Layer Normalization (similar to Pre-Activation ResNet), or replacing Layer Normalization by other techniques (Adaptive Normalization, Power Normalization).

Nevertheless, many applications and papers still use the original Transformer architecture with Adam, because warm-up is a simple, yet effective way of solving the gradient problem in the first iterations. There are many different schedulers we could use. For instance, the original Transformer paper used an exponential decay scheduler with a warm-up. However, the currently most popular scheduler is the cosine warm-up scheduler, which combines warm-up with a cosine-shaped learning rate decay. We can implement it below, and visualize the learning rate factor over epochs.

```
class CosineWarmupScheduler(optim.lr_scheduler._LRScheduler):

    def __init__(self, optimizer, warmup, max_iters):
        self.warmup = warmup
        self.max_num_iters = max_iters
        super().__init__(optimizer)

    def get_lr(self):
```

```
        lr_factor = self.get_lr_factor(epoch=self.last_epoch)
        return [base_lr * lr_factor for base_lr in self.base_lrs]

    def get_lr_factor(self, epoch):
        lr_factor = 0.5 * (1 + np.cos(np.pi * epoch / self.max_num_iters))
        if epoch <= self.warmup:
            lr_factor *= epoch * 1.0 / self.warmup
        return lr_factor
```

```
# Needed for initializing the lr scheduler
p = nn.Parameter(torch.empty(4,4))
optimizer = optim.Adam([p], lr=1e-3)
lr_scheduler = CosineWarmupScheduler(optimizer=optimizer, warmup=100, max_iters=2000)

# Plotting
epochs = list(range(2000))
sns.set()
plt.figure(figsize=(8,3))
plt.plot(epochs, [lr_scheduler.get_lr_factor(e) for e in epochs])
plt.ylabel("Learning rate factor")
plt.xlabel("Iterations (in batches)")
plt.title("Cosine Warm-up Learning Rate Scheduler")
plt.show()
sns.reset_orig()
```

In the first 100 iterations, we increase the learning rate factor from 0 to 1, whereas for all later iterations, we decay it using the cosine wave. Pre-implementations of this scheduler can be found in the popular NLP Transformer library huggingface.

41.3.10. PyTorch Lightning Module

Finally, we can embed the Transformer architecture into a PyTorch lightning module. PyTorch Lightning simplifies our training and test code, as well as structures the code nicely in separate functions. We will implement a template for a classifier based on the Transformer encoder. Thereby, we have a prediction output per sequence element. If we would need a classifier over the whole sequence, the common approach is to add an additional `[CLS]` token to the sequence (`CLS` stands for classification, i.e., the first token of every sequence is always a special classification token, `CLS`). However, here we focus on tasks where we have an output per element.

Additionally to the Transformer architecture, we add a small input network (maps input dimensions to model dimensions), the positional encoding, and an output network (transforms output encodings to predictions). We also add the learning rate scheduler, which takes a step each iteration instead of once per epoch. This is needed for the warmup and the smooth cosine decay. The training, validation, and test step is left empty for now and will be filled for our task-specific models.

```
class TransformerPredictor(pl.LightningModule):

    def __init__(self, input_dim, model_dim, num_classes, num_heads, num_layers, lr, warmup, max_ite
        """
        Inputs:
            input_dim - Hidden dimensionality of the input
            model_dim - Hidden dimensionality to use inside the Transformer
            num_classes - Number of classes to predict per sequence element
            num_heads - Number of heads to use in the Multi-Head Attention blocks
            num_layers - Number of encoder blocks to use.
            lr - Learning rate in the optimizer
            warmup - Number of warmup steps. Usually between 50 and 500
            max_iters - Number of maximum iterations the model is trained for. This is needed for th
            dropout - Dropout to apply inside the model
            input_dropout - Dropout to apply on the input features
        """
        super().__init__()
        self.save_hyperparameters()
        self._create_model()

    def _create_model(self):
        # Input dim -> Model dim
        self.input_net = nn.Sequential(
            nn.Dropout(self.hparams.input_dropout),
            nn.Linear(self.hparams.input_dim, self.hparams.model_dim)
        )
        # Positional encoding for sequences
```

```
        self.positional_encoding = PositionalEncoding(d_model=self.hparams.model_dim)
        # Transformer
        self.transformer = TransformerEncoder(num_layers=self.hparams.num_layers,
                                              input_dim=self.hparams.model_dim,
                                              dim_feedforward=2*self.hparams.model_dim
                                              num_heads=self.hparams.num_heads,
                                              dropout=self.hparams.dropout)
        # Output classifier per sequence lement
        self.output_net = nn.Sequential(
            nn.Linear(self.hparams.model_dim, self.hparams.model_dim),
            nn.LayerNorm(self.hparams.model_dim),
            nn.ReLU(inplace=True),
            nn.Dropout(self.hparams.dropout),
            nn.Linear(self.hparams.model_dim, self.hparams.num_classes)
        )

    def forward(self, x, mask=None, add_positional_encoding=True):
        """
        Inputs:
            x - Input features of shape [Batch, SeqLen, input_dim]
            mask - Mask to apply on the attention outputs (optional)
            add_positional_encoding - If True, we add the positional encoding to the i
                                      Might not be desired for some tasks.
        """
        x = self.input_net(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        x = self.transformer(x, mask=mask)
        x = self.output_net(x)
        return x

    @torch.no_grad()
    def get_attention_maps(self, x, mask=None, add_positional_encoding=True):
        """
        Function for extracting the attention matrices of the whole Transformer for a
        Input arguments same as the forward pass.
        """
        x = self.input_net(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        attention_maps = self.transformer.get_attention_maps(x, mask=mask)
        return attention_maps

    def configure_optimizers(self):
```

```
        optimizer = optim.Adam(self.parameters(), lr=self.hparams.lr)

        # Apply lr scheduler per step
        lr_scheduler = CosineWarmupScheduler(optimizer,
                                             warmup=self.hparams.warmup,
                                             max_iters=self.hparams.max_iters)
        return [optimizer], [{'scheduler': lr_scheduler, 'interval': 'step'}]

    def training_step(self, batch, batch_idx):
        raise NotImplementedError

    def validation_step(self, batch, batch_idx):
        raise NotImplementedError

    def test_step(self, batch, batch_idx):
        raise NotImplementedError
```

41.4. Experiment: Sequence to Sequence

After having finished the implementation of the Transformer architecture, we can start experimenting and apply it to various tasks. We will focus on parallel Sequence-to-Sequence.

A Sequence-to-Sequence task represents a task where the input *and* the output is a sequence, not necessarily of the same length. Popular tasks in this domain include machine translation and summarization. For this, we usually have a Transformer encoder for interpreting the input sequence, and a decoder for generating the output in an autoregressive manner. Here, however, we will go back to a much simpler example task and use only the encoder. Given a sequence of N numbers between 0 and M, the task is to reverse the input sequence. In Numpy notation, if our input is x, the output should be x[::-1]. Although this task sounds very simple, RNNs can have issues with such because the task requires long-term dependencies. Transformers are built to support such, and hence, we expect it to perform very well.

41.4.1. Dataset and Data Loaders

First, let's create a dataset class below.

```
class ReverseDataset(data.Dataset):

    def __init__(self, num_categories, seq_len, size):
        super().__init__()
```

```
        self.num_categories = num_categories
        self.seq_len = seq_len
        self.size = size

        self.data = torch.randint(self.num_categories, size=(self.size, self.seq_len))

    def __len__(self):
        return self.size

    def __getitem__(self, idx):
        inp_data = self.data[idx]
        labels = torch.flip(inp_data, dims=(0,))
        return inp_data, labels
```

We create an arbitrary number of random sequences of numbers between 0 and `num_categories-1`. The label is simply the tensor flipped over the sequence dimension. We can create the corresponding data loaders below.

```
dataset = partial(ReverseDataset, 10, 16)
train_loader = data.DataLoader(dataset(50000),
                               batch_size=128,
                               shuffle=True,
                               drop_last=True,
                               pin_memory=True)
val_loader   = data.DataLoader(dataset(1000), batch_size=128)
test_loader  = data.DataLoader(dataset(10000), batch_size=128)
```

```
inp_data, labels = train_loader.dataset[0]
print("Input data:", inp_data)
print("Labels:    ", labels)
```

```
Input data: tensor([0, 4, 1, 2, 5, 5, 7, 6, 9, 6, 3, 1, 9, 3, 1, 9])
Labels:     tensor([9, 1, 3, 9, 1, 3, 6, 9, 6, 7, 5, 5, 2, 1, 4, 0])
```

During training, we pass the input sequence through the Transformer encoder and predict the output for each input token. We use the standard Cross-Entropy loss to perform this. Every number is represented as a one-hot vector. Remember that representing the categories as single scalars decreases the expressiveness of the model extremely as 0 and 1 are not closer related than 0 and 9 in our example. An alternative to a one-hot vector is using a learned embedding vector as it is provided by the PyTorch module `nn.Embedding`. However, using a one-hot vector with an additional linear layer as in our case has the same effect as an embedding layer (`self.input_net` maps one-hot vector to a dense vector, where each row of the weight matrix represents the embedding for a specific category).

41.4.2. The Reverse Predictor Class

To implement the training dynamic, we create a new class inheriting from `TransformerPredictor` and overwriting the training, validation and test step functions, which were left empty in the base class. We also add a `_calculate_loss` function to calculate the loss and accuracy for a batch.

```
class ReversePredictor(TransformerPredictor):

    def _calculate_loss(self, batch, mode="train"):
        # Fetch data and transform categories to one-hot vectors
        inp_data, labels = batch
        inp_data = F.one_hot(inp_data, num_classes=self.hparams.num_classes).float()

        # Perform prediction and calculate loss and accuracy
        preds = self.forward(inp_data, add_positional_encoding=True)
        loss = F.cross_entropy(preds.view(-1,preds.size(-1)), labels.view(-1))
        acc = (preds.argmax(dim=-1) == labels).float().mean()

        # Logging
        self.log(f"{mode}_loss", loss)
        self.log(f"{mode}_acc", acc)
        return loss, acc

    def training_step(self, batch, batch_idx):
        loss, _ = self._calculate_loss(batch, mode="train")
        return loss

    def validation_step(self, batch, batch_idx):
        _ = self._calculate_loss(batch, mode="val")

    def test_step(self, batch, batch_idx):
        _ = self._calculate_loss(batch, mode="test")
```

Finally, we can create a training function. We create a `pl.Trainer` object, running for N epochs, logging in TensorBoard, and saving our best model based on the validation. Afterward, we test our models on the test set.

41.4.3. Gradient Clipping

An additional parameter we pass to the trainer here is `gradient_clip_val`. This clips the norm of the gradients for all parameters before taking an optimizer step and prevents the model from diverging if we obtain very high gradients at, for instance, sharp

loss surfaces (see many good blog posts on gradient clipping, like DeepAI glossary). For Transformers, gradient clipping can help to further stabilize the training during the first few iterations, and also afterward. In plain PyTorch, you can apply gradient clipping via `torch.nn.utils.clip_grad_norm_(...)` (see documentation). The clip value is usually between 0.5 and 10, depending on how harsh you want to clip large gradients.

41.4.4. Implementation of the Lightning Trainer

The Lightning trainer can be implemented as follows:

```
def train_reverse(**kwargs):
    # Create a PyTorch Lightning trainer with the generation callback
    root_dir = os.path.join(CHECKPOINT_PATH, "ReverseTask")
    os.makedirs(root_dir, exist_ok=True)
    trainer = pl.Trainer(default_root_dir=root_dir,
                         callbacks=[ModelCheckpoint(save_weights_only=True,
                                    mode="max", monitor="val_acc")],
                         accelerator="gpu" if str(device).startswith("cuda") else "cpu
                         devices=1,
                         max_epochs=10,
                         gradient_clip_val=5)
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, "ReverseTask.ckpt")
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        model = ReversePredictor.load_from_checkpoint(pretrained_filename)
    else:
        model = ReversePredictor(max_iters=trainer.max_epochs*len(train_loader), **kwa
        trainer.fit(model, train_loader, val_loader)

    # Test best model on validation and test set
    val_result = trainer.test(model, val_loader, verbose=False)
    test_result = trainer.test(model, test_loader, verbose=False)
    result = {"test_acc": test_result[0]["test_acc"], "val_acc": val_result[0]["test_a

    model = model.to(device)
    return model, result
```

41.4.5. Training the Model

Finally, we can train the model. In this setup, we will use a single encoder block and a single head in the Multi-Head Attention. This is chosen because of the simplicity of the task, and in this case, the attention can actually be interpreted as an "explanation" of the predictions (compared to the other papers above dealing with deep Transformers).

```
reverse_model, reverse_result = train_reverse(input_dim=train_loader.dataset.num_categories,
                                              model_dim=32,
                                              num_heads=1,
                                              num_classes=train_loader.dataset.num_categories,
                                              num_layers=1,
                                              dropout=0.0,
                                              lr=5e-4,
                                              warmup=50)
```

```
Found pretrained model, loading...
```

```
Testing: |          | 0/? [00:00<?, ?it/s]
```

```
Testing: |          | 0/? [00:00<?, ?it/s]
```

The warning of PyTorch Lightning regarding the number of workers can be ignored for now. As the data set is so simple and the `__getitem__` finishes a neglectable time, we don't need subprocesses to provide us the data (in fact, more workers can slow down the training as we have communication overhead among processes/threads). First, let's print the results:

```
print(f"Val accuracy:  {(100.0 * reverse_result['val_acc']):4.2f}%")
print(f"Test accuracy: {(100.0 * reverse_result['test_acc']):4.2f}%")
```

```
Val accuracy:  100.00%
Test accuracy: 100.00%
```

As we would have expected, the Transformer can correctly solve the task.

41.5. Visualizing Attention Maps

How does the attention in the Multi-Head Attention block looks like for an arbitrary input? Let's try to visualize it below.

```
data_input, labels = next(iter(val_loader))
inp_data = F.one_hot(data_input, num_classes=reverse_model.hparams.num_classes).float
inp_data = inp_data.to(device)
attention_maps = reverse_model.get_attention_maps(inp_data)
```

The object `attention_maps` is a list of length N where N is the number of layers. Each element is a tensor of shape [Batch, Heads, SeqLen, SeqLen], which we can verify below.

```
attention_maps[0].shape
```

```
torch.Size([128, 1, 16, 16])
```

Next, we will write a plotting function that takes as input the sequences, attention maps, and an index indicating for which batch element we want to visualize the attention map. We will create a plot where over rows, we have different layers, while over columns, we show the different heads. Remember that the softmax has been applied for each row separately.

```
def plot_attention_maps(input_data, attn_maps, idx=0):
    if input_data is not None:
        input_data = input_data[idx].detach().cpu().numpy()
    else:
        input_data = np.arange(attn_maps[0][idx].shape[-1])
    attn_maps = [m[idx].detach().cpu().numpy() for m in attn_maps]

    num_heads = attn_maps[0].shape[0]
    num_layers = len(attn_maps)
    seq_len = input_data.shape[0]
    fig_size = 4 if num_heads == 1 else 3
    fig, ax = plt.subplots(num_layers, num_heads, figsize=(num_heads*fig_size, num_lay
    if num_layers == 1:
        ax = [ax]
    if num_heads == 1:
        ax = [[a] for a in ax]
    for row in range(num_layers):
        for column in range(num_heads):
            ax[row][column].imshow(attn_maps[row][column], origin='lower', vmin=0)
```

```
            ax[row][column].set_xticks(list(range(seq_len)))
            ax[row][column].set_xticklabels(input_data.tolist())
            ax[row][column].set_yticks(list(range(seq_len)))
            ax[row][column].set_yticklabels(input_data.tolist())
            ax[row][column].set_title(f"Layer {row+1}, Head {column+1}")
    fig.subplots_adjust(hspace=0.5)
    cax = fig.add_axes([0.95, 0.15, 0.01, 0.7])
    cbar = fig.colorbar(ax[0][0].imshow(attn_maps[0][0], origin='lower', vmin=0), cax=cax)
    cbar.set_label('Attention')
    plt.show()
```

Finally, we can plot the attention map of our trained Transformer on the reverse task:

```
plot_attention_maps(data_input, attention_maps, idx=0)
```

The model has learned to attend to the token that is on the flipped index of itself. Hence, it actually does what we intended it to do. We see that it however also pays some attention to values close to the flipped index. This is because the model doesn't need the perfect, hard attention to solve this problem, but is fine with this approximate,

noisy attention map. The close-by indices are caused by the similarity of the positional encoding, which we also intended with the positional encoding.

41.6. Conclusion

In this chapter, we took a closer look at the Multi-Head Attention layer which uses a scaled dot product between queries and keys to find correlations and similarities between input elements. The Transformer architecture is based on the Multi-Head Attention layer and applies multiple of them in a ResNet-like block. The Transformer is a very important, recent architecture that can be applied to many tasks and datasets. Although it is best known for its success in NLP, there is so much more to it. We have seen its application on sequence-to-sequence tasks. Its property of being permutation-equivariant if we do not provide any positional encodings, allows it to generalize to many settings. Hence, it is important to know the architecture, but also its possible issues such as the gradient problem during the first iterations solved by learning rate warm-up. If you are interested in continuing with the study of the Transformer architecture, please have a look at the blog posts listed in the "Further Reading" section below.

41.7. Additional Considerations

41.7.1. Complexity and Path Length

We can compare the self-attention operation with our other common layer competitors for sequence data: convolutions and recurrent neural networks. In Figure 41.7 you can find a table by Vaswani et al. (2017) on the complexity per layer, the number of sequential operations, and maximum path length. The complexity is measured by the upper bound of the number of operations to perform, while the maximum path length represents the maximum number of steps a forward or backward signal has to traverse to reach any other position. The lower this length, the better gradient signals can backpropagate for long-range dependencies. Let's take a look at the table in Figure 41.7.

n is the sequence length, d is the representation dimension and k is the kernel size of convolutions. In contrast to recurrent networks, the self-attention layer can parallelize all its operations making it much faster to execute for smaller sequence lengths. However, when the sequence length exceeds the hidden dimensionality, self-attention becomes more expensive than RNNs. One way of reducing the computational cost for long sequences is by restricting the self-attention to a neighborhood of inputs to attend over, denoted by r. Nevertheless, there has been recently a lot of work on more efficient Transformer architectures that still allow long dependencies, of which you can find an overview in the paper by Tay et al. (2020) if interested.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Figure 41.7.: Comparison of complexity and path length of different sequence layers. Table taken from Lippe (2022)

41.8. Further Reading

There are of course many more tutorials out there about attention and Transformers. Below, we list a few that are worth exploring if you are interested in the topic and might want yet another perspective on the topic after this one:

- Transformer: A Novel Neural Network Architecture for Language Understanding (Jakob Uszkoreit, 2017) - The original Google blog post about the Transformer paper, focusing on the application in machine translation.
- The Illustrated Transformer (Jay Alammar, 2018) - A very popular and great blog post intuitively explaining the Transformer architecture with many nice visualizations. The focus is on NLP.
- Attention? Attention! (Lilian Weng, 2018) - A nice blog post summarizing attention mechanisms in many domains including vision.
- Illustrated: Self-Attention (Raimi Karim, 2019) - A nice visualization of the steps of self-attention. Recommended going through if the explanation below is too abstract for you.
- The Transformer family (Lilian Weng, 2020) - A very detailed blog post reviewing more variants of Transformers besides the original one.

42. Hyperparameter Tuning of a Transformer Network with PyTorch Lightning

42.1. Basic Setup

This section provides an overview of the hyperparameter tuning process using `spotpython` and `PyTorch` Lightning. It uses the `Diabetes` data set (see Section E.1) for a regression task.

In this section, we will show how `spotpython` can be integrated into the `PyTorch` Lightning training workflow for a regression task. It demonstrates how easy it is to use `spotpython` to tune hyperparameters for a `PyTorch` Lightning model.

After importing the necessary libraries, the `fun_control` dictionary is set up via the `fun_control_init` function. The `fun_control` dictionary contains

- `PREFIX`: a unique identifier for the experiment
- `fun_evals`: the number of function evaluations
- `max_time`: the maximum run time in minutes
- `data_set`: the data set. Here we use the `Diabetes` data set that is provided by `spotpython`.
- `core_model_name`: the class name of the neural network model. This neural network model is provided by `spotpython`.
- `hyperdict`: the hyperparameter dictionary. This dictionary is used to define the hyperparameters of the neural network model. It is also provided by `spotpython`.
- `_L_in`: the number of input features. Since the `Diabetes` data set has 10 features, `_L_in` is set to 10.
- `_L_out`: the number of output features. Since we want to predict a single value, `_L_out` is set to 1.

The method `set_hyperparameter` allows the user to modify default hyperparameter settings. Here we set the `initialization` method to `["Default"]`. No other initializations are used in this experiment. The `HyperLight` class is used to define the objective function `fun`. It connects the `PyTorch` and the `spotpython` methods and is provided by `spotpython`. Finally, a `Spot` object is created.

```
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_co
from spotpython.utils.eda import print_exp_table
from spotpython.hyperparameters.values import set_hyperparameter
from spotpython.spot import Spot
from spotpython.utils.file import get_experiment_filename
from spotpython.utils.scaler import TorchStandardScaler

fun_control = fun_control_init(
    PREFIX="603",
    TENSORBOARD_CLEAN=True,
    tensorboard_log=True,
    fun_evals=inf,
    max_time=1,
    data_set = Diabetes(),
    scaler=TorchStandardScaler(),
    core_model_name="light.regression.NNTransformerRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1)

set_hyperparameter(fun_control, "optimizer", [
                "Adadelta",
                "Adagrad",
                "Adam",
                "AdamW",
                "Adamax",
            ])
set_hyperparameter(fun_control, "epochs", [5, 7])
set_hyperparameter(fun_control, "nhead", [1, 2])
set_hyperparameter(fun_control, "dim_feedforward_mult", [1, 1])

design_control = design_control_init(init_size=5)
surrogate_control = surrogate_control_init(
    noise=True,
    min_Lambda=1e-3,
    max_Lambda=10,
)

fun = HyperLight().fun

spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control, surr
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2025_02_12_23_05_23_0
Created spot_tensorboard_path: runs/spot_logs/603_maans08_2025-02-12_23-05-23 for SummaryWriter()
module_name: light
submodule_name: regression
model_name: NNTransformerRegressor
```

We can take a look at the design table to see the initial design.

```
print_exp_table(fun_control)
```

```
| name                  | type   | default         |   lower |   upper | transform              |
|-----------------------|--------|-----------------|---------|---------|------------------------|
| d_model_mult          | int    | 4               |    1    |     5   | transform_power_2_int  |
| nhead                 | int    | 3               |    1    |     2   | transform_power_2_int  |
| num_encoder_layers    | int    | 1               |    1    |     4   | transform_power_2_int  |
| dim_feedforward_mult  | int    | 1               |    1    |     1   | transform_power_2_int  |
| epochs                | int    | 7               |    5    |     7   | transform_power_2_int  |
| batch_size            | int    | 5               |    5    |     8   | transform_power_2_int  |
| optimizer             | factor | Adam            |    0    |     4   | None                   |
| dropout               | float  | 0.1             |    0.01 |     0.1 | None                   |
| lr_mult               | float  | 0.1             |    0.01 |     0.3 | None                   |
| patience              | int    | 5               |    4    |     7   | transform_power_2_int  |
| initialization        | factor | xavier_uniform  |    0    |     3   | None                   |
```

Calling the method `run()` starts the hyperparameter tuning process on the local machine.

```
res = spot_tuner.run()
```

```
d_model: 8, dim_feedforward: 16

train_model result: {'val_loss': 23955.66015625, 'hp_metric': 23955.66015625}
d_model: 128, dim_feedforward: 256

train_model result: {'val_loss': 20858.498046875, 'hp_metric': 20858.498046875}
d_model: 32, dim_feedforward: 64

train_model result: {'val_loss': 23231.283203125, 'hp_metric': 23231.283203125}
d_model: 16, dim_feedforward: 32

train_model result: {'val_loss': 23903.546875, 'hp_metric': 23903.546875}
d_model: 8, dim_feedforward: 16
```

```
train_model result: {'val_loss': 23954.796875, 'hp_metric': 23954.796875}
d_model: 128, dim_feedforward: 256


train_model result: {'val_loss': 20798.4765625, 'hp_metric': 20798.4765625}
spotpython tuning: 20798.4765625 [###-------] 27.50%


d_model: 128, dim_feedforward: 256


train_model result: {'val_loss': 21316.568359375, 'hp_metric': 21316.568359375}
spotpython tuning: 20798.4765625 [#####-----] 54.80%


d_model: 128, dim_feedforward: 256


train_model result: {'val_loss': 20846.00390625, 'hp_metric': 20846.00390625}
spotpython tuning: 20798.4765625 [########--] 80.89%


d_model: 128, dim_feedforward: 256


train_model result: {'val_loss': 20704.849609375, 'hp_metric': 20704.849609375}
spotpython tuning: 20704.849609375 [##########] 100.00% Done...

Experiment saved to 603_res.pkl
```

Note that we have enabled Tensorboard-Logging, so we can visualize the results with Tensorboard. Execute the following command in the terminal to start Tensorboard.

```
tensorboard --logdir="runs/"
```

42.2. Looking at the Results

42.2.1. Tuning Progress

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with `spotpython`'s method `plot_progress`. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress(log_y=True, filename=None)
```

42.2.2. Tuned Hyperparameters and Their Importance

Results can be printed in tabular form.

```
from spotpython.utils.eda import print_res_table
print_res_table(spot_tuner)
```

```
| name                 | type   | default        |   lower |   upper | tuned                | transfo
|----------------------|--------|----------------|---------|---------|----------------------|--------
| d_model_mult         | int    | 4              |     1.0 |     5.0 | 5.0                  | transfo
| nhead                | int    | 3              |     1.0 |     2.0 | 2.0                  | transfo
| num_encoder_layers   | int    | 1              |     1.0 |     4.0 | 2.0                  | transfo
| dim_feedforward_mult | int    | 1              |     1.0 |     1.0 | 1.0                  | transfo
| epochs               | int    | 7              |     5.0 |     7.0 | 7.0                  | transfo
| batch_size           | int    | 5              |     5.0 |     8.0 | 5.0                  | transfo
| optimizer            | factor | Adam           |     0.0 |     4.0 | Adagrad              | None
| dropout              | float  | 0.1            |    0.01 |     0.1 | 0.03278787269519674  | None
| lr_mult              | float  | 0.1            |    0.01 |     0.3 | 0.1699181080561065   | None
| patience             | int    | 5              |     4.0 |     7.0 | 5.0                  | transfo
| initialization       | factor | xavier_uniform |     0.0 |     3.0 | kaiming_normal       | None
```

42.3. Hyperparameter Considerations

1. `d_model` (or `d_embedding`):
    - This is the dimension of the embedding space or the number of expected features in the input.

- All input features are projected into this dimensional space before entering the transformer encoder.
- This dimension must be divisible by `nhead` since each head in the multi-head attention mechanism will process a subset of `d_model/nhead` features.

2. `nhead`:

   - This is the number of attention heads in the multi-head attention mechanism.
   - It allows the transformer to jointly attend to information from different representation subspaces.
   - It's important that `d_model % nhead == 0` to ensure the dimensions are evenly split among the heads.

3. `num_encoder_layers`:

   - This specifies the number of transformer encoder layers stacked together.
   - Each layer contains a multi-head attention mechanism followed by position-wise feedforward layers.

4. `dim_feedforward`:

   - This is the dimension of the feedforward network model within the transformer encoder layer.
   - Typically, this dimension is larger than `d_model` (e.g., 2048 for a Transformer model with `d_model=512`).

42.3.1. Important: Constraints and Interconnections:

- `d_model` and `nhead`:
  - As mentioned, `d_model` must be divisible by `nhead`. This is critical because each attention head operates simultaneously on a part of the embedding, so `d_model/nhead` should be an integer.
- `num_encoder_layers` and `dim_feedforward`**:
  - These parameters are more flexible and can be chosen independently of `d_model` and `nhead`.
  - However, the choice of `dim_feedforward` does influence the computational cost and model capacity, as larger dimensions allow learning more complex representations.
- One hyperparameter does not strictly need to be a multiple of others except for ensuring `d_model % nhead == 0`.

42.3.2. Practical Considerations:

1. Setting `d_model`:

   - Common choices for `d_model` are powers of 2 (e.g., 256, 512, 1024).
   - Ensure that it matches the size of the input data after the linear projection layer.

2. Setting `nhead`:

   - Typically, values are 1, 2, 4, 8, etc., depending on the `d_model` value.
   - Each head works on a subset of features, so `d_model / nhead` should be large enough to be meaningful.

3. Setting `num_encoder_layers`:

   - Practical values range from 1 to 12 or more depending on the depth desired.
   - Deeper models can capture more complex patterns but are also more computationally intensive.

4. Setting `dim_feedforward`:

   - Often set to a multiple of `d_model`, such as 2048 when `d_model` is 512.
   - Ensures sufficient capacity in the intermediate layers for complex feature transformations.

> **Note: `d_model` Calculation**
>
> Since `d_model % nhead == 0` is a critical constraint to ensure that the multi-head attention mechanism can operate effectively, `spotpython` computes the value of `d_model` based on the `nhead` value provided by the user. This ensures that the hyperparameter configuration is valid. So, the final value of `d_model` is a multiple of `nhead`. `spotpython` uses the hyperparameter `d_model_mult` to determine the multiple of `nhead` to use for `d_model`, i.e., `d_model = nhead * d_model_mult`.

> **Note: `dim_feedforward` Calculation**
>
> Since this dimension is typically larger than `d_model` (e.g., 2048 for a Transformer model with `d_model=512`), `spotpython` uses the hyperparameter `dim_feedforward_mult` to determine the multiple of `d_model` to use for `dim_feedforward`, i.e., `dim_feedforward = d_model * dim_feedforward_mult`.

42.4. Summary

This section presented an introduction to the basic setup of hyperparameter tuning of a transformer with `spotpython` and `PyTorch` Lightning.

43. Saving and Loading

This tutorial shows how to save and load objects in `spotpython`. It is split into the following parts:

- Section 43.1 shows how to save and load objects in `spotpython`, if `spotpython` is used as an optimizer.
- Section 43.2 shows how to save and load hyperparameter tuning experiments.
- Section 43.3 shows how to save and load `PyTorch Lightning` models.
- Section 43.4 shows how to convert a `PyTorch Lightning` model to a plain `PyTorch` model.

43.1. spotpython: Saving and Loading Optimization Experiments

In this section, we will show how results from `spotpython` can be saved and reloaded. Here, `spotpython` can be used as an optimizer. If `spotpython` is used as an optimizer, no dictionary of hyperparameters has be specified. The `fun_control` dictionary is sufficient.

```
import os
import pprint
from spotpython.utils.file import load_experiment
from spotpython.utils.file import get_experiment_filename
import numpy as np
from math import inf
from spotpython.spot import Spot
from spotpython.utils.init import (
    fun_control_init,
    design_control_init,
    surrogate_control_init,
    optimizer_control_init)
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical().fun_branin
fun_control = fun_control_init(
            PREFIX="branin",
            lower = np.array([0, 0]),
```

```
            upper = np.array([10, 10]),
            fun_evals=8,
            fun_repeats=1,
            max_time=inf,
            noise=False,
            tolerance_x=0,
            ocba_delta=0,
            var_type=["num", "num"],
            infill_criterion="ei",
            n_points=1,
            seed=123,
            log_level=20,
            show_models=False,
            show_progress=True)
design_control = design_control_init(
            init_size=5,
            repeats=1)
surrogate_control = surrogate_control_init(
            model_fun_evals=10000,
            min_theta=-3,
            max_theta=3,
            n_theta=2,
            theta_init_zero=True,
            n_p=1,
            optim_p=False,
            var_type=["num", "num"],
            seed=124)
optimizer_control = optimizer_control_init(
            max_iter=1000,
            seed=125)
spot_tuner = Spot(fun=fun,
            fun_control=fun_control,
            design_control=design_control,
            surrogate_control=surrogate_control,
            optimizer_control=optimizer_control)
spot_tuner.run()
PREFIX = fun_control["PREFIX"]
filename = get_experiment_filename(PREFIX)
spot_tuner.save_experiment(filename=filename)
print(f"filename: {filename}")
```

```
(spot_tuner_1, fun_control_1, design_control_1,
    surrogate_control_1, optimizer_control_1) = load_experiment(filename)
```

Table 43.1.

```
spot_tuner.print_results()
```

The progress of the original experiment is shown in Figure 43.1 and the reloaded experiment in Figure 43.2.

```
spot_tuner.plot_progress(log_y=True)
```

Figure 43.1.

```
spot_tuner_1.plot_progress(log_y=True)
```

Figure 43.2.

The results from the original experiment are shown in Table 43.1 and the reloaded experiment in Table 43.2.

43.1.1. Getting the Tuned Hyperparameters

The tuned hyperparameters can be obtained as a dictionary with the following code. Since `spotpython` is used as an optimizer, the numerical levels of the hyperparameters are identical to the optimized values of the underlying optimization problem, here: the Branin function.

```
from spotpython.hyperparameters.values import get_tuned_hyperparameters
get_tuned_hyperparameters(spot_tuner=spot_tuner)
```

Summary: Saving and Loading Optimization Experiments

- If `spotpython` is used as an optimizer (without an hyperparameter dictionary), experiments can be saved and reloaded with the `save_experiment` and `load_experiment` functions.
- The tuned hyperparameters can be obtained with the `get_tuned_hyperparameters` function.

Table 43.2.

```
spot_tuner_1.print_results()
```

43.2. spotpython as a Hyperparameter Tuner

If `spotpython` is used as a hyperparameter tuner, in addition to the `fun_control` dictionary a `core_model` dictionary has to be specified. Furthermore, a data set has to be selected and added to the `fun_control` dictionary. Here, we will use the `Diabetes` data set.

43.2.1. The Diabetes Data Set

The hyperparameter tuning of a `PyTorch Lightning` network on the `Diabetes` data set is used as an example. The `Diabetes` data set is a PyTorch Dataset for regression, which originates from the `scikit-learn` package, see https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_diabetes.html#sklearn.datasets.load_diabetes.

Ten baseline variables, age, sex, body mass index, average blood pressure, and six blood serum measurements were obtained for each of n = 442 diabetes patients, as well as the response of interest, a quantitative measure of disease progression one year after baseline. The `Diabetes` data set is has the following properties:

- Samples total: 442
- Dimensionality: 10
- Features: real, $-.2 < x < .2$
- Targets: integer $25 - 346$

```
from spotpython.data.diabetes import Diabetes
data_set = Diabetes()
```

```
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_co
from spotpython.utils.eda import print_exp_table
from spotpython.spot import Spot
from spotpython.utils.file import get_experiment_filename

PREFIX="604"
fun_control = fun_control_init(
```

```
    save_experiment=True,
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1)

fun = HyperLight().fun

from spotpython.hyperparameters.values import set_hyperparameter
set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,5])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,3])

design_control = design_control_init(init_size=10)

print_exp_table(fun_control)
```

In contrast to the default setttin, where `sava_experiment` is set to `False`, here the `fun_control` dictionary is initialized `save_experiment=True`. Alternatively, an existing `fun_control` dictionary can be updated with `{"save_experiment": True}` as shown in the following code.

```
fun_control.update({"save_experiment": True})
```

If `save_experiment` is set to `True`, the results of the hyperparameter tuning experiment are stored in a pickle file with the name `PREFIX` after the tuning is finished in the current directory.

Alternatively, the spot object and the corresponding dictionaries can be saved with the `save_experiment` method, which is part of the `spot` object. Therefore, the `spot` object has to be created as shown in the following code.

```
spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
spot_tuner.save_experiment(path="userExperiment", overwrite=False)
```

Here, we have added a `path` argument to specify the directory where the experiment is saved. The resulting pickle file can be copied to another directory or computer and

reloaded with the `load_experiment` function. It can also be used for performing the tuning run. Here, we will execute the tuning run on the local machine, which can be done with the following code.

```
res = spot_tuner.run()
```

After the tuning run is finished, a pickle file with the name `spot_604_experiment.pickle` is stored in the local directory. This is a result of setting the `save_experiment` argument to `True` in the `fun_control` dictionary. We can load the experiment with the following code. Here, we have specified the `PREFIX` as an argument to the `load_experiment` function. Alternatively, the filename (`filename`) can be used as an argument.

```
from spotpython.utils.file import load_experiment
(spot_tuner_1, fun_control_1, design_control_1,
    surrogate_control_1, optimizer_control_1) = load_experiment(PREFIX=PREFIX)
```

For comparison, the tuned hyperparameters of the original experiment are shown first:

```
get_tuned_hyperparameters(spot_tuner, fun_control)
```

Second, the tuned hyperparameters of the reloaded experiment are shown:

```
get_tuned_hyperparameters(spot_tuner_1, fun_control_1)
```

Note: The numerical levels of the hyperparameters are used as keys in the dictionary. If the `fun_control` dictionary is used, the names of the hyperparameters are used as keys in the dictionary.

```
get_tuned_hyperparameters(spot_tuner_1, fun_control_1)
```

Plot the progress of the original experiment are identical to the reloaded experiment.

```
spot_tuner.plot_progress()
```

Figure 43.3.

```
spot_tuner_1.plot_progress()
```

Figure 43.4.

> **Summary: Saving and Loading Hyperparameter-Tuning Experiments**
>
> - If `spotpython` is used as an hyperparameter tuner (with an hyperparameter dictionary), experiments can be saved and reloaded with the `save_experiment` and `load_experiment` functions.
> - The tuned hyperparameters can be obtained with the `get_tuned_hyperparameters` function.

43.3. Saving and Loading PyTorch Lightning Models

Section 43.1 and Section 43.2 explained how to save and load optimization and hyperparameter tuning experiments and how to get the tuned hyperparameters as a dictionary. This section shows how to save and load `PyTorch Lightning` models.

43.3.1. Get the Tuned Architecture

In contrast to the function `get_tuned_hyperparameters`, the function `get_tuned_architecture` returns the tuned architecture of the model as a dictionary. Here, the transformations are already applied to the numerical levels of the hyperparameters and the encoding (and types) are the original types of the hyperparameters used by the model. Important: The `config` dictionary from `get_tuned_architecture` can be passed to the model without any modifications.

```
from spotpython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(spot_tuner)
pprint.pprint(config)
```

After getting the tuned architecture, the model can be created and tested with the following code.

```
from spotpython.light.testmodel import test_model
test_model(config, fun_control)
```

43.3.2. Load a Model from Checkpoint

The method `load_light_from_checkpoint` loads a model from a checkpoint file. Important: The model has to be trained before the checkpoint is loaded. As shown here, loading a model with trained weights is possible, but requires two steps:

1. The model weights have to be learned using `test_model`. The `test_model` method writes a checkpoint file.
2. The model has to be loaded from the checkpoint file.

43.3.2.1. Details About the `load_light_from_checkpoint` Method

- The `test_model` method saves the last checkpoint to a file using the following code:

```
ModelCheckpoint(
    dirpath=os.path.join(fun_control["CHECKPOINT_PATH"], config_id), save_last=True
),
```

The filename of the last checkpoint has a specific structure:

- A `config_id` is generated from the `config` dictionary. It does not use a timestamp. This differs from the config id generated in cvmodel.py and trainmodel.py, which provide time information for the TensorBoard logging.
- Furthermore, the postfix `_TEST` is added to the `config_id` to indicate that the model is tested.
- For example: `runs/saved_models/16_16_64_LeakyReLU_Adadelta_0.0014_8.5895_8_False_ka`

```
from spotpython.light.loadmodel import load_light_from_checkpoint
model_loaded = load_light_from_checkpoint(config, fun_control)
```

```
vars(model_loaded)
```

```
import torch
torch.save(model_loaded, "model.pt")
```

```
mymodel = torch.load("model.pt")
```

```
# show all attributes of the model
vars(mymodel)
```

43.4. Converting a Lightning Model to a Plain Torch Model

43.4.1. The Function `get_removed_attributes_and_base_net`

`spotpython` provides a function to covert a `PyTorch Lightning` model to a plain `PyTorch` model. The function `get_removed_attributes_and_base_net` returns a tuple with the removed attributes and the base net. The base net is a plain `PyTorch` model. The removed attributes are the attributes of the `PyTorch Lightning` model that are not part of the base net.

This conversion can be reverted.

```
import numpy as np
import torch
from spotpython.utils.device import getDevice
from torch.utils.data import random_split
from spotpython.utils.classes import get_removed_attributes_and_base_net
from spotpython.hyperparameters.optimizer import optimizer_handler
removed_attributes, torch_net = get_removed_attributes_and_base_net(net=mymodel)
```

```
print(removed_attributes)
```

```
print(torch_net)
```

43.4.2. An Example how to use the Plain Torch Net

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

# Load the Diabetes dataset from sklearn
diabetes = load_diabetes()
X = diabetes.data
y = diabetes.target
```

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=

# Scale the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

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

# Create a PyTorch dataset
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

# Create a PyTorch dataloader
batch_size = 32
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size)

torch_net.to(getDevice("cpu"))

# train the net
criterion = nn.MSELoss()
optimizer = optim.Adam(torch_net.parameters(), lr=0.01)
n_epochs = 100
losses = []
for epoch in range(n_epochs):
    for inputs, targets in train_dataloader:
        targets = targets.view(-1, 1)
        optimizer.zero_grad()
        outputs = torch_net(inputs)
        loss = criterion(outputs, targets)
        losses.append(loss.item())
        loss.backward()
        optimizer.step()
# visualize the network training
plt.plot(losses)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```

44. Hyperparameter Tuning with spotpython and PyTorch Lightning for the Diabetes Data Set Using a ResNet Model

In this section, we will show how spotpython can be integrated into the PyTorch Lightning training workflow for a regression task. It demonstrates how easy it is to use spotpython to tune hyperparameters for a PyTorch Lightning model.

After importing the necessary libraries, the fun_control dictionary is set up via the fun_control_init function. The fun_control dictionary contains

- PREFIX: a unique identifier for the experiment
- fun_evals: the number of function evaluations
- max_time: the maximum run time in minutes
- data_set: the data set. Here we use the Diabetes data set that is provided by spotpython.
- core_model_name: the class name of the neural network model. This neural network model is provided by spotpython.
- hyperdict: the hyperparameter dictionary. This dictionary is used to define the hyperparameters of the neural network model. It is also provided by spotpython.
- _L_in: the number of input features. Since the Diabetes data set has 10 features, _L_in is set to 10.
- _L_out: the number of output features. Since we want to predict a single value, _L_out is set to 1.

The HyperLight class is used to define the objective function fun. It connects the PyTorch and the spotpython methods and is provided by spotpython.

```
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_control_init)
from spotpython.utils.eda import print_exp_table
from spotpython.spot import Spot
from spotpython.utils.file import get_experiment_filename
```

```
PREFIX="605"

data_set = Diabetes()

fun_control = fun_control_init(
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNResNetRegressor",
    hyperdict=LightHyperDict,
    _L_in=10,
    _L_out=1)

fun = HyperLight().fun
```

```
module_name: light
submodule_name: regression
model_name: NNResNetRegressor
```

The method `set_hyperparameter` allows the user to modify default hyperparameter settings. Here we modify some hyperparameters to keep the model small and to decrease the tuning time.

```
from spotpython.hyperparameters.values import set_hyperparameter
set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,7])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,3])
set_hyperparameter(fun_control, "lr_mult", [0.1, 20.0])

design_control = design_control_init(init_size=10)

print_exp_table(fun_control)
```

```
| name           | type    | default    |   lower |   upper | transform             |
|----------------|---------|------------|---------|---------|-----------------------|
| l1             | int     | 3          |     3   |   4     | transform_power_2_int |
| epochs         | int     | 4          |     3   |   7     | transform_power_2_int |
| batch_size     | int     | 4          |     4   |  11     | transform_power_2_int |
```

```
| act_fn         | factor | ReLU      |     0   |   5     | None                  |
| optimizer      | factor | SGD       |     0   |   2     | None                  |
| dropout_prob   | float  | 0.01      |     0   |   0.025 | None                  |
| lr_mult        | float  | 1.0       |     0.1 |  20     | None                  |
| patience       | int    | 2         |     2   |   3     | transform_power_2_int |
| initialization | factor | Default   |     0   |   4     | None                  |
```

Finally, a `Spot` object is created. Calling the method `run()` starts the hyperparameter tuning process.

```
spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
res = spot_tuner.run()
```

```
train_model result: {'val_loss': 24041.166015625, 'hp_metric': 24041.166015625}

train_model result: {'val_loss': 22516.333984375, 'hp_metric': 22516.333984375}

train_model result: {'val_loss': 23957.216796875, 'hp_metric': 23957.216796875}

train_model result: {'val_loss': 23526.126953125, 'hp_metric': 23526.126953125}

train_model result: {'val_loss': 23435.740234375, 'hp_metric': 23435.740234375}

train_model result: {'val_loss': 5741.01416015625, 'hp_metric': 5741.01416015625}

train_model result: {'val_loss': 23971.83984375, 'hp_metric': 23971.83984375}

train_model result: {'val_loss': 24112.75, 'hp_metric': 24112.75}

train_model result: {'val_loss': 20110.208984375, 'hp_metric': 20110.208984375}
train_model result: {'val_loss': 23546.658203125, 'hp_metric': 23546.658203125}

train_model result: {'val_loss': 4446.037109375, 'hp_metric': 4446.037109375}
spotpython tuning: 4446.037109375 [#---------] 5.07%

train_model result: {'val_loss': 4507.2099609375, 'hp_metric': 4507.2099609375}
spotpython tuning: 4446.037109375 [#---------] 10.72%

train_model result: {'val_loss': 3540.979248046875, 'hp_metric': 3540.979248046875}
spotpython tuning: 3540.979248046875 [##--------] 17.73%
```

```
train_model result: {'val_loss': 3842.3193359375, 'hp_metric': 3842.3193359375}
spotpython tuning: 3540.979248046875 [###-------] 28.32%


train_model result: {'val_loss': 4208.6962890625, 'hp_metric': 4208.6962890625}
spotpython tuning: 3540.979248046875 [####------] 35.07%


train_model result: {'val_loss': 22659.771484375, 'hp_metric': 22659.771484375}
spotpython tuning: 3540.979248046875 [####------] 37.69%


train_model result: {'val_loss': 23285.0, 'hp_metric': 23285.0}
spotpython tuning: 3540.979248046875 [######----] 55.81%


train_model result: {'val_loss': 3424.77734375, 'hp_metric': 3424.77734375}
spotpython tuning: 3424.77734375 [#######---] 71.51%


train_model result: {'val_loss': 3610.14599609375, 'hp_metric': 3610.14599609375}
spotpython tuning: 3424.77734375 [########--] 80.83%


train_model result: {'val_loss': 3504.15673828125, 'hp_metric': 3504.15673828125}
spotpython tuning: 3424.77734375 [#########-] 86.53%


train_model result: {'val_loss': 23873.259765625, 'hp_metric': 23873.259765625}
spotpython tuning: 3424.77734375 [#########-] 89.97%


train_model result: {'val_loss': 23728.546875, 'hp_metric': 23728.546875}
spotpython tuning: 3424.77734375 [#########-] 92.04%


train_model result: {'val_loss': 3299.1708984375, 'hp_metric': 3299.1708984375}
spotpython tuning: 3299.1708984375 [##########] 100.00%


train_model result: {'val_loss': 4529.72998046875, 'hp_metric': 4529.72998046875}
spotpython tuning: 3299.1708984375 [##########] 100.00% Done...

Experiment saved to 605_res.pkl
```

44.1. Looking at the Results

44.1.1. Tuning Progress

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with `spotpython`'s method `plot_progress`. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress()
```

44.1.2. Tuned Hyperparameters and Their Importance

Results can be printed in tabular form.

```
from spotpython.utils.eda import print_res_table
print_res_table(spot_tuner)
```

```
| name           | type   | default    |   lower |   upper | tuned                | transform
|----------------|--------|------------|---------|---------|----------------------|------------------
| l1             | int    | 3          |     3.0 |     4.0 | 3.0                  | transform_power_2
| epochs         | int    | 4          |     3.0 |     7.0 | 6.0                  | transform_power_2
| batch_size     | int    | 4          |     4.0 |    11.0 | 5.0                  | transform_power_2
| act_fn         | factor | ReLU       |     0.0 |     5.0 | ELU                  | None
| optimizer      | factor | SGD        |     0.0 |     2.0 | Adadelta             | None
| dropout_prob   | float  | 0.01       |     0.0 |   0.025 | 0.007965747080898462 | None
| lr_mult        | float  | 1.0        |     0.1 |    20.0 | 14.596273734272406   | None
| patience       | int    | 2          |     2.0 |     3.0 | 2.0                  | transform_power_2
| initialization | factor | Default    |     0.0 |     4.0 | xavier_uniform       | None
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=1.0)
```

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
l1:  0.002148012090321149
epochs:  0.0029948586652520748
batch_size:  1.1261073219435407
act_fn:  0.04013110934383669
optimizer:  5.970061245935568
dropout_prob:  100.0
lr_mult:  0.002863244135039334
patience:  0.002148012090321149
initialization:  0.002148012090321149
```

44.1.3. Get the Tuned Architecture

```
import pprint
from spotpython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(spot_tuner)
pprint.pprint(config)
```

```
{'act_fn': ELU(),
 'batch_size': 32,
 'dropout_prob': 0.007965747080898462,
 'epochs': 64,
 'initialization': 'xavier_uniform',
 'l1': 8,
 'lr_mult': 14.596273734272406,
 'optimizer': 'Adadelta',
 'patience': 4}
```

44.1.4. Test on the full data set

```
# set the value of the key "TENSORBOARD_CLEAN" to True in the fun_control dictionary a
import os
# if the directory "./runs" exists, delete it
if os.path.exists("./runs"):
    os.system("rm -r ./runs")
fun_control.update({"tensorboard_log": True})
```

```
from spotpython.light.testmodel import test_model
from spotpython.utils.init import get_feature_names

test_model(config, fun_control)
get_feature_names(fun_control)
```

```
        Test metric               DataLoader 0

         hp_metric              2492.49951171875
         val_loss               2492.49951171875


test_model result: {'val_loss': 2492.49951171875, 'hp_metric': 2492.49951171875}

['age',
 'sex',
 'bmi',
 'bp',
 's1_tc',
 's2_ldl',
 's3_hdl',
 's4_tch',
 's5_ltg',
 's6_glu']
```

44.1.5. Cross Validation With Lightning

- The `KFold` class from `sklearn.model_selection` is used to generate the folds for cross-validation.
- These mechanism is used to generate the folds for the final evaluation of the model.
- The `CrossValidationDataModule` class [SOURCE] is used to generate the folds for the hyperparameter tuning process.

- It is called from the `cv_model` function [SOURCE].

```
config
```

```
{'l1': 8,
 'epochs': 64,
 'batch_size': 32,
 'act_fn': ELU(),
 'optimizer': 'Adadelta',
 'dropout_prob': 0.007965747080898462,
 'lr_mult': 14.596273734272406,
 'patience': 4,
 'initialization': 'xavier_uniform'}
```

```
from spotpython.light.cvmodel import cv_model
fun_control.update({"k_folds": 2})
fun_control.update({"test_size": 0.6})
cv_model(config, fun_control)
```

```
k: 0

train_model result: {'val_loss': 3301.22509765625, 'hp_metric': 3301.22509765625}
k: 1
train_model result: {'val_loss': 3874.31982421875, 'hp_metric': 3874.31982421875}

3587.7724609375
```

44.2. Summary

This section presented an introduction to the basic setup of hyperparameter tuning with `spotpython` and `PyTorch` Lightning using a ResNet model for the Diabetes data set.

45. Hyperparameter Tuning with spotpython and PyTorch Lightning for the Diabetes Data Set Using a User Specified ResNet Model

After importing the necessary libraries, the `fun_control` dictionary is set up via the `fun_control_init` function. The `fun_control` dictionary contains

- `PREFIX`: a unique identifier for the experiment
- `fun_evals`: the number of function evaluations
- `max_time`: the maximum run time in minutes
- `data_set`: the data set. Here we use the `Diabetes` data set that is provided by `spotpython`.
- `core_model_name`: the class name of the neural network model. This neural network model is provided by `spotpython`.
- `hyperdict`: the hyperparameter dictionary. This dictionary is used to define the hyperparameters of the neural network model. It is also provided by `spotpython`.
- `_L_in`: the number of input features. Since the `Diabetes` data set has 10 features, `_L_in` is set to 10.
- `_L_out`: the number of output features. Since we want to predict a single value, `_L_out` is set to 1.

The `HyperLight` class is used to define the objective function `fun`. It connects the `PyTorch` and the `spotpython` methods and is provided by `spotpython`.

To access the user specified ResNet model, the path to the user model must be added to the Python path:

```
import sys
sys.path.insert(0, './userModel')
import my_resnet
import my_hyper_dict
```

In the following code, we do not specify the ResNet model in the `fun_control` dictionary. It will be added in a second step as the user specified model.

```
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_co
from spotpython.utils.eda import print_exp_table
from spotpython.spot import Spot
from spotpython.utils.file import get_experiment_filename

PREFIX="606-user-resnet"

data_set = Diabetes()

fun_control = fun_control_init(
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    _L_in=10,
    _L_out=1)

fun = HyperLight().fun
```

In a second step, we can add the user specified ResNet model to the `fun_control` dictionary:

```
from spotpython.hyperparameters.values import add_core_model_to_fun_control
add_core_model_to_fun_control(fun_control=fun_control,
                              core_model=my_resnet.MyResNet,
                              hyper_dict=my_hyper_dict.MyHyperDict)
```

The method `set_hyperparameter` allows the user to modify default hyperparameter settings. Here we modify some hyperparameters to keep the model small and to decrease the tuning time.

```
from spotpython.hyperparameters.values import set_hyperparameter
set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,7])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,3])
set_hyperparameter(fun_control, "lr_mult", [0.1, 20.0])
```

```
design_control = design_control_init(init_size=10)

print_exp_table(fun_control)
```

```
| name           | type   | default   |   lower |   upper | transform             |
|----------------|--------|-----------|---------|---------|-----------------------|
| l1             | int    | 3         |     3   |   4     | transform_power_2_int |
| epochs         | int    | 4         |     3   |   7     | transform_power_2_int |
| batch_size     | int    | 4         |     4   |  11     | transform_power_2_int |
| act_fn         | factor | ReLU      |     0   |   5     | None                  |
| optimizer      | factor | SGD       |     0   |   2     | None                  |
| dropout_prob   | float  | 0.01      |     0   |   0.025 | None                  |
| lr_mult        | float  | 1.0       |     0.1 |  20     | None                  |
| patience       | int    | 2         |     2   |   3     | transform_power_2_int |
| initialization | factor | Default   |     0   |   4     | None                  |
```

Finally, a `Spot` object is created. Calling the method `run()` starts the hyperparameter tuning process.

```
spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
res = spot_tuner.run()
```

```
Milestones: [16, 32, 48]

train_model result: {'val_loss': 24041.166015625, 'hp_metric': 24041.166015625}
Milestones: [2, 4, 6]

train_model result: {'val_loss': 22516.333984375, 'hp_metric': 22516.333984375}
Milestones: [16, 32, 48]

train_model result: {'val_loss': 23957.216796875, 'hp_metric': 23957.216796875}
Milestones: [2, 4, 6]

train_model result: {'val_loss': 23526.126953125, 'hp_metric': 23526.126953125}
Milestones: [32, 64, 96]

train_model result: {'val_loss': 23435.740234375, 'hp_metric': 23435.740234375}
Milestones: [32, 64, 96]

train_model result: {'val_loss': 5741.01416015625, 'hp_metric': 5741.01416015625}
Milestones: [4, 8, 12]
```

```
train_model result: {'val_loss': 23971.83984375, 'hp_metric': 23971.83984375}
Milestones: [4, 8, 12]


train_model result: {'val_loss': 24112.75, 'hp_metric': 24112.75}
Milestones: [8, 16, 24]


train_model result: {'val_loss': 20110.208984375, 'hp_metric': 20110.208984375}
Milestones: [8, 16, 24]
train_model result: {'val_loss': 23546.658203125, 'hp_metric': 23546.658203125}


Milestones: [32, 64, 96]
train_model result: {'val_loss': 4446.037109375, 'hp_metric': 4446.037109375}
spotpython tuning: 4446.037109375 [#---------] 5.27%


Milestones: [32, 64, 96]
train_model result: {'val_loss': 4507.2099609375, 'hp_metric': 4507.2099609375}
spotpython tuning: 4446.037109375 [#---------] 11.06%


Milestones: [32, 64, 96]
train_model result: {'val_loss': 3540.979248046875, 'hp_metric': 3540.979248046875}
spotpython tuning: 3540.979248046875 [##--------] 18.22%


Milestones: [4, 8, 12]


train_model result: {'val_loss': 3842.3193359375, 'hp_metric': 3842.3193359375}
spotpython tuning: 3540.979248046875 [###-------] 28.86%


Milestones: [32, 64, 96]
train_model result: {'val_loss': 4208.6962890625, 'hp_metric': 4208.6962890625}
spotpython tuning: 3540.979248046875 [####------] 35.65%


Milestones: [2, 4, 6]


train_model result: {'val_loss': 22659.771484375, 'hp_metric': 22659.771484375}
spotpython tuning: 3540.979248046875 [####------] 38.22%


Milestones: [32, 64, 96]
train_model result: {'val_loss': 23285.0, 'hp_metric': 23285.0}
spotpython tuning: 3540.979248046875 [######----] 55.48%
```

```
Milestones: [8, 16, 24]
train_model result: {'val_loss': 3424.77734375, 'hp_metric': 3424.77734375}
spotpython tuning: 3424.77734375 [#######---] 70.85%

Milestones: [32, 64, 96]
train_model result: {'val_loss': 3610.14599609375, 'hp_metric': 3610.14599609375}
spotpython tuning: 3424.77734375 [########--] 79.90%

Milestones: [2, 4, 6]

train_model result: {'val_loss': 3504.15673828125, 'hp_metric': 3504.15673828125}
spotpython tuning: 3424.77734375 [#########-] 85.56%

Milestones: [2, 4, 6]

train_model result: {'val_loss': 23873.259765625, 'hp_metric': 23873.259765625}
spotpython tuning: 3424.77734375 [#########-] 88.98%

Milestones: [32, 64, 96]
train_model result: {'val_loss': 23728.546875, 'hp_metric': 23728.546875}
spotpython tuning: 3424.77734375 [#########-] 91.06%

Milestones: [16, 32, 48]
train_model result: {'val_loss': 3299.1708984375, 'hp_metric': 3299.1708984375}
spotpython tuning: 3299.1708984375 [##########] 99.05%

Milestones: [8, 16, 24]
train_model result: {'val_loss': 4529.72998046875, 'hp_metric': 4529.72998046875}
spotpython tuning: 3299.1708984375 [##########] 100.00% Done...

Experiment saved to 606-user-resnet_res.pkl
```

45.1. Looking at the Results

45.1.1. Tuning Progress

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with `spotpython`'s method `plot_progress`. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress()
```

45.1.2. Tuned Hyperparameters and Their Importance

Results can be printed in tabular form.

```
from spotpython.utils.eda import print_res_table
print_res_table(spot_tuner)
```

```
| name           | type   | default   |   lower |   upper | tuned                | tra
|----------------|--------|-----------|---------|---------|----------------------|----
| l1             | int    | 3         |     3.0 |     4.0 | 3.0                  | tra
| epochs         | int    | 4         |     3.0 |     7.0 | 6.0                  | tra
| batch_size     | int    | 4         |     4.0 |    11.0 | 5.0                  | tra
| act_fn         | factor | ReLU      |     0.0 |     5.0 | ELU                  | Nor
| optimizer      | factor | SGD       |     0.0 |     2.0 | Adadelta             | Nor
| dropout_prob   | float  | 0.01      |     0.0 |   0.025 | 0.007965747080898462 | Nor
| lr_mult        | float  | 1.0       |     0.1 |    20.0 | 14.596273734272406   | Nor
| patience       | int    | 2         |     2.0 |     3.0 | 2.0                  | tra
| initialization | factor | Default   |     0.0 |     4.0 | xavier_uniform       | Nor
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=1.0)
```

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
l1:  0.002148012090321149
epochs:  0.0029948586652520748
batch_size:  1.1261073219435407
act_fn:  0.04013110934383669
optimizer:  5.970061245935568
dropout_prob:  100.0
lr_mult:  0.002863244135039334
patience:  0.002148012090321149
initialization:  0.002148012090321149
```

45.1.3. Get the Tuned Architecture

```
import pprint
from spotpython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(spot_tuner)
pprint.pprint(config)
```

```
{'act_fn': ELU(),
 'batch_size': 32,
 'dropout_prob': 0.007965747080898462,
 'epochs': 64,
 'initialization': 'xavier_uniform',
 'l1': 8,
 'lr_mult': 14.596273734272406,
 'optimizer': 'Adadelta',
 'patience': 4}
```

45.2. Details of the User-Specified ResNet Model

The specification of a user model requires three files:

- `my_resnet.py`: the Python file containing the user specified ResNet model
- `my_hyperdict.py`: the Python file for loading the hyperparameter dictionary `my_hyperdict.json` for the user specified ResNet model
- `my_hyperdict.json`: the JSON file containing the hyperparameter dictionary for the user specified ResNet model

45.2.1. `my_resnet.py`

```
import lightning as L
import torch
from torch import nn
from spotpython.hyperparameters.optimizer import optimizer_handler
import torchmetrics.functional.regression
import torch.optim as optim

class ResidualBlock(nn.Module):
    def __init__(self, input_dim, output_dim, act_fn, dropout_prob):
        super(ResidualBlock, self).__init__()
        self.fc1 = nn.Linear(input_dim, output_dim)
        self.bn1 = nn.BatchNorm1d(output_dim)
        self.ln1 = nn.LayerNorm(output_dim)
        self.fc2 = nn.Linear(output_dim, output_dim)
        self.bn2 = nn.BatchNorm1d(output_dim)
        self.ln2 = nn.LayerNorm(output_dim)
        self.act_fn = act_fn
        self.dropout = nn.Dropout(dropout_prob)
        self.shortcut = nn.Sequential()

        if input_dim != output_dim:
            self.shortcut = nn.Sequential(
                nn.Linear(input_dim, output_dim),
                nn.BatchNorm1d(output_dim)
            )

    def forward(self, x):
        identity = self.shortcut(x)

        out = self.fc1(x)
        out = self.bn1(out)
        out = self.ln1(out)
        out = self.act_fn(out)
        out = self.dropout(out)
        out = self.fc2(out)
```

```
        out = self.bn2(out)
        out = self.ln2(out)
        out += identity  # Residual connection
        out = self.act_fn(out)
        return out

class MyResNet(L.LightningModule):
    def __init__(
        self,
        l1: int,
        epochs: int,
        batch_size: int,
        initialization: str,
        act_fn: nn.Module,
        optimizer: str,
        dropout_prob: float,
        lr_mult: float,
        patience: int,
        _L_in: int,
        _L_out: int,
        _torchmetric: str,
    ):
        super().__init__()
        self._L_in = _L_in
        self._L_out = _L_out
        if _torchmetric is None:
            _torchmetric = "mean_squared_error"
        self._torchmetric = _torchmetric
        self.metric = getattr(torchmetrics.functional.regression, _torchmetric)
        self.save_hyperparameters(ignore=["_L_in", "_L_out", "_torchmetric"])
        self.example_input_array = torch.zeros((batch_size, self._L_in))

        if self.hparams.l1 < 4:
            raise ValueError("l1 must be at least 4")

        # Get hidden sizes
        hidden_sizes = self._get_hidden_sizes()
        layer_sizes = [self._L_in] + hidden_sizes

        # Construct the layers with Residual Blocks and Linear Layer at the end
        layers = []
        for i in range(len(layer_sizes) - 1):
            layers.append(
                ResidualBlock(
```

```
                    layer_sizes[i],
                    layer_sizes[i + 1],
                    self.hparams.act_fn,
                    self.hparams.dropout_prob
                )
            )
        layers.append(nn.Linear(layer_sizes[-1], self._L_out))

        self.layers = nn.Sequential(*layers)

        # Initialization (Xavier, Kaiming, or Default)
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            if self.hparams.initialization == "xavier_uniform":
                nn.init.xavier_uniform_(module.weight)
            elif self.hparams.initialization == "xavier_normal":
                nn.init.xavier_normal_(module.weight)
            elif self.hparams.initialization == "kaiming_uniform":
                nn.init.kaiming_uniform_(module.weight)
            elif self.hparams.initialization == "kaiming_normal":
                nn.init.kaiming_normal_(module.weight)
            else: # "Default"
                nn.init.uniform_(module.weight)
            if module.bias is not None:
                nn.init.zeros_(module.bias)

    def _generate_div2_list(self, n, n_min) -> list:
        result = []
        current = n
        repeats = 1
        max_repeats = 4
        while current >= n_min:
            result.extend([current] * min(repeats, max_repeats))
            current = current // 2
            repeats = repeats + 1
        return result

    def _get_hidden_sizes(self):
        n_low = max(2, int(self._L_in / 4))  # Ensure minimum reasonable size
        n_high = max(self.hparams.l1, 2 * n_low)
        hidden_sizes = self._generate_div2_list(n_high, n_low)
        return hidden_sizes
```

```
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.layers(x)
        return x

    def _calculate_loss(self, batch):
        x, y = batch
        y = y.view(len(y), 1)
        y_hat = self(x)
        loss = self.metric(y_hat, y)
        return loss

    def training_step(self, batch: tuple) -> torch.Tensor:
        val_loss = self._calculate_loss(batch)
        return val_loss

    def validation_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) ->
        val_loss = self._calculate_loss(batch)
        self.log("val_loss", val_loss, prog_bar=prog_bar)
        self.log("hp_metric", val_loss, prog_bar=prog_bar)
        return val_loss

    def test_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> torch
        val_loss = self._calculate_loss(batch)
        self.log("val_loss", val_loss, prog_bar=prog_bar)
        self.log("hp_metric", val_loss, prog_bar=prog_bar)
        return val_loss

    def predict_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> to
        x, y = batch
        yhat = self(x)
        y = y.view(len(y), 1)
        yhat = yhat.view(len(yhat), 1)
        return (x, y, yhat)

    def configure_optimizers(self):
        optimizer = optimizer_handler(
            optimizer_name=self.hparams.optimizer,
            params=self.parameters(),
            lr_mult=self.hparams.lr_mult
        )

        # Dynamic creation of milestones based on the number of epochs.
        num_milestones = 3  # Number of milestones to divide the epochs
        milestones = [int(self.hparams.epochs / (num_milestones + 1) * (i + 1)) for i
```

```
        # Print milestones for debug purposes
        print(f"Milestones: {milestones}")

        # Create MultiStepLR scheduler with dynamic milestones and learning rate multiplier.
        scheduler = optim.lr_scheduler.MultiStepLR(
            optimizer,
            milestones=milestones,
            gamma=0.1  # Decay factor
        )

        # Learning rate scheduler configuration
        lr_scheduler_config = {
            "scheduler": scheduler,
            "interval": "epoch",  # Adjust learning rate per epoch
            "frequency": 1,      # Apply the scheduler at every epoch
        }

        return {"optimizer": optimizer, "lr_scheduler": lr_scheduler_config}
```

45.2.2. my_hyperdict.py

```
import json
from spotpython.data import base
import pathlib


class MyHyperDict(base.FileConfig):
    """User specified hyperparameter dictionary.

    This class extends the FileConfig class to provide a dictionary for storing hyperparameters.

    Attributes:
        filename (str):
            The name of the file where the hyperparameters are stored.
    """

    def __init__(
        self,
        filename: str = "my_hyper_dict.json",
        directory: None = None,
    ) -> None:
```

```
        super().__init__(filename=filename, directory=directory)
        self.filename = filename
        self.directory = directory
        self.hyper_dict = self.load()

    @property
    def path(self):
        if self.directory:
            return pathlib.Path(self.directory).joinpath(self.filename)
        return pathlib.Path(__file__).parent.joinpath(self.filename)

    def load(self) -> dict:
        """Load the hyperparameters from the file.

        Returns:
            dict: A dictionary containing the hyperparameters.

        Examples:
            # Assume the user specified file `my_hyper_dict.json` is in the `./hyperdi
            >>> user_lhd = MyHyperDict(filename='my_hyper_dict.json', directory='./hyp
        """
        with open(self.path, "r") as f:
            d = json.load(f)
        return d
```

45.2.3. my_hyperdict.json

```
"MyResNet": {
        "l1": {
            "type": "int",
            "default": 3,
            "transform": "transform_power_2_int",
            "lower": 3,
            "upper": 10
        },
        "epochs": {
            "type": "int",
            "default": 4,
            "transform": "transform_power_2_int",
            "lower": 4,
            "upper": 9
        },
```

```
        "batch_size": {
            "type": "int",
            "default": 4,
            "transform": "transform_power_2_int",
            "lower": 1,
            "upper": 6
        },
        "act_fn": {
            "levels": [
                "Sigmoid",
                "Tanh",
                "ReLU",
                "LeakyReLU",
                "ELU",
                "Swish"
            ],
            "type": "factor",
            "default": "ReLU",
            "transform": "None",
            "class_name": "spotpython.torch.activation",
            "core_model_parameter_type": "instance()",
            "lower": 0,
            "upper": 5
        },
        "optimizer": {
            "levels": [
                "Adadelta",
                "Adagrad",
                "Adam",
                "AdamW",
                "SparseAdam",
                "Adamax",
                "ASGD",
                "NAdam",
                "RAdam",
                "RMSprop",
                "Rprop",
                "SGD"
            ],
            "type": "factor",
            "default": "SGD",
            "transform": "None",
            "class_name": "torch.optim",
            "core_model_parameter_type": "str",
```

```
            "lower": 0,
            "upper": 11
        },
        "dropout_prob": {
            "type": "float",
            "default": 0.01,
            "transform": "None",
            "lower": 0.0,
            "upper": 0.25
        },
        "lr_mult": {
            "type": "float",
            "default": 1.0,
            "transform": "None",
            "lower": 0.1,
            "upper": 10.0
        },
        "patience": {
            "type": "int",
            "default": 2,
            "transform": "transform_power_2_int",
            "lower": 2,
            "upper": 6
        },
        "initialization": {
            "levels": [
                "Default",
                "kaiming_uniform",
                "kaiming_normal",
                "xavier_uniform",
                "xavier_normal"
            ],
            "type": "factor",
            "default": "Default",
            "transform": "None",
            "core_model_parameter_type": "str",
            "lower": 0,
            "upper": 4
        }
    }
```

45.3. Summary

This section presented an introduction to the basic setup of hyperparameter tuning with `spotpython` and `PyTorch` Lightning using a ResNet model for the Diabetes data set.

46. Hyperparameter Tuning with spotpython and PyTorch Lightning Using a CondNet Model

- We use the `Diabetes` dataset to illustrate the hyperparameter tuning process of a `CondNet` model using the `spotpython` package.
- The CondNet model is a conditional neural network that can be used to model conditional distributions [LINK].

```
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_control_init)
from spotpython.utils.eda import print_exp_table
from spotpython.spot import Spot
from spotpython.utils.file import get_experiment_filename
from math import inf
from spotpython.hyperparameters.values import set_hyperparameter

PREFIX="CondNet_01"

data_set = Diabetes()
input_dim = 10
output_dim = 1
cond_dim = 2

fun_control = fun_control_init(
    PREFIX=PREFIX,
    fun_evals=inf,
    max_time=1,
    data_set = data_set,
    core_model_name="light.regression.NNCondNetRegressor",
    hyperdict=LightHyperDict,
    _L_in=input_dim - cond_dim,
    _L_out=1,
    _L_cond=cond_dim,)
```

```
fun = HyperLight().fun


set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [3,4])
set_hyperparameter(fun_control, "epochs", [3,7])
set_hyperparameter(fun_control, "batch_size", [4,5])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,3])
set_hyperparameter(fun_control, "lr_mult", [0.1, 20.0])

design_control = design_control_init(init_size=10)

print_exp_table(fun_control)
```

```
module_name: light
submodule_name: regression
model_name: NNCondNetRegressor
| name           | type   | default   |   lower |   upper | transform             |
|----------------|--------|-----------|---------|---------|-----------------------|
| l1             | int    | 3         |     3   |   4     | transform_power_2_int |
| epochs         | int    | 4         |     3   |   7     | transform_power_2_int |
| batch_size     | int    | 4         |     4   |   5     | transform_power_2_int |
| act_fn         | factor | ReLU      |     0   |   5     | None                  |
| optimizer      | factor | SGD       |     0   |   2     | None                  |
| dropout_prob   | float  | 0.01      |     0   |   0.025 | None                  |
| lr_mult        | float  | 1.0       |     0.1 |  20     | None                  |
| patience       | int    | 2         |     2   |   3     | transform_power_2_int |
| batch_norm     | factor | 0         |     0   |   1     | None                  |
| initialization | factor | Default   |     0   |   4     | None                  |
```

```
spot_tuner = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
res = spot_tuner.run()
```

```
train_model result: {'val_loss': 24159.23828125, 'hp_metric': 24159.23828125}

train_model result: {'val_loss': 23455.20703125, 'hp_metric': 23455.20703125}

train_model result: {'val_loss': 25569.056640625, 'hp_metric': 25569.056640625}

train_model result: {'val_loss': 24069.580078125, 'hp_metric': 24069.580078125}
```

```
train_model result: {'val_loss': 22380.91015625, 'hp_metric': 22380.91015625}

train_model result: {'val_loss': 23910.111328125, 'hp_metric': 23910.111328125}

train_model result: {'val_loss': 23796.3828125, 'hp_metric': 23796.3828125}

train_model result: {'val_loss': 5523.42138671875, 'hp_metric': 5523.42138671875}

train_model result: {'val_loss': 22435.35546875, 'hp_metric': 22435.35546875}

train_model result: {'val_loss': 22645.388671875, 'hp_metric': 22645.388671875}

train_model result: {'val_loss': 7792.2529296875, 'hp_metric': 7792.2529296875}
spotpython tuning: 5523.42138671875 [----------] 3.04%

train_model result: {'val_loss': 21968.62890625, 'hp_metric': 21968.62890625}
spotpython tuning: 5523.42138671875 [##--------] 15.94%

train_model result: {'val_loss': 23941.822265625, 'hp_metric': 23941.822265625}
spotpython tuning: 5523.42138671875 [##--------] 20.39%

train_model result: {'val_loss': 23916.990234375, 'hp_metric': 23916.990234375}
spotpython tuning: 5523.42138671875 [##--------] 22.88%

train_model result: {'val_loss': 4589.27880859375, 'hp_metric': 4589.27880859375}
spotpython tuning: 4589.27880859375 [###-------] 29.52%

train_model result: {'val_loss': 5450.64453125, 'hp_metric': 5450.64453125}
spotpython tuning: 4589.27880859375 [####------] 35.93%

train_model result: {'val_loss': 11993.2900390625, 'hp_metric': 11993.2900390625}
spotpython tuning: 4589.27880859375 [####------] 39.97%

train_model result: {'val_loss': 18367.02734375, 'hp_metric': 18367.02734375}
spotpython tuning: 4589.27880859375 [#######---] 68.06%

train_model result: {'val_loss': 4835.94921875, 'hp_metric': 4835.94921875}
spotpython tuning: 4589.27880859375 [#######---] 74.87%

train_model result: {'val_loss': 2918.798828125, 'hp_metric': 2918.798828125}
spotpython tuning: 2918.798828125 [########--] 82.63%
```

```
train_model result: {'val_loss': 22200.45703125, 'hp_metric': 22200.45703125}
spotpython tuning: 2918.798828125 [#########-] 89.57%


train_model result: {'val_loss': nan, 'hp_metric': nan}


train_model result: {'val_loss': 5188.3837890625, 'hp_metric': 5188.3837890625}
spotpython tuning: 2918.798828125 [#########-] 94.34%


train_model result: {'val_loss': 20643.349609375, 'hp_metric': 20643.349609375}
spotpython tuning: 2918.798828125 [##########] 100.00% Done...

Experiment saved to CondNet_01_res.pkl
```

46.1. Looking at the Results

46.1.1. Tuning Progress

After the hyperparameter tuning run is finished, the progress of the hyperparameter tuning can be visualized with `spotpython`'s method `plot_progress`. The black points represent the performace values (score or metric) of hyperparameter configurations from the initial design, whereas the red points represents the hyperparameter configurations found by the surrogate model based optimization.

```
spot_tuner.plot_progress()
```

46.1.2. Tuned Hyperparameters and Their Importance

Results can be printed in tabular form.

```
from spotpython.utils.eda import print_res_table
print_res_table(spot_tuner)
```

```
| name           | type   | default   |   lower |   upper | tuned                  | transform
|----------------|--------|-----------|---------|---------|------------------------|-----------------
| l1             | int    | 3         |     3.0 |     4.0 | 4.0                    | transform_power_
| epochs         | int    | 4         |     3.0 |     7.0 | 7.0                    | transform_power_
| batch_size     | int    | 4         |     4.0 |     5.0 | 4.0                    | transform_power_
| act_fn         | factor | ReLU      |     0.0 |     5.0 | LeakyReLU              | None
| optimizer      | factor | SGD       |     0.0 |     2.0 | Adadelta               | None
| dropout_prob   | float  | 0.01      |     0.0 |   0.025 | 6.391858037193926e-08  | None
| lr_mult        | float  | 1.0       |     0.1 |    20.0 | 2.0155424752084863     | None
| patience       | int    | 2         |     2.0 |     3.0 | 2.0                    | transform_power_
| batch_norm     | factor | 0         |     0.0 |     1.0 | 0                      | None
| initialization | factor | Default   |     0.0 |     4.0 | kaiming_uniform        | None
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=1.0)
```

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
l1:  0.030319973009095193
epochs:  0.018848160830909
batch_size:  0.0171083730383338
act_fn:  0.017649082757451858
optimizer:  0.024579854097065482
dropout_prob:  5.236464260686369
lr_mult:  0.015481768141873367
patience:  0.022922948290482713
batch_norm:  100.0
initialization:  0.02277159394652154
```

46.1.3. Get the Tuned Architecture

```
import pprint
from spotpython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(spot_tuner)
pprint.pprint(config)
```

```
{'act_fn': LeakyReLU(),
 'batch_norm': False,
 'batch_size': 16,
 'dropout_prob': 6.391858037193926e-08,
 'epochs': 128,
 'initialization': 'kaiming_uniform',
 'l1': 16,
 'lr_mult': 2.0155424752084863,
 'optimizer': 'Adadelta',
 'patience': 4}
```

A. Introduction to Jupyter Notebook

Jupyter Notebook is a widely used tool in the Data Science community. It is easy to use and the produced code can be run per cell. This has a huge advantage, because with other tools e.g. (pycharm, vscode, etc.) the whole script is executed. This can be a time consuming process, especially when working with huge data sets.

A.1. Different Notebook cells

There are different cells that the notebook is currently supporting:

- code cells
- markdown cells
- raw cells

As a default, every cells in jupyter is set to "code"

A.1.1. Code cells

The code cells are used to execute the code. They are following the logic of the choosen kernel. Therefore, it is important to keep in mind which programming language is currently used. Otherwise one might yield an error because of the wrong syntax.

The code cells are executed my be **Run** button (can be found in the header of the notebook).

A.1.2. Markdown cells

The markdown cells are a usefull tool to comment the written code. Especially with the help of headers can the code be brought in a more readable format. If you are not familiar with the markdown syntax, you can find a usefull cheat sheet here: Markdown Cheat Sheeet

A.1.3. Raw cells

The "Raw NBConvert" cell type can be used to render different code formats into HTML or LaTeX by Sphinx. This information is stored in the notebook metadata and converted appropriately.

A.1.3.1. Usage

To select a desired format from within Jupyter, select the cell containing your special code and choose options from the following dropdown menus:

1. Select "Raw NBConvert"
2. Switch the Cell Toolbar to "Raw Cell Format" (The cell toolbar can be found under View)
3. Chose the appropriate "Raw NBConvert Format" within the cell

Data Science is fun

A.2. Install Packages

Because python is a heavily used programming language, there are many different packags that can make your life easier. Sadly, there are only a few standard packages that are already included in your python enviroment. If you have the need to install a new package in your enviroment, you can simply do that by exectuing the following code snippet in a **code cell**

```
!pip install numpy
```

- The *!* is used to run the cell as a shell command
- *pip* is package manager for python packages.
- *numpy* is the the package you want to install

Hint: It is often usefull to restart the kernel after installing a package, otherwise loading the package could lead to an error.

A.3. Load Packages

After successfully installing the package it is necessary to import them before you can work with them. The import of the packages is done in the following way:

```
import numpy as np
```

The imported packages are often abbreviated. This is because you need to specify where the function is coming from.

The most common abbreviations for data science packages are:

Table A.1.: Abbreviations for data science packages

Abbreviation	Package	Import
np	numpy	import numpy as np
pd	pandas	import pandas as pd
plt	matplotlib	import matplotlib.pyplot as plt
px	plotly	import plotly.exprss as px
tf	tensorflow	import tensorflow as tf
sns	seaborn	import seaborn as sns
dt	datetime	import datetime as dt
pkl	pickle	import pickle as pkl

A.4. Functions in Python

Because python is not using Semicolon's it is import to keep track of indentation in your code. The indentation works as a placeholder for the semicolons. This is especially important if your are defining loops, functions, etc. …

Example: We are defining a function that calculates the squared sum of its input parameters

```
def squared_sum(x,y):
    z = x**2 + y**2
    return z
```

If you are working with something that needs indentation, it will be already done by the notebook.

Hint: Keep in mind that is good practice to use the *return* parameter. If you are not using *return* and a function has multiple paramaters that you would like to return, it will only return the last one defined.

A.5. List of Useful Jupyter Notebook Shortcuts

Table A.2.: List of useful Jupyter Notebook Shortcuts

Function	Keyboard Shortcut	Menu Tools
Save notebook	Esc + s	File → Save and Checkpoint
Create new Cell	Esc + a (above), Esc + b (below)	Insert → Cell above; Insert → Cell below
Run Cell	Ctrl + enter	Cell → Run Cell
Copy Cell	c	Copy Key
Paste Cell	v	Paste Key
Interrupt Kernel	Esc + i i	Kernel → Interrupt
Restart Kernel	Esc + 0 0	Kernel → Restart

If you combine everything you can create beautiful graphics

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

# Generate 100 random data points along 3 dimensions
x, y, scale = np.random.randn(3, 100)
fig, ax = plt.subplots()

# Map each onto a scatterplot we'll create with Matplotlib
ax.scatter(x=x, y=y, c=scale, s=np.abs(scale)*500)
ax.set(title="Some random data, created with the Jupyter Notebook!")
plt.show()
```

Some random data, created with the Jupyter Notebook!
2
1
0
−1
−2
−2
−1
0
1
2

B. Git Introduction

B.1. Learning Objectives

In this learning unit, you will learn how to set up Git as a version control system for a project. The most important Git commands will be explained. You will learn how to track and manage changes to your projects with Git. Specifically:

- Initializing a repository: `git init`
- Ignoring files: `.gitignore`
- Adding files to the staging area: `git add`
- Checking status changes: `git status`
- Reviewing history: `git log`
- Creating a new branch: `git branch`
- Switching to the current branch: `git switch` and `git checkout`
- Merging two branches: `git merge`
- Resolving conflicts
- Reverting changes: `git revert`
- Uploading changes to GitLab: `git push`
- Downloading changes from GitLab: `git pull`
- Advanced: `git rebase`

B.2. Basics of Git

B.2.1. Initializing a Repository: `git init`

To set up Git as a version control system for your project, you need to initialize a new Git repository at the top-level folder, which is the working directory of your project. This is done using the `git init` command.

All files in this folder and its subfolders will automatically become part of the repository. Creating a Git repository is similar to adding an all-powerful passive observer of all things to your project. Git sits there, observes, and takes note of even the smallest changes, such as a single character in a file within a repository with hundreds of files. And it will tell you where these changes occurred if you forget. Once Git is initialized, it monitors all changes made within the working directory, and it tracks the history of events from that point forward. For this purpose, a historical timeline is created

for your project, referred to as a "branch," and the initial branch is named `main`. So, when someone says they are on the `main branch` or working on the `main branch`, it means they are in the historical main timeline of the project. The Git repository, often abbreviated as `repo`, is a virtual representation of your project, including its history and branches, a book, if you will, where you can look up and retrieve the entire history of the project: you work in your working directory, and the Git repository tracks and stores your work.

B.2.2. Ignoring Files: `.gitignore`

It's useful that Git watches and keeps an eye on everything in your project. However, in most projects, there are files and folders that you don't need or want to keep an eye on. These may include system files, local project settings, libraries with dependencies, and so on.

You can exclude any file or folder from your Git repository by including them in the `.gitignore` file. In the `.gitignore` file, you create a list of file names, folder names, and other items that Git should not track, and Git will ignore these items. Hence the name "gitignore." Do you want to track a file that you previously ignored? Simply remove the mention of the file in the gitignore file, and Git will start tracking it again.

B.2.3. Adding Changes to the Staging Area: `git add`

The interesting thing about Git as an all-powerful, passive observer of all things is that it's very passive. As long as you don't tell Git what to remember, it will passively observe the changes in the project folder but do nothing.

When you make a change to your project that you want Git to include in the project's history to take a snapshot of so you can refer back to it later, your personal checkpoint, if you will, you need to first stage the changes in the staging area. What is the staging area? The staging area is where you collect changes to files that you want to include in the project's history.

This is done using the `git add` command. You can specify which files you want to add by naming them, or you can add all of them using `-A`. By doing this, you're telling Git that you've made changes and want it to remember these particular changes so you can recall them later if needed. This is important because you can choose which changes you want to stage, and those are the changes that will eventually be transferred to the history.

Note: When you run `git add`, the changes are not transferred to the project's history. They are only transferred to the staging area.

Example B.1 (Example of git add from the beginning)**.**

```
# Create a new directory for your
# repository and navigate to that directory:

mkdir my-repo
cd my-repo

# Initialize the repository with git init:

git init

# Create a .gitignore file for Python code.
# You can use a template from GitHub:

curl https://raw.githubusercontent.com/github/gitignore/master/Python.gitignore -o .gitignore

# Add your files to the repository using git add:

git add .
```

This adds all files in the current directory to the repository, except for the files listed in the .gitignore file.

B.2.4. Transferring Changes to Memory: `git commit`

The power of Git becomes evident when you start transferring changes to the project history. This is done using the `git commit` command. When you run `git commit`, you inform Git that the changes in the staging area should be added to the history of the project so that they can be referenced or retrieved later.

Additionally, you can add a commit message with the `-m` option to explain what changes were made. So when you look back at the project history, you can see that you added a new feature.

`git commit` creates a snapshot, an image of the current state of your project at that specific time, and adds it to the branch you are currently working on.

As you work on your project and transfer more snapshots, the branch grows and forms a timeline of events. This means you can now look back at every transfer in the branch and see what your code looked like at that time.

You can compare any phase of your code with any other phase of your code to find errors, restore deleted code, or do things that would otherwise not be possible, such as resetting the project to a previous state or creating a new timeline from any point.

So how often should you add these commits? My rule of thumb is not to commit too often. It's better to have a Git repository with too many commits than one with too few commits.

Example B.2 (Continuing the example from above:)**.** After adding your files with `git add`, you can create a commit to save your changes. Use the `git commit` command with the `-m` option to specify your commit message:

```
git commit -m "My first commit message"
```

This creates a new commit with the added files and the specified commit message.

B.2.5. Check the Status of Your Repository: `git status`

If you're wondering what you've changed in your project since the last commit snapshot, you can always check the Git status. Git will list every modified file and the current status of each file.

This status can be either:

- Unchanged (`unmodified`), meaning nothing has changed since you last transferred it, or
- It's been changed (`changed`) but not staged (`staged`) to be transferred into the history, or
- Something has been added to staging (`staged`) and is ready to be transferred into the history.

When you run `git status`, you get an overview of the current state of your project.

Example B.3 (Continuing the example from above:)**.** The `git status` command displays the status of your working directory and the staging area. It shows you which files have been modified, which files are staged for commit, and which files are not yet being tracked:

```
git status
```

`git status` is a useful tool to keep track of your changes and ensure that you have added all the desired files for commit.

B.2.6. Review Your Repository's History: `git log`

Example B.4 (Continuing the example from above:)**.** You can view the history of your commits with the `git log` command. This command displays a list of all the commits in the current branch, along with information such as the author, date, and commit message:

```
git log
```

There are many options to customize the output of `git log`. For example, you can use the `--pretty` option to change the format of the output:

```
git log --pretty=oneline
```

This displays each commit in a single line.

B.3. Branches (Timelines)

B.3.1. Creating an Alternative Timeline: `git branch`

In the course of developing a project, you often reach a point where you want to add a new feature, but doing so might require changing the existing code in a way that could be challenging to undo later.

Or maybe you just want to experiment and be able to discard your work if the experiment fails. In such cases, Git allows you to create an alternative timeline called a `branch` to work in.

This new `branch` has its own name and exists in parallel with the `main branch` and all other branches in your project.

During development, you can switch between branches and work on different versions of your code concurrently. This way, you can have a stable codebase in the `main branch` while developing an experimental feature in a separate `branch`. When you switch from one `branch` to another, the code you're working on is automatically reset to the latest commit of the branch you're currently in.

If you're working in a team, different team members can work on their own branches, creating an entire universe of alternative timelines for your project. When features are completed, they can be seamlessly merged back into the `main branch`.

Example B.5 (Continuing the example from above:)**.** To create a new `branch`, you can use the `git branch` command with the name of the new `branch` as an argument:

```
git branch my-tests
```

B.3.2. The Pointer to the Current Branch: HEAD

How does Git know where you are on the timeline, and how can you keep track of your position?

You're always working at the tip (HEAD) of the currently active branch. The HEAD pointer points there quite literally. In a new project archive with just a single main branch and only new commits being added, HEAD always points to the latest commit in the main branch. That's where you are.

However, if you're in a repository with multiple branches, meaning multiple alternative timelines, HEAD will point to the latest commit in the branch you're currently working on.

B.3.3. Switching to an Alternative Timeline: git switch

As your project grows, and you have multiple branches, you need to be able to switch between these branches. This is where the switch command comes into play.

At any time, you can use the git switch command with the name of the branch you want to switch to, and HEAD moves from your current branch to the one you specified.

If you've made changes to your code before switching, Git will attempt to carry those changes over to the branch you're switching to. However, if these changes conflict with the target branch, the switch will be canceled.

To resolve this issue without losing your changes, return to the original branch, add and commit your recent changes, and then perform the switch.

B.3.4. Switching to an Alternative Timeline and Making Changes: git checkout

To switch between branches, you can also use the git checkout command. It works similarly to git switch for this purpose: you pass the name of the branch you want to switch to, and HEAD moves to the beginning of that branch.

But checkout can do more than just switch to another timeline. With git checkout, you can also move to any commit point in any timeline. In other words, you can travel back in time and work on code from the past.

To do this, use git checkout and provide the commit ID. This is an automatically generated, random combination of letters and numbers that identifies each commit.

You can retrieve the commit ID using `git log`. When you run `git log`, you get a list of all the commits in your repository, starting with the most recent ones.

When you use `git checkout` with an older commit ID, you check out a commit in the middle of a branch. This disrupts the timeline, as you're actively attempting to change history. Git doesn't want you to do that because, much like in a science fiction movie, altering the past might also alter the future. In our case, it would break the version control branch's coherence.

To prevent you from accidentally disrupting time and altering history, checking out an earlier commit in any branch results in the warning "Detached Head," which sounds rather ominous. The "Detached Head" warning is appropriate because it accurately describes what's happening. Git literally detaches the head from the branch and sets it aside.

Now, you're working outside of time in a space unbound to any timeline, which again sounds rather threatening but is perfectly fine in reality.

To continue working on this past code, all you need to do is reattach it to the timeline. You can use `git branch` to create a new branch, and the detached head will automatically attach to this new branch.

Instead of breaking the history, you've now created a new alternative timeline that starts in the past, allowing you to work safely. You can continue working on the branch as usual.

Example B.6 (Continuing the example from above:)**.** To switch to a new branch, you can use the `git checkout` command:

```
git checkout meine-tests
```

Now you're using the new branch and can make changes independently from the original branch.

B.3.5. The Difference Between `checkout` and `switch`

What is the difference between `git switch` and `git checkout`? `git switch` and `git checkout` are two different commands that both serve the purpose of switching between branches. You can use both to switch between branches, but they have an important distinction. `git switch` is a new command introduced with Git 2.23. `git checkout` is an older command that has existed since Git 1.6.0. So, `git switch` and `git checkout` have different origins. `git switch` was introduced to separate the purposes of `git checkout`. `git checkout` has two different purposes: 1. It can be used to switch between branches, and 2. It can be used to reset files to the state of the last commit.

Here's an example: In my project, I made a change since the last commit, but I haven't staged it yet. Then, I realized that I actually don't want this change. I want to reset the file to the state before the last commit. As long as I haven't committed my changes, I can do this with `git checkout` by targeting the specific file. So, if that file is named `main.js`, I can say: `git checkout main.js`. And the file will be reset to the state of the last commit, which makes sense. I'm checking out the file from the last commit.

But that's quite different from switching between the beginning of one branch to another. `git switch` and `git restore` were introduced to separate these two operations. `git switch` is for switching between branches, and `git restore` is for resetting the specified file to the state of the last commit. If you try to restore a file with `git switch`, it simply won't work. It's not intended for that. As I mentioned earlier, it's about separating concerns.

Example B.7 (Difference between `git switch` and `git checkout`)**.** Here's an example demonstrating how to initialize a repository and switch between branches:

```
# Create a new directory for your repository
# and navigate to that directory:
mkdir my-repo
cd my-repo

# Initialize the repository with git init:
git init

# Create a new branch with git branch:
git branch my-new-branch

# Switch to the new branch using git switch:
git switch my-new-branch

# Alternatively, you can also use git checkout
# to switch to the new branch:

git checkout my-new-branch
```

Both commands lead to the same result: You are now on the new branch.

B.4. Merging Branches and Resolving Conflicts

B.4.1. git `merge`: Merging Two Timelines

Git allows you to split your development work into as many branches or alternative timelines as you like, enabling you to work on many different versions of your code

simultaneously without losing or overwriting any of your work.

This is all well and good, but at some point, you need to bring those various versions of your code back together into one branch. That's where `git merge` comes in.

Consider an example where you have two branches, a `main branch` and an experimental branch called `experimental-branch`. In the experimental branch, there is a new feature. To merge these two branches, you set `HEAD` to the branch where you want to incorporate the code and execute `git merge` followed by the name of the branch you want to merge. `HEAD` is a special pointer that points to the current branch. When you run `git merge`, it combines the code from the branch associated with `HEAD` with the code from the branch specified by the branch name you provide.

```
# Initialize the repository
git init

# Create a new branch called "experimental-branch"
git branch experimental-branch

# Switch to the "experimental-branch"
git checkout experimental-branch

# Add the new feature here and
# make a commit
# ...

# Switch back to the "main" branch
git checkout main

# Perform the merge
git merge experimental-branch
```

During the merge, matching pieces of code in the branches overlap, and any new code from the branch being merged is added to the project. So now, the main branch also contains the code from the experimental branch, and the events of the two separate timelines have been merged into a single one. What's interesting is that even though the experimental branch was merged with the main branch, the last commit of the experimental branch remains intact, allowing you to continue working on the experimental branch separately if you wish.

B.4.2. Resolving Conflicts When Merging

Merging branches where there are no code changes at the same place in both branches is a straightforward process. It's also a rare process. In most cases, there will be

some form of conflict between the branches – the same code or the same code area has been modified differently in the different branches. Merging two branches with such conflicts will not work, at least not automatically.

In this case, Git doesn't know how to merge this code. So, when such a situation occurs, it's marked as a conflict, and the merging process is halted. This might sound more dramatic than it is. When you get a conflict warning, Git is saying there are two different versions here, and Git needs to know which one you want to keep. To help you figure out the conflict, Git combines all the code into a single file and automatically marks the conflicting code as the current change, which is the original code from the branch you're working on, or as the incoming change, which is the code from the file you're trying to merge.

To resolve this conflict, you'll edit the file to literally resolve the code conflict. This might mean accepting either the current or incoming change and discarding the other. It could mean combining both changes or something else entirely. It's up to you. So, you edit the code to resolve the conflict. Once you've resolved the conflict by editing the code, you add the new conflict-free version to the staging area with `git add` and then commit the merged code with `git commit`. That's how the conflict is resolved.

A merge conflict occurs when Git struggles to automatically merge changes from two different branches. This usually happens when changes were made to the same line in the same file in both branches. To resolve a merge conflict, you must manually edit the affected files and choose the desired changes. Git marks the conflict areas in the file with special markings like `<<<<<<<`, `=======`, and `>>>>>>>`. You can search for these markings and manually select the desired changes. After resolving the conflicts, you can add the changes with `git add` and create a new commit with `git commit` to complete the merge.

Example B.8.

```
# Perform the merge (this will cause a conflict)
git merge experimenteller-branch

# Open the affected file in an editor and manually resolve the conflicts
# ...

# Add the modified file
git add <filename>

# Create a new commit
git commit -m "Resolved conflicts"
```

B.4.3. git `revert`: Undoing Something

One of the most powerful features of any software tool is the "Undo" button. Make a mistake, press "Undo," and it's as if it never happened. However, that's not quite as simple when an all-powerful, passive observer is watching and recording your project's history. How do you undo something that you've added to the history without rewriting the history?

The answer is that you can overwrite the history with the `git reset` command, but that's quite risky and not a good practice.

A better solution is to work with the historical timeline and simply place an older version of your code at the top of the branch. This is done with `git revert`. To make this work, you need to know the commit ID of the commit you want to go back to.

The commit ID is a machine-generated set of random numbers and letters, also known as a hash. To get a list of all the commits in the repository, including the commit ID and commit message, you can run `git log`.

```
# Show the list of all operations in the repository
git log
```

By the way, it's a good idea to leave clear and informative commit messages for this reason. This way, you know what happened in your previous commits. Once you've found the commit you want to revert to, call that commit ID with `git revert`, and then the ID. This will create a new commit at the top of the branch with the code from the reference commit. To transfer the code to the branch, add a commit message and save it. Now, the last commit in your branch matches the commit you're reverting to, and your project's history remains intact.

Example B.9 (An example with `git revert`)**.**

```
# Initialize a new repository
git init

# Create a new file
echo "Hello, World" > file.txt

# Add the file to the repository
git add file.txt

# Create a new commit
git commit -m "First commit"

# Modify the file
```

```
echo "Goodbye, World" > file.txt

# Add the modified file
git add file.txt

# Create a new commit
git commit -m "Second commit"

# Use git log to find the commit ID of the second commit
git log

# Use git revert to undo the changes from the second commit
git revert <commit-id>
```

To download the `students` branch from the repository `git@git-ce.rwth-aachen.de:spotseven-lab/n` to your local machine, add a file, and upload the changes, you can follow these steps:

Example B.10 (An example with `git clone`, `git checkout`, `git add`, `git commit`, `git push`)**.**

```
# Clone the repository to your local machine:
git clone git@git-ce.rwth-aachen.de:spotseven-lab/numerische-mathematik-sommersemester

# Change to the cloned repository:
cd numerische-mathematik-sommersemester2023

# Switch to the students branch:
git checkout students

# Create the Test folder if it doesn't exist:
mkdir Test

# Create the Testdatei.txt file in the Test folder:
touch Test/Testdatei.txt

# Add the file with git add:
git add Test/Testdatei.txt

# Commit the changes with git commit:
git commit -m "Added Testdatei.txt"

# Push the changes with git push:
git push origin students
```

This will upload the changes to the server and update the students branch in the repository.

B.5. Downloading from GitLab

To download changes from a GitLab repository to your local machine, you can use the `git pull` command. This command downloads the latest changes from the specified remote repository and merges them with your local repository.

Here is an example:

Example B.11 (An example with `git pull`).

```
# Navigate to the local repository
# linked to the GitHub repository:
cd my-local-repository

# Make sure you are in the correct branch:
git checkout main

# Download the latest changes from GitHub:
git pull origin main
```

This downloads the latest changes from the main branch of the remote repository named "origin" and merges them with your local repository.

If there are conflicts between the downloaded changes and your local changes, you will need to resolve them manually before proceeding.

B.6. Advanced

B.6.1. git `rebase`: Moving the Base of a Branch

In some cases, you may need to "rewrite history." A common scenario is that you've been working on a new feature in a feature branch, and you realize that the work should have actually happened in the `main branch`.

To resolve this issue and make it appear as if the work occurred in the `main branch`, you can reset the experimental branch. "Rebase" literally means detaching the base of the experimental branch and moving it to the beginning of another branch, giving the branch a new base, thus "rebasing."

This operation is performed from the branch you want to "rebase." You use `git rebase` and specify the branch you want to use as the new base. If there are no conflicts between the experimental branch and the branch you want to rebase onto, this process happens automatically.

If there are conflicts, Git will guide you through the conflict resolution process for each commit from the rebase branch.

This may sound like a lot, but there's a good reason for it. You are literally rewriting history by transferring commits from one branch to another. To maintain the coherence of the new version history, there should be no conflicts within the commits. So, you need to resolve them one by one until the history is clean. It goes without saying that this can be a fairly labor-intensive process. Therefore, you should not use `git rebase` frequently.

Example B.12 (An example with `git rebase`)**.** `git rebase` is a command used to change the base of a branch. This means that commits from the branch are applied to a new base, which is usually another branch. It can be used to clean up the repository history and avoid merge conflicts.

Here is an example showing how to use `git rebase`:

- In this example, we initialize a new Git repository and create a new file. We add the file to the repository and make an initial commit. Then, we create a new branch called "feature" and switch to that branch. We make changes to the file in the feature branch and create a new commit.
- Then, we switch back to the main branch and make changes to the file again. We add the modified file and make another commit.
- To rebase the feature branch onto the main branch, we first switch to the feature branch and then use the `git rebase` command with the name of the main branch as an argument. This applies the commits from the feature branch to the main branch and changes the base of the feature branch.

```
# Initialize a new repository
git init
# Create a new file
echo "Hello World" > file.txt
# Add the file to the repository
git add file.txt
# Create an initial commit
git commit -m "Initial commit"
# Create a new branch called "feature"
git branch feature
# Switch to the "feature" branch
git checkout feature
```

```
# Make changes to the file in the "feature" branch
echo "Hello Feature World" > file.txt
# Add the modified file
git add file.txt
# Create a new commit in the "feature" branch
git commit -m "Feature commit"
# Switch back to the "main" branch
git checkout main
# Make changes to the file in the "main" branch
echo "Hello Main World" > file.txt
# Add the modified file
git add file.txt
# Create a new commit in the "main" branch
git commit -m "Main commit"
# Use git rebase to rebase the "feature" branch
# onto the "main" branch
git checkout feature
git rebase main
```

B.7. Exercises

In order to be able to carry out this exercise, we provide you with a functional working environment. This can be accessed here. You can log in using your GMID. If you do not have one, you can generate one here. Once you have successfully logged in to the server, you must open a terminal instance. You are now in a position to carry out the exercise.

Alternatively, you can also carry out the exercise locally on your computer, but then you will need to install git.

B.7.1. Create project folder

First create the `test-repo` folder via the command line and then navigate to this folder using the corresponding command.

B.8. Initialize repo

Now initialize the repository so that the future project, which will be saved in the `test-repo` folder, and all associated files are versioned.

B.8.1. Do not upload / ignore certain file types

In order to carry out this exercise, you must first download a file which you then have git ignore. To do this, download the current examination regulations for the Bachelor's degree program in Electrical Engineering using the following command `curl -o pruefungsordnung.pdf https://www.th-koeln.de/mam/downloads/deutsch/studium/studiengaen`

The PDF file has been stored in the root directory of your repo and you must now exclude it from being uploaded so that no changes to this file are tracked. Please note that not only this one PDF file should be ignored, but all PDF files in the repo.

B.8.2. Create file and stage it

In order to be able to commit a change later and thus make it traceable, it must first be staged. However, as we only have a PDF file so far, which is to be ignored by git, we cannot stage anything. Therefore, in this task, a file `test.txt` with some string as content is to be created and then staged.

B.8.3. Create another file and check status

To understand the status function, you should create the file `test2.txt` and then call the status function of git.

B.8.4. Commit changes

After the changes to the `test.txt` file have been staged and these are now to be transferred to the project process, they must be committed. Therefore, in this step you should perform a corresponding commit in the current branch with the message `test-commit`. Finally, you should also display the history of the commits.

B.8.5. Create a new branch and switch to it

In this task, you are to create a new branch with the name `change-text` in which you will later make changes. You should then switch to this branch.

B.8.6. Commit changes in the new branch

To be able to merge the new branch into the main branch later, you must first make changes to the `test.txt` file. To do this, open the file and simply change the character string in this file before saving the changes and closing the file. Before you now commit the file, you should reset the file to the status of the last commit for practice purposes and thus undo the change. After you have done this, open the file `test.txt` again and change the character string again before saving and closing the file. This time you should commit the file `test.txt` and then commit it with the message `test-commit2`.

B.8.7. Merge branch into main

After you have committed the change to the `test.txt` file, you should merge the `change-text` branch including the change into the main branch so that it is also available there.

B.8.8. Resolve merge conflict

To simulate a merge conflict, you must first change the content of the `test.txt` file before you commit the change. Then switch to the branch `change-text` and change the file `test.txt` there as well before you commit the change. Now you should try to merge the branch `change-text` into the main branch and solve the problems that occur in order to be able to perform the merge successfully.

C. Python Introduction

C.1. Recommendations

Beginner's Guide to Python

D. Documentation of the Sequential Parameter Optimization

```
import numpy as np
from math import inf
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
from scipy.optimize import shgo
from scipy.optimize import direct
from scipy.optimize import differential_evolution
import matplotlib.pyplot as plt
```

This document describes the `Spot` features. The official `spotpython` documentation can be found here: https://sequential-parameter-optimization.github.io/spotpython/.

D.1. An Initial Example

The `spotpython` package provides several classes of objective functions. We will use an analytical objective function, i.e., a function that can be described by a (closed) formula:

$$f(x) = x^2.$$

```
fun = Analytical().fun_sphere
```

```
x = np.linspace(-1,1,100).reshape(-1,1)
y = fun(x)
plt.figure()
plt.plot(x,y, "k")
plt.show()
```

```
from spotpython.utils.init import fun_control_init, design_control_init, surrogate_con
spot_1 = Spot(fun=fun,
                    fun_control=fun_control_init(
                         lower = np.array([-10]),
                         upper = np.array([100]),
                         fun_evals = 7,
                         fun_repeats = 1,
                         max_time = inf,
                         noise = False,
                         tolerance_x = np.sqrt(np.spacing(1)),
                         var_type=["num"],
                         infill_criterion = "y",
                         n_points = 1,
                         seed=123,
                         log_level = 50),
                    design_control=design_control_init(
                         init_size=5,
                         repeats=1),
                    surrogate_control=surrogate_control_init(
                         noise=False,
                         min_theta=-4,
                         max_theta=3,
                         n_theta=1,
                         model_optimizer=differential_evolution,
                         model_fun_evals=10000))
```

```
spot_1.run()
```

```
spotpython tuning: 2.0168953451588423 [#########-] 85.71%
spotpython tuning: 0.010160677392696235 [##########] 100.00% Done...

Experiment saved to 000_res.pkl

<spotpython.spot.spot.Spot at 0x10a1505f0>
```

D.2. Organization

`Spot` organizes the surrogate based optimization process in four steps:

1. Selection of the objective function: `fun`.
2. Selection of the initial design: `design`.
3. Selection of the optimization algorithm: `optimizer`.
4. Selection of the surrogate model: `surrogate`.

For each of these steps, the user can specify an object:

```
from spotpython.fun.objectivefunctions import Analytical
fun = Analytical().fun_sphere
from spotpython.design.spacefilling import SpaceFilling
design = SpaceFilling(2)
from scipy.optimize import differential_evolution
optimizer = differential_evolution
from spotpython.build.kriging import Kriging
surrogate = Kriging()
```

For each of these steps, the user can specify a dictionary of control parameters.

1. `fun_control`
2. `design_control`
3. `optimizer_control`
4. `surrogate_control`

Each of these dictionaries has an initialzaion method, e.g., `fun_control_init()`. The initialization methods set the default values for the control parameters.

> **!** Important:
>
> - The specification of an lower bound in `fun_control` is mandatory.

```
from spotpython.utils.init import fun_control_init, design_control_init, optimizer_con
fun_control=fun_control_init(lower=np.array([-1, -1]),
                             upper=np.array([1, 1]))
design_control=design_control_init()
optimizer_control=optimizer_control_init()
surrogate_control=surrogate_control_init()
```

D.3. The Spot Object

Based on the definition of the `fun`, `design`, `optimizer`, and `surrogate` objects, and their corresponding control parameter dictionaries, `fun_control`, `design_control`, `optimizer_control`, and `surrogate_control`, the `spot` object can be build as follows:

```
from spotpython.spot import Spot
spot_tuner = Spot(fun=fun,
                        fun_control=fun_control,
                        design_control=design_control,
                        optimizer_control=optimizer_control,
                        surrogate_control=surrogate_control)
```

D.4. Run

```
spot_tuner.run()
```

```
spotpython tuning: 1.5904060546935205e-05 [#######---] 73.33%
spotpython tuning: 1.5904060546935205e-05 [########--] 80.00%
spotpython tuning: 1.5904060546935205e-05 [#########-] 86.67%
spotpython tuning: 1.5904060546935205e-05 [#########-] 93.33%
spotpython tuning: 1.2084513018724136e-05 [##########] 100.00% Done...

Experiment saved to 000_res.pkl
```

```
<spotpython.spot.spot.Spot at 0x160a59610>
```

D.5. Print the Results

```
spot_tuner.print_results()
```

```
min y: 1.2084513018724136e-05
x0: -0.003294464459178357
x1: -0.0011095120305498227
```

```
[['x0', np.float64(-0.003294464459178357)],
 ['x1', np.float64(-0.0011095120305498227)]]
```

D.6. Show the Progress

```
spot_tuner.plot_progress()
```

D.7. Visualize the Surrogate

- The plot method of the `kriging` surrogate is used.
- Note: the plot uses the interval defined by the ranges of the natural variables.

```
spot_tuner.surrogate.plot()
```

D.8. Run With a Specific Start Design

To pass a specific start design, use the `X_start` argument of the `run` method.

```
spot_x0 = Spot(fun=fun,
                    fun_control=fun_control_init(
                        lower = np.array([-10]),
                        upper = np.array([100]),
                        fun_evals = 7,
                        fun_repeats = 1,
                        max_time = inf,
                        noise = False,
                        tolerance_x = np.sqrt(np.spacing(1)),
                        var_type=["num"],
                        infill_criterion = "y",
                        n_points = 1,
                        seed=123,
                        log_level = 50),
                    design_control=design_control_init(
                        init_size=5,
                        repeats=1),
```

```
                    surrogate_control=surrogate_control_init(
                        noise=False,
                        min_theta=-4,
                        max_theta=3,
                        n_theta=1,
                        model_optimizer=differential_evolution,
                        model_fun_evals=10000))
spot_x0.run(X_start=np.array([0.5, -0.5]))
spot_x0.plot_progress()
```

```
spotpython tuning: 2.0168953451588423 [#########-] 85.71%
spotpython tuning: 0.010160677392696235 [##########] 100.00% Done...

Experiment saved to 000_res.pkl
```

D.9. Init: Build Initial Design

```
from spotpython.design.spacefilling import SpaceFilling
from spotpython.build.kriging import Kriging
from spotpython.fun.objectivefunctions import Analytical
from spotpython.utils.init import fun_control_init
gen = SpaceFilling(2)
rng = np.random.RandomState(1)
lower = np.array([-5,-0])
upper = np.array([10,15])
fun = Analytical().fun_branin

fun_control = fun_control_init(sigma=0)
```

```
X = gen.scipy_lhd(10, lower=lower, upper = upper)
print(X)
y = fun(X, fun_control=fun_control)
print(y)
```

```
[[ 8.97647221 13.41926847]
 [ 0.66946019  1.22344228]
 [ 5.23614115 13.78185824]
 [ 5.6149825  11.5851384 ]
 [-1.72963184  1.66516096]
 [-4.26945568  7.1325531 ]
 [ 1.26363761 10.17935555]
 [ 2.88779942  8.05508969]
 [-3.39111089  4.15213772]
 [ 7.30131231  5.22275244]]
[128.95676449  31.73474356 172.89678121 126.71295908  64.34349975
  70.16178611  48.71407916  31.77322887  76.91788181  30.69410529]
```

D.10. Replicability

Seed

```
gen = SpaceFilling(2, seed=123)
X0 = gen.scipy_lhd(3)
gen = SpaceFilling(2, seed=345)
X1 = gen.scipy_lhd(3)
X2 = gen.scipy_lhd(3)
gen = SpaceFilling(2, seed=123)
X3 = gen.scipy_lhd(3)
X0, X1, X2, X3
```

```
(array([[0.77254938, 0.31539299],
        [0.59321338, 0.93854273],
        [0.27469803, 0.3959685 ]]),
 array([[0.78373509, 0.86811887],
        [0.06692621, 0.6058029 ],
        [0.41374778, 0.00525456]]),
 array([[0.121357  , 0.69043832],
        [0.41906219, 0.32838498],
        [0.86742658, 0.52910374]]),
 array([[0.77254938, 0.31539299],
```

```
[0.59321338, 0.93854273],
[0.27469803, 0.3959685 ]]))
```

D.11. Surrogates

D.11.1. A Simple Predictor

The code below shows how to use a simple model for prediction. Assume that only two (very costly) measurements are available:

1. f(0) = 0.5
2. f(2) = 2.5

We are interested in the value at $x_0 = 1$, i.e., $f(x_0 = 1)$, but cannot run an additional, third experiment.

```
from sklearn import linear_model
X = np.array([[0], [2]])
y = np.array([0.5, 2.5])
S_lm = linear_model.LinearRegression()
S_lm = S_lm.fit(X, y)
X0 = np.array([[1]])
y0 = S_lm.predict(X0)
print(y0)
```

```
[1.5]
```

Central Idea: Evaluation of the surrogate model `S_lm` is much cheaper (or / and much faster) than running the real-world experiment f.

D.12. Tensorboard Setup

D.12.1. Tensorboard Configuration

The `TENSORBOARD_CLEAN` argument can be set to `True` in the `fun_control` dictionary to archive the TensorBoard folder if it already exists. This is useful if you want to start a hyperparameter tuning process from scratch. If you want to continue a hyperparameter tuning process, set `TENSORBOARD_CLEAN` to `False`. Then the TensorBoard folder will not be archived and the old and new TensorBoard files will shown in the TensorBoard dashboard.

D.12.2. Starting TensorBoard

`TensorBoard` can be started as a background process with the following command, where `./runs` is the default directory for the TensorBoard log files:

```
tensorboard --logdir="./runs"
```

> **TENSORBOARD_PATH**
>
> The TensorBoard path can be printed with the following command (after a `fun_control` object has been created):
>
> ```
> from spotpython.utils.init import get_tensorboard_path
> get_tensorboard_path(fun_control)
> ```

D.13. Demo/Test: Objective Function Fails

SPOT expects `np.nan` values from failed objective function values. These are handled. Note: SPOT's counter considers only successful executions of the objective function.

```
import numpy as np
from spotpython.fun.objectivefunctions import Analytical
from spotpython.spot import Spot
import numpy as np
from math import inf
# number of initial points:
ni = 20
# number of points
n = 30

fun = Analytical().fun_random_error
fun_control=fun_control_init(
    lower = np.array([-1]),
    upper= np.array([1]),
    fun_evals = n,
    show_progress=False)
design_control=design_control_init(init_size=ni)

spot_1 = Spot(fun=fun,
                     fun_control=fun_control,
                     design_control=design_control)
```

```
# assert value error from the run method
try:
    spot_1.run()
except ValueError as e:
    print(e)
```

```
Experiment saved to 000_res.pkl
```

D.14. Handling Results: Printing, Saving, and Loading

The results can be printed with the following command:

```
spot_tuner.print_results(print_screen=False)
```

The tuned hyperparameters can be obtained as a dictionary with the following command:

```
from spotpython.hyperparameters.values import get_tuned_hyperparameters
get_tuned_hyperparameters(spot_tuner, fun_control)
```

The results can be saved and reloaded with the following commands:

```
from spotpython.utils.file import save_pickle, load_pickle
from spotpython.utils.init import get_experiment_name
experiment_name = get_experiment_name("024")
SAVE_AND_LOAD = False
if SAVE_AND_LOAD == True:
    save_pickle(spot_tuner, experiment_name)
    spot_tuner = load_pickle(experiment_name)
```

D.15. `spotpython` as a Hyperparameter Tuner

D.15.1. Modifying Hyperparameter Levels

`spotpython` distinguishes between different types of hyperparameters. The following types are supported:

- `int` (integer)
- `float` (floating point number)
- `boolean` (boolean)
- `factor` (categorical)

D.15.1.1. Integer Hyperparameters

Integer hyperparameters can be modified with the `set_int_hyperparameter_values()` [SOURCE] function. The following code snippet shows how to modify the `n_estimators` hyperparameter of a random forest model:

```
from spotriver.hyperdict.river_hyper_dict import RiverHyperDict
from spotpython.utils.init import fun_control_init
from spotpython.hyperparameters.values import set_int_hyperparameter_values
from spotpython.utils.eda import print_exp_table
fun_control = fun_control_init(
    core_model_name="forest.AMFRegressor",
    hyperdict=RiverHyperDict,
)
print("Before modification:")
print_exp_table(fun_control)
set_int_hyperparameter_values(fun_control, "n_estimators", 2, 5)
print("After modification:")
print_exp_table(fun_control)
```

```
Before modification:
| name            | type   |   default |   lower |   upper | transform             |
|-----------------|--------|-----------|---------|---------|-----------------------|
| n_estimators    | int    |         3 |     2   |       5 | transform_power_2_int |
| step            | float  |         1 |     0.1 |      10 | None                  |
| use_aggregation | factor |         1 |     0   |       1 | None                  |
Setting hyperparameter n_estimators to value [2, 5].
Variable type is int.
Core type is None.
Calling modify_hyper_parameter_bounds().
After modification:
| name            | type   |   default |   lower |   upper | transform             |
|-----------------|--------|-----------|---------|---------|-----------------------|
| n_estimators    | int    |         3 |     2   |       5 | transform_power_2_int |
| step            | float  |         1 |     0.1 |      10 | None                  |
| use_aggregation | factor |         1 |     0   |       1 | None                  |
```

D.15.1.2. Float Hyperparameters

Float hyperparameters can be modified with the `set_float_hyperparameter_values()` [SOURCE] function. The following code snippet shows how to modify the `step` hyperparameter of a hyperparameter of a Mondrian Regression Tree model:

```
from spotriver.hyperdict.river_hyper_dict import RiverHyperDict
from spotpython.utils.init import fun_control_init
from spotpython.hyperparameters.values import set_float_hyperparameter_values
from spotpython.utils.eda import print_exp_table
fun_control = fun_control_init(
    core_model_name="forest.AMFRegressor",
    hyperdict=RiverHyperDict,
)
print("Before modification:")
print_exp_table(fun_control)
set_float_hyperparameter_values(fun_control, "step", 0.2, 5)
print("After modification:")
print_exp_table(fun_control)
```

```
Before modification:
| name            | type   |   default |   lower |   upper | transform             |
|-----------------|--------|-----------|---------|---------|-----------------------|
| n_estimators    | int    |         3 |     2   |       5 | transform_power_2_int |
| step            | float  |         1 |     0.1 |      10 | None                  |
| use_aggregation | factor |         1 |     0   |       1 | None                  |
Setting hyperparameter step to value [0.2, 5].
Variable type is float.
Core type is None.
Calling modify_hyper_parameter_bounds().
After modification:
| name            | type   |   default |   lower |   upper | transform             |
|-----------------|--------|-----------|---------|---------|-----------------------|
| n_estimators    | int    |         3 |     2   |       5 | transform_power_2_int |
| step            | float  |         1 |     0.2 |       5 | None                  |
| use_aggregation | factor |         1 |     0   |       1 | None                  |
```

D.15.1.3. Boolean Hyperparameters

Boolean hyperparameters can be modified with the `set_boolean_hyperparameter_values()` [SOURCE] function. The following code snippet shows how to modify the `use_aggregation` hyperparameter of a Mondrian Regression Tree model:

```
from spotriver.hyperdict.river_hyper_dict import RiverHyperDict
from spotpython.utils.init import fun_control_init
from spotpython.hyperparameters.values import set_boolean_hyperparameter_values
from spotpython.utils.eda import print_exp_table
fun_control = fun_control_init(
    core_model_name="forest.AMFRegressor",
```

```
    hyperdict=RiverHyperDict,
)
print("Before modification:")
print_exp_table(fun_control)
set_boolean_hyperparameter_values(fun_control, "use_aggregation", 0, 0)
print("After modification:")
print_exp_table(fun_control)
```

```
Before modification:
| name            | type   |   default |   lower |   upper | transform             |
|-----------------|--------|-----------|---------|---------|-----------------------|
| n_estimators    | int    |         3 |     2   |       5 | transform_power_2_int |
| step            | float  |         1 |     0.1 |      10 | None                  |
| use_aggregation | factor |         1 |     0   |       1 | None                  |
Setting hyperparameter use_aggregation to value [0, 0].
Variable type is factor.
Core type is bool.
Calling modify_boolean_hyper_parameter_levels().
After modification:
| name            | type   |   default |   lower |   upper | transform             |
|-----------------|--------|-----------|---------|---------|-----------------------|
| n_estimators    | int    |         3 |     2   |       5 | transform_power_2_int |
| step            | float  |         1 |     0.1 |      10 | None                  |
| use_aggregation | factor |         1 |     0   |       0 | None                  |
```

D.15.1.4. Factor Hyperparameters

Factor hyperparameters can be modified with the `set_factor_hyperparameter_values()` [SOURCE] function. The following code snippet shows how to modify the `leaf_model` hyperparameter of a Hoeffding Tree Regressor model:

```
from spotriver.hyperdict.river_hyper_dict import RiverHyperDict
from spotpython.utils.init import fun_control_init
from spotpython.hyperparameters.values import set_factor_hyperparameter_values
from spotpython.utils.eda import print_exp_table
fun_control = fun_control_init(
    core_model_name="tree.HoeffdingTreeRegressor",
    hyperdict=RiverHyperDict,
)
print("Before modification:")
print_exp_table(fun_control)
set_factor_hyperparameter_values(fun_control, "leaf_model", ['LinearRegression',
```

```
                                                        'Perceptron'])
print("After modification:")
```

```
Before modification:
| name                    | type   | default           |   lower |    upper | transform               |
|-------------------------|--------|-------------------|---------|----------|-------------------------|
| grace_period            | int    | 200               |  10     | 1000     | None                    |
| max_depth               | int    | 20                |   2     |   20     | transform_power_2_int   |
| delta                   | float  | 1e-07             |   1e-08 |    1e-06 | None                    |
| tau                     | float  | 0.05              |   0.01  |    0.1   | None                    |
| leaf_prediction         | factor | mean              |   0     |    2     | None                    |
| leaf_model              | factor | LinearRegression  |   0     |    2     | None                    |
| model_selector_decay    | float  | 0.95              |   0.9   |    0.99  | None                    |
| splitter                | factor | EBSTSplitter      |   0     |    2     | None                    |
| min_samples_split       | int    | 5                 |   2     |   10     | None                    |
| binary_split            | factor | 0                 |   0     |    1     | None                    |
| max_size                | float  | 500.0             | 100     | 1000     | None                    |
| memory_estimate_period  | int    | 6                 |   3     |    8     | transform_power_10_int  |
| stop_mem_management     | factor | 0                 |   0     |    1     | None                    |
| remove_poor_attrs       | factor | 0                 |   0     |    1     | None                    |
| merit_preprune          | factor | 1                 |   0     |    1     | None                    |
After modification:
```

E. Datasets

E.1. The Diabetes Data Set

This section describes the `Diabetes` data set. This is a PyTorch Dataset for regression, which is derived from the `Diabetes` data set from `scikit-learn` (sklearn). Ten baseline variables, age, sex, body mass index, average blood pressure, and six blood serum measurements were obtained for each of n = 442 diabetes patients, as well as the response of interest, a quantitative measure of disease progression one year after baseline.

E.1.1. Data Exploration of the sklearn Diabetes Data Set

```
from sklearn.datasets import load_diabetes
from spotpython.plot.xy import plot_y_vs_X
data = load_diabetes()
X, y = data.data, data.target
plot_y_vs_X(X, y, nrows=5, ncols=2, figsize=(20, 15))
```

- Each of these 10 feature variables have been mean centered and scaled by the standard deviation times the square root of n_samples (i.e., the sum of squares of each column totals 1).
- `s3_hdl` shows a different behavior than the other features. It has a negative slope. `HDL` (high-density lipoprotein) cholesterol, sometimes called "good" cholesterol, absorbs cholesterol in the blood and carries it back to the liver. The liver then flushes it from the body. High levels of HDL cholesterol can lower your risk for heart disease and stroke.

E.1.2. Generating the PyTorch Data Set

`spotpython` provides a `Diabetes` class to load the diabetes data set. The `Diabetes` class is a subclass of `torch.utils.data.Dataset`. It loads the diabetes data set from `sklearn` and returns the data set as a `torch.utils.data.Dataset` object, so that features and targets can be accessed as `torch.tensors`. [CODE REFERENCE].

```
from spotpython.data.diabetes import Diabetes
data_set = Diabetes()
```

```
print(len(data_set))
print(data_set.names)
```

```
442
['age', 'sex', 'bmi', 'bp', 's1_tc', 's2_ldl', 's3_hdl', 's4_tch', 's5_ltg', 's6_glu']
```

E.2. The Friedman Drift Dataset

E.2.1. The Friedman Drift Dataset as Implemented in `river`

We will describe the Friedman synthetic dataset with concept drifts [SOURCE], see also Friedman (1991) and Ikonomovska, Gama, and Džeroski (2011). Each observation is composed of ten features. Each feature value is sampled uniformly in [0, 1]. Only the first five features are relevant. The target is defined by different functions depending on the type of the drift. Global Recurring Abrupt drift will be used, i.e., the concept drift appears over the whole instance space.

The target is defined by the following function:

$$y = 10\sin(\pi x_0 x_1) + 20(x_2 - 0.5)^2 + 10x_3 + 5x_4 + \epsilon,$$

where $\epsilon \sim \mathcal{N}(0, 1)$ is normally distributed noise.

If the Global Recurring Abrupt drift variant of the Friedman Drift dataset is used, the target function changes at two points in time, namely p_1 and p_2. At the first point, the concept changes to:

$$y = 10\sin(\pi x_3 x_5) + 20(x_1 - 0.5)^2 + 10x_0 + 5x_2 + \epsilon,$$

At the second point of drift the old concept reoccurs. This can be implemented as follows, see https://riverml.xyz/latest/api/datasets/synth/FriedmanDrift/:

```
def __iter__(self):
    rng = random.Random(self.seed)

    i = 0
    while True:
        x = {i: rng.uniform(a=0, b=1) for i in range(10)}
        y = self._global_recurring_abrupt_gen(x, i) + rng.gauss(mu=0, sigma=1)

        yield x, y
        i += 1
```

```
def _global_recurring_abrupt_gen(self, x, index: int):
    if index < self._change_point1 or index >= self._change_point2:
        # The initial concept is recurring
        return (
            10 * math.sin(math.pi * x[0] * x[1]) + 20 * (x[2] - 0.5) ** 2 + 10 * x[3]
        )
    else:
        # Drift: the positions of the features are swapped
        return (
            10 * math.sin(math.pi * x[3] * x[5]) + 20 * (x[1] - 0.5) ** 2 + 10 * x[0]
        )
```

spotpython requires the specification of a **train** and **test** data set. These data sets can be generated as follows:

```
from river.datasets import synth
import pandas as pd
import numpy as np
from spotriver.utils.data_conversion import convert_to_df

seed = 123
shuffle = True
n_train = 6_000
n_test = 4_000
n_samples = n_train + n_test
target_column = "y"

dataset = synth.FriedmanDrift(
   drift_type='gra',
   position=(n_train/4, n_train/2),
   seed=123
)

train = convert_to_df(dataset, n_total=n_train)
train.columns = [f"x{i}" for i in range(1, 11)] + [target_column]


dataset = synth.FriedmanDrift(
   drift_type='gra',
   position=(n_test/4, n_test/2),
   seed=123
)
test = convert_to_df(dataset, n_total=n_test)
test.columns = [f"x{i}" for i in range(1, 11)] + [target_column]
```

```
def plot_data_with_drift_points(data, target_column, n_train, title=""):
    indices = range(len(data))
    y_values = data[target_column]

    plt.figure(figsize=(10, 6))
    plt.plot(indices, y_values, label="y Value", color='blue')

    drift_points = [n_train / 4, n_train / 2]
    for dp in drift_points:
        plt.axvline(x=dp, color='red', linestyle='--', label=f'Drift Point at {int(dp)}')

    handles, labels = plt.gca().get_legend_handles_labels()
    by_label = dict(zip(labels, handles))
    plt.legend(by_label.values(), by_label.keys())

    plt.xlabel('Index')
    plt.ylabel('Target Value (y)')
    plt.title(title)
    plt.grid(True)
    plt.show()
```

```
plot_data_with_drift_points(train, target_column, n_train, title="Training Data with Drift Points")
```

```
plot_data_with_drift_points(test, target_column, n_train, title="Testing Data with Dri
```

E.2.2. The Friedman Drift Data Set from `spotpython`

A data generator for the Friedman Drift dataset is implemented in the `spotpython` package, see friedman.py. The `spotpython` version is a simplified version of the `river` implementation. The `spotPyton` version allows the generation of constant input values for the features. This is useful for visualizing the concept drifts. For the productive use the `river version` should be used.

Plotting the first 100 samples of the Friedman Drift dataset, we can not see the concept drifts at p_1 and p_2. Drift can be visualized by plotting the target values over time for constant features, e,g, if x_0 is set to 1 and all other features are set to 0. This is illustrated in the following plot.

```
from spotpython.data.friedman import FriedmanDriftDataset

def plot_friedman_drift_data(n_samples, seed, change_point1, change_point2, constant=1
    data_generator = FriedmanDriftDataset(n_samples=n_samples, seed=seed, change_point
    data = [data for data in data_generator]
    indices = [i for _, _, i in data]
    values = {f"x{i}": [] for i in range(5)}
```

```
    values["y"] = []
    for x, y, _ in data:
        for i in range(5):
            values[f"x{i}"].append(x[i])
        values["y"].append(y)

    plt.figure(figsize=(10, 6))
    for label, series in values.items():
        plt.plot(indices, series, label=label)
    plt.xlabel('Index')
    plt.ylabel('Value')
    plt.axvline(x=change_point1, color='k', linestyle='--', label='Drift Point 1')
    plt.axvline(x=change_point2, color='r', linestyle='--', label='Drift Point 2')
    plt.legend()
    plt.grid(True)
    plt.show()

plot_friedman_drift_data(n_samples=100, seed=42, change_point1=50, change_point2=75, constant=False)
plot_friedman_drift_data(n_samples=100, seed=42, change_point1=50, change_point2=75, constant=True)
```

E. Datasets

17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
Value
0
20
40
60
80
100
Index
x0
x1
x2
x3
x4
y
Drift Point 1
Drift Point 2

F. Using Slurm

F.1. Introduction

This chapter describes how to generate a `spotpython` configuration on a local machine and run the `spotpython` code on a remote machine using Slurm.

F.2. Prepare the Slurm Scripts on the Remote Machine

Two scripts are required to run the `spotpython` code on the remote machine:

- `startSlurm.sh` and
- `startPython.py`.

They should be saved in the same directory as the configuration (`pickle`) file. These two scripts must be generated only once and can be reused for different configurations.

The `startSlurm.sh` script is a shell script that contains the following code:

```
#!/bin/bash

### Vergabe von Ressourcen
#SBATCH --job-name=Test
#SBATCH --cpus-per-task=20
#SBATCH --gres=gpu:1
#SBATCH --time=48:00:00
#SBATCH --error=job.%J.err
#SBATCH --output=job.%J.out
#----
#SBATCH --partition=gpu

if [ -z "$1" ]; then
    echo "Usage: $0 <path_to_spot.pkl>"
    exit 1
fi
```

```
SPOT_PKL=$1

module load conda

### change to your conda environment with spotpython installed via
### pip install spotpython
conda activate spot312

python startPython.py "$SPOT_PKL"

exit
```

Save the code in a file named `startSlurm.sh` and copy the file to the remote machine via `scp`, i.e.,

```
scp startSlurm.sh user@144.33.22.1:
```

The `startPython.py` script is a Python script that contains the following code:

```
import argparse
import pickle
from spotpython.utils.file import load_and_run_spot_python_experiment
from spotpython.data.manydataset import ManyToManyDataset

# Uncomment the following if you want to use a custom model (python source code)
# import sys
# sys.path.insert(0, './userModel')
# import my_regressor
# import my_hyper_dict


def main(pickle_file):
    spot_tuner = load_and_run_spot_python_experiment(filename=pickle_file)

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Process a pickle file.')
    parser.add_argument('pickle_file', type=str, help='The path to the pickle file to

    args = parser.parse_args()
    main(args.pickle_file)
```

Save the code in a file named `startPython.py` and copy the file to the remote machine via `scp`, i.e.,

```
scp startPython.py user@144.33.22.1:
```

F.3. Generate a spotpython Configuration

The configuration can be generated on a local machine using the following command:

```
from spotpython.data.diabetes import Diabetes
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.fun.hyperlight import HyperLight
from spotpython.utils.init import (fun_control_init, surrogate_control_init, design_control_init)
from spotpython.spot import Spot
from spotpython.hyperparameters.values import set_hyperparameter, get_tuned_architecture
from math import inf
import torch
from torch.utils.data import TensorDataset
# generate data
num_samples = 100_000
input_dim = 100
X = torch.randn(num_samples, input_dim)  # random data for example
Y = torch.randn(num_samples, 1)  # random target for example
data_set = TensorDataset(X, Y)

PREFIX="42"


fun_control = fun_control_init(
    accelerator="gpu",
    devices="auto",
    num_nodes=1,
    num_workers=19,
    precision="32",
    strategy="auto",
    save_experiment=True,
    PREFIX=PREFIX,
    fun_evals=50,
    max_time=inf,
    data_set = data_set,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=input_dim,
    _L_out=1)
```

```
fun = HyperLight().fun

set_hyperparameter(fun_control, "optimizer", [ "Adadelta", "Adam", "Adamax"])
set_hyperparameter(fun_control, "l1", [5,10])
set_hyperparameter(fun_control, "epochs", [10,12])
set_hyperparameter(fun_control, "batch_size", [4,11])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.025])
set_hyperparameter(fun_control, "patience", [2,9])

design_control = design_control_init(init_size=10)

S = Spot(fun=fun,fun_control=fun_control, design_control=design_control)
```

The configuration is saved as a pickle-file that contains the full information. In our example, the filename is `42_exp.pkl`.

F.4. Copy the Configuration to the Remote Machine

You can copy the configuration to the remote machine using the `scp` command. The following command copies the configuration to the remote machine `144.33.22.1`:

```
scp 42_exp.pkl user@144.33.22.1:
```

F.5. Run the `spotpython` Code on the Remote Machine

Login on the remote machine and run the following command to start the `spotpython` code:

```
ssh user@144.33.22.1
# change this to your conda environment!
conda activate spot312
sbatch ./startSlurm.sh 42_exp.pkl
```

F.6. Copy the Results to the Local Machine

After the `spotpython` code has finished, you can copy the results back to the local machine using the `scp` command. The following command copies the results to the local machine:

```
scp user@144.33.22.1:42_res.pkl .
```

> **Experiment and Result Files**
>
> - `spotpython` generates two files:
> - `PREFIX_exp.pkl` (experiment file), which stores the information about running the experiment, and
> - `PREFIX_res.pkl` (result file), which stores the results of the experiment.

F.7. Analyze the Results on the Local Machine

The file `42_res.pkl` contains the results of the `spotpython` code. You can analyze the results on the local machine using the following code. Note: `PREFIX` is the same as in the previous steps, i.e., `"42"`.

```
from spotpython.utils.file import load_result
spot_tuner = load_result(PREFIX)
```

F.7.1. Visualizing the Tuning Progress

Now the `spot_tuner` object is loaded and you can analyze the results interactively.

```
spot_tuner.plot_progress(log_y=True, filename=None)
```

F.7.2. Design Table with Default and Tuned Hyperparameters

```
from spotpython.utils.eda import print_res_table
print_res_table(spot_tuner)
```

F.7.3. Plotting Important Hyperparameters

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

F.7.4. The Tuned Hyperparameters

```
get_tuned_architecture(spot_tuner)
```

G. Python Package Building

Introduction

This notebook will guide you through the process of creating a Python package.

- All examples can be found in the `userPackage` directory, see: userPackage

G.1. Create a Conda Environment

- `conda create -n userpackage python=3.12`
- `conda activate userpackage`
- Install the following packages:

```
python -m pip install build flake8 black mkdocs mkdocs-gen-files mkdocs-literate-nav mkdocs-section-
```

G.2. Download the User Package

The user package can be found in the `userPackage` directory, see: userPackage

G.3. Build the User Package

- cd into the `userPackage` directory and run the following command:
  - `./makefile.sh`
  - Alternatively, you can run the following commands:
    - `rm -f dist/userpackage*; python -m build; python -m pip install dist/userpackage*.tar.gz`
    - `python -m mkdocs build`

G.4. Open the Documentation of the User Package

- `mkdocs serve` to view the documentation

H. Solutions to Selected Exercises

> ⚠ Warning
>
> - Solutions are incomplete and need to be corrected!
> - They serve as a starting point for the final solution.

H.1. Data-Driven Modeling and Optimization

H.1.1. Histograms

Solution H.1 (Density Curve).

- We can calculate propabilities.
- We only need two parameters (the mean and the sd) to form the curve -> Store data more efficently
- Blanks can be filled

H.1.2. The Normal Distribution

Solution H.2 (TwoSDAnswer). 95%

Solution H.3 (OneSDAnswer). 68%

Solution H.4 (ThreeSDAnswer). 99,7%

Solution H.5 (DataRangeAnswer). 80 - 120

Solution H.6 (PeakHeightAnswer). low

H.1.3. The mean, the media, and the mode

H.1.4. The exponential distribution

H.1.5. Population and Estimated Parameters

Solution H.7 (ProbabilityAnswer). 50%

H.1.6. Calculating the Mean, Variance and Standard Deviation

Solution H.8 (MeanDifferenceAnswer). If we have all the data, μ is the population mean and x-bar is the sample mean. We don't have the full information.

Solution H.9 (EstimateMeanAnswer). Sum of the values divided by n.

Solution H.10 (SigmaSquaredAnswer). Variance

Solution H.11 (EstimatedSDAnswer). The same as the normal standard deviation, but using n-1.

Solution H.12 (VarianceDifferenceAnswer). n and $n-1$

Solution H.13 (ModelBenefitsAnswer).

- Approximation
- Prediction
- Understanding

Solution H.14 (SampleDefinitionAnswer). It's a subset of the data.

H.1.7. Hypothesis Testing and the Null-Hypothesis

Solution H.15 (RejectHypothesisAnswer). It means the evidence supports the alternative hypothesis, indicating that the null hypothesis is unlikely to be true.

Solution H.16 (NullHypothesisAnswer). It's a statement that there is no effect or no difference, and it serves as the default or starting assumption in hypothesis testing.

Solution H.17 (BetterDrugAnswer). By conducting experiments and statistical tests to compare the new drug's effectiveness against the current standard and demonstrating a significant improvement.

H.1.8. Alternative Hypotheses, Main Ideas

H.1.9. p-values: What they are and how to interpret them

Solution H.18 (PValueIntroductionAnswer). We can reject the null hypothesis. We can make a decision.

Solution H.19 (PValueRangeAnswer). It can only be between 0 and 1.

Solution H.20 (PValueRangeAnswer). It can only be between 0 and 1.

Solution H.21 (TypicalPValueAnswer). The chance that we wrongly reject the null hypothesis.

Solution H.22 (FalsePositiveAnswer). If we have a false-positive, we succeed in rejecting the null hypothesis. But in fact/reality, this is false -> False positive.

H.1.10. How to calculate p-values

Solution H.23 (CalculatePValueAnswer). Probability of specific result, probability of outcome with the same probability, and probability of events with smaller probability.

Solution H.24 (SDCalculationAnswer). 7 is the SD.

Solution H.25 (SidedPValueAnswer). If we are not interested in the direction of the change, we use the two-sided. If we want to know about the direction, the one-sided.

Solution H.26 (CoinTestAnswer). TBD

Solution H.27 (BorderPValueAnswer). TBD

Solution H.28 (OneSidedPValueCautionAnswer). If you look in the wrong direction, there is no change.

Solution H.29 (BinomialDistributionAnswer). TBD

H.1.11. p-hacking: What it is and how to avoid it

Solution H.30 (PHackingWaysAnswer).

- Performing repeats until you find one result with a small p-value -> false positive result.
- Increasing the sample size within one experiment when it is close to the threshold.

Solution H.31 (AvoidPHackingAnswer). Specify the number of repeats and the sample sizes at the beginning.

Solution H.32 (MultipleTestingProblemAnswer). TBD

H.1.12. Covariance

Solution H.33 (CovarianceDefinitionAnswer). Formula

Solution H.34 (CovarianceMeaningAnswer). Large values in the first variable result in large values in the second variable.

Solution H.35 (CovarianceVarianceRelationshipAnswer). Formula

Solution H.36 (HighCovarianceAnswer). No, size doesn't matter.

Solution H.37 (ZeroCovarianceAnswer). No relationship

Solution H.38 (NegativeCovarianceAnswer). Yes

Solution H.39 (NegativeVarianceAnswer). No

H.1.13. Pearson's Correlation

Solution H.40 (CorrelationValueAnswer). Recalculate

Solution H.41 (CorrelationRangeAnswer). From -1 to 1

Solution H.42 (CorrelationFormulaAnswer). Formula

H.1.14. Boxplots

Solution H.43 (UnderstandingStatisticalPower). It is the probability of correctly rejecting the null hypothesis.

Solution H.44 (DistributionEffectOnPower). Power analysis is not applicable.

Solution H.45 (IncreasingPower). By taking more samples.

Solution H.46 (PreventingPHacking). TBD

Solution H.47 (SampleSizeAndPower). The power will be low.

H.1.15. Power Analysis

Solution H.48 (MainFactorsAffectingPower). The overlap (distance of the two means) and sample sizes.

Solution H.49 (PowerAnalysisOutcome). The sample size needed.

Solution H.50 (RisksInExperiments). Few experiments lead to very low power, and many experiments might result in p-hacking.

Solution H.51 (StepsToPerformPowerAnalysis).

1. Select power
2. Select threshold for significance (alpha)
3. Estimate the overlap (done by the effect size)

H.1.16. The Central Limit Theorem

Solution H.52 (CentralLimitTheoremAnswer). TBD

H.1.17. Boxplots

Solution H.53 (MedianAnswer). The median.

Solution H.54 (BoxContentAnswer). 50% of the data.

H.1.18. R-squared

Solution H.55 (RSquaredFormulaAnswer). TBD

Solution H.56 (NegativeRSquaredAnswer). If you fit a line, no, but there are cases where it could be negative. However, these are usually considered useless.

Solution H.57 (RSquaredCalculationAnswer). TBD

H.1.18.1. The main ideas of fitting a line to data (The main ideas of least squares and linear regression.)

Solution H.58 (LeastSquaresAnswer). It is the calculation of the smallest sum of residuals when you fit a model to data.

H.1.19. Linear Regression

H.1.20. Multiple Regression

H.1.21. A Gentle Introduction to Machine Learning

Solution H.59 (RegressionVsClassificationAnswer). Regression involves predicting continuous values (e.g., temperature, size), while classification involves predicting discrete values (e.g., categories like cat, dog).

H.1.22. Maximum Likelihood

Solution H.60 (LikelihoodConceptAnswer). The distribution that fits the data best.

H.1.23. Probability is not Likelihood

Solution H.61 (ProbabilityVsLikelihoodAnswer). Likelihood: Finding the curve that best fits the data. Probability: Calculating the probability of an event given a specific curve.

H.1.24. Cross Validation

Solution H.62 (TrainVsTestDataAnswer). Training data is used to fit the model, while testing data is used to evaluate how well the model fits.

Solution H.63 (SingleValidationIssueAnswer). The performance might not be representative because the data may not be equally distributed between training and testing sets.

Solution H.64 (FoldDefinitionAnswer). TBD

Solution H.65 (LeaveOneOutValidationAnswer). Only one data point is used as the test set, and the rest are used as the training set.

H.1.25. The Confusion Matrix

Solution H.66 (ConfusionMatrixAnswer). TBD

H.1.26. Sensitivity and Specificity

Solution H.67 (SensitivitySpecificityAnswer1). TBD

Solution H.68 (SensitivitySpecificityAnswer2). TBD

H.1.27. Bias and Variance

Solution H.69 (BiasAndVarianceAnswer). TBD

H.1.28. Mutual Information

Solution H.70 (MutualInformationExampleAnswer). TBD

H.1.29. Principal Component Analysis (PCA)

Solution H.71 (WhatIsPCAAnswer). A dimension reduction technique that helps discover important variables.

Solution H.72 (screePlotAnswer). It shows how much variation is defined by the data.

Solution H.73 (LeastSquaresInPCAAnswer). No, in the first step it tries to maximize distances.

Solution H.74 (PCAStepsAnswer).

1. Calculate mean
2. Shift the data to the center of the coordinate system
3. Fit a line by maximizing the distances
4. Calculate the sum of squared distances
5. Calculate the slope
6. Rotate

Solution H.75 (EigenvaluePC1Answer). Formula (to be specified).

Solution H.76 (DifferencesBetweenPointsAnswer). No, because the first difference is measured on the PC1 scale and it is more important.

Solution H.77 (ScalingInPCAAnswer). Scaling by dividing by the standard deviation (SD).

Solution H.78 (DetermineNumberOfComponentsAnswer). TBD

Solution H.79 (LimitingNumberOfComponentsAnswer).

1. The dimension of the problem
2. Number of samples

H.1.30. t-SNE

Solution H.80 (WhyUseTSNEAnswer). For dimension reduction and picking out the relevant clusters.

Solution H.81 (MainIdeaOfTSNEAnswer). To reduce the dimensions of the data by reconstructing the relationships in a lower-dimensional space.

Solution H.82 (BasicConceptOfTSNEAnswer).

1. First, randomly arrange the points in a lower dimension

2. Decide whether to move points left or right, depending on distances in the original dimension
3. Finally, arrange points in the lower dimension similarly to the original dimension

Solution H.83 (TSNEStepsAnswer).

1. Project data to get random points
2. Set up a matrix of distances
3. Calculate the inner variances of the clusters and the Gaussian distribution
4. Do the same with the projected points
5. Move projected points so the second matrix gets more similar to the first matrix

H.1.31. K-means clustering

Solution H.84 (HowKMeansWorksAnswer).

1. Select the number of clusters
2. Randomly select distinct data points as initial cluster centers
3. Measure the distance between each point and the cluster centers
4. Assign each point to the nearest cluster
5. Repeat the process

Solution H.85 (QualityOfClustersAnswer). Calculate the within-cluster variation.

Solution H.86 (IncreasingKAnswer). If k is too high, each point would be its own cluster. If k is too low, you cannot see the structures.

H.1.32. DBSCAN

Solution H.87 (CorePointInDBSCANAnswer). A point that is close to at least k other points.

Solution H.88 (AddingVsExtendingAnswer). Adding means we add a point and then stop. Extending means we add a point and then look for other neighbors from that point.

Solution H.89 (OutliersInDBSCANAnswer). Points that are not core points and do not belong to existing clusters.

H.1.33. K-nearest neighbors

Solution H.90 (AdvantagesAndDisadvantagesOfKAnswer).

- k = 1: Noise can disturb the process because of possibly incorrect measurements of points.
- k = 100: The majority can be wrong for some groups. It is smoother, but there is less chance to discover the structure of the data.

H.1.34. Naive Bayes

Solution H.91 (NaiveBayesFormulaAnswer). TBD

Solution H.92 (CalculateProbabilitiesAnswer). TBD

H.1.35. Gaussian Naive Bayes

Solution H.93 (UnderflowProblemAnswer). Small values multiplied together can become smaller than the limits of computer memory, resulting in zero. Using logarithms (e.g., log(1/2) -> -1, log(1/4) -> -2) helps prevent underflow.

H.1.36. Trees

Solution H.94 (Tree Usage). Classication, Regression, Clustering

Solution H.95 (Tree Usage). TBD

Solution H.96 (Tree Feature Importance). The most important feature.

Solution H.97 (Regression Tree Limitations). High dimensions

Solution H.98 (Regression Tree Score). SSR + alpha * T

Solution H.99 (Regression Tree Alpha Value Small). The tree is more complex.

Solution H.100 (Regression Tree Increase Alpha Value). We get smaller trees

Solution H.101 (Regression Tree Pruning). Decreases the complexity of the tree to enhance performance and reduce overfitting

H.2. Machine Learning and Artificial Intelligence

H.2.1. Backpropagation

Solution H.102 (ChainRuleAndGradientDescentAnswer). Combination of the chain rule and gradient descent.

Solution H.103 (BackpropagationNamingAnswer). Because you start at the end and go backwards.

H.2.2. Gradient Descent

Solution H.104 (GradDescStepSize). learning rate x slope

Solution H.105 (GradDescIntercept). Old intercept - step size

Solution H.106 (GradDescIntercept). When the step size is small or after a certain number of steps

H.2.3. ReLU

Solution H.107 (Graph ReLU). Graph of ReLU function: f(x) = max(0, x)

H.2.4. CNNs

Solution H.108 (CNNImageRecognitionAnswer).

- too many features for input layer -> high memory consumption
- always shift in data
- it learns local informations and local correlations

Solution H.109 (CNNFiltersInitializationAnswer). The filter values in CNNs are randomly initialized and then trained and optimized through the process of backpropagation.

Solution H.110 (CNNFilterInitializationAnswer). The filter values in CNNs are initially set by random initialization. These filters undergo training via backpropagation, where gradients are computed and used to adjust the filter values to optimize performance.

Solution H.111 (GenNNStockPredictionAnswer). A limitation of using classical neural networks for stock market prediction is their reliance on fixed inputs. Stock market data is dynamic and requires models that can adapt to changing conditions over time.

H.2.5. RNN

Solution H.112 (RNNUnrollingAnswer). In the unrolling process of RNNs, the network is copied and the output from the inner loop is fed into the second layer of the copied network.

Solution H.113 (RNNReliabilityAnswer). RNNs sometimes fail to work reliably due to the vanishing gradient problem (where gradients are less than 1) and the exploding gradient problem (where gradients are greater than 1). Additionally, reliability issues arise because the network and the weights are copied during the unrolling process.

H.2.6. LSTM

Solution H.114 (LSTMSigmoidTanhAnswer). The sigmoid activation function outputs values between 0 and 1, making it suitable for probability determination, whereas the tanh activation function outputs values between -1 and 1.

Solution H.115 (LSTMSigmoidTanhAnswer). State how much of the long term memory should be used.

Solution H.116 (LSTMGatesAnswer). An LSTM network has three types of gates: the forget gate, the input gate, and the output gate. The forget gate decides what information to discard from the cell state, the input gate updates the cell state with new information, and the output gate determines what part of the cell state should be output.

Solution H.117 (LSTMLongTermInfoAnswer). Long-term information is used in the output gate of an LSTM network.

Solution H.118 (LSTMUpdateGatesAnswer). In the input and forget gates.

H.2.7. Pytorch/Lightning

Solution H.119 (PyTorchRequiresGradAnswer). In PyTorch, `requires_grad` indicates whether a tensor should be trained. If set to False, the tensor will not be trained.

H.2.8. Embeddings

Solution H.120 (NN STrings). No, they process numerical values.

Solution H.121 (Embedding Definition). Representation of a word as a vector.

Solution H.122 (Embedding Dimensions). We can model similarities.

H.2.9. Sequence to Sequence Models

Solution H.123 (LSTM). Because they are able to consider "far away" information.

Solution H.124 (Teacher Forcing). We need to force the correct words for the training.

Solution H.125 (Attention). Attention scores compute similarities for one input to the others.

H.2.10. Transformers

Solution H.126 (ChatGPT). Decoder only.

Solution H.127 (Translation). Encoder-Decoder structure.

Solution H.128 (Difference Encoder-Decoder and Decoder Only.).

- Encoder-Decoder: self-attention.
- Decoder only: masked self-attention.

Solution H.129 (Weights).

- a: Randomly
- b: Backpropagation

Solution H.130 (Order of Words). Positional Encoding

Solution H.131 (Relationship Between Words). Masked self-attention which looks at the previous tokens.

Solution H.132 (Masked Self Attention). It works by investigating how similar each word is to itself and all of the proceeding words in the sentence.

Solution H.133 (Softmax). Transformation to values between 0 and 1.

Solution H.134 (Softmax Output). We create two new numbers: Values – like K and Q with different weights. We scale these values by the percentage. -> we get the scaled V´s

Solution H.135 (V´s). Lastly, we sum these values together, which combine separate encodings for both words relative to their similarities to "is", are the masked-self-attention values for "is".

Solution H.136 (Residual Connections). They are bypasses, which combine the position encoded values with masked-self-attention values.

Solution H.137 (Generate Known Word in Sequence).

- Training
- Because it is a Decoder-Only transformer used for prediction and the calculations that you need.

Solution H.138 (Masked-Self-Attention Values and Bypass). We use a simple neural network with two inputs and five outputs for the vocabulary.

References

Abadi, Martin, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, et al. 2016. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems." *arXiv e-Prints*, March, arXiv:1603.04467.

Aggarwal, Charu, ed. 2007. *Data Streams – Models and Algorithms*. Springer-Verlag.

Arlot, Sylvain, Alain Celisse, et al. 2010. "A Survey of Cross-Validation Procedures for Model Selection." *Statistics Surveys* 4: 40–79.

Bartz, Eva, Thomas Bartz-Beielstein, Martin Zaefferer, and Olaf Mersmann, eds. 2022. *Hyperparameter Tuning for Machine and Deep Learning with R - A Practical Guide*. Springer.

Bartz-Beielstein, Thomas. 2023. "PyTorch Hyperparameter Tuning with SPOT: Comparison with Ray Tuner and Default Hyperparameters on CIFAR10." https://github.com/sequential-parameter-optimization/spotpython/blob/main/notebooks/14_spot_ray_hpt_torch_cifar10.ipynb.

———. 2024a. "Evaluation and Performance Measurement." In, edited by Eva Bartz and Thomas Bartz-Beielstein, 47–62. Singapore: Springer Nature Singapore.

———. 2024b. "Hyperparameter Tuning." In, edited by Eva Bartz and Thomas Bartz-Beielstein, 125–40. Singapore: Springer Nature Singapore.

———. 2024c. "Introduction: From Batch to Online Machine Learning." In *Online Machine Learning: A Practical Guide with Examples in Python*, edited by Eva Bartz and Thomas Bartz-Beielstein, 1–11. Singapore: Springer Nature Singapore. https://doi.org/10.1007/978-981-99-7007-0_1.

Bartz-Beielstein, Thomas, Jürgen Branke, Jörn Mehnen, and Olaf Mersmann. 2014. "Evolutionary Algorithms." *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 4 (3): 178–95.

Bartz-Beielstein, Thomas, and Martina Friese. 2011. "Sequential Parameter Optimization and Optimal Computational Budget Allocation for Noisy Optimization Problems." Cologne University of Applied Science, Faculty of Computer Science; Engineering Science.

Bartz-Beielstein, Thomas, and Lukas Hans. 2024. "Drift Detection and Handling." In *Online Machine Learning: A Practical Guide with Examples in Python*, edited by Eva Bartz and Thomas Bartz-Beielstein, 23–39. Singapore: Springer Nature Singapore. https://doi.org/10.1007/978-981-99-7007-0_3.

Bartz-Beielstein, Thomas, Christian Lasarczyk, and Mike Preuss. 2005. "Sequential Parameter Optimization." In *Proceedings 2005 Congress on Evolutionary Computation (CEC'05), Edinburgh, Scotland*, edited by B McKay et al., 773–80. Piscataway NJ: IEEE Press.

Bartz-Beielstein, Thomas, and Martin Zaefferer. 2022. "Hyperparameter Tuning Approaches." In *Hyperparameter Tuning for Machine and Deep Learning with R - A Practical Guide*, edited by Eva Bartz, Thomas Bartz-Beielstein, Martin Zaefferer, and Olaf Mersmann, 67–114. Springer.

Bifet, Albert. 2010. *Adaptive Stream Mining: Pattern Learning and Mining from Evolving Data Streams.* Vol. 207. Frontiers in Artificial Intelligence and Applications. IOS Press.

Bifet, Albert, and Ricard Gavaldà. 2007. "Learning from Time-Changing Data with Adaptive Windowing." In *Proceedings of the 2007 SIAM International Conference on Data Mining (SDM)*, 443–48.

———. 2009. "Adaptive Learning from Evolving Data Streams." In *Proceedings of the 8th International Symposium on Intelligent Data Analysis: Advances in Intelligent Data Analysis VIII*, 249–60. IDA '09. Berlin, Heidelberg: Springer-Verlag.

Bifet, Albert, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. 2010a. "MOA: Massive Online Analysis." *Journal of Machine Learning Research* 99: 1601–4.

———. 2010b. "MOA: Massive Online Analysis." *Journal of Machine Learning Research* 11: 1601–4.

Chen, Chun Hung. 2010. *Stochastic simulation optimization: an optimal computing budget allocation.* World Scientific.

Chen, Ricky T. Q., Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. 2018. "Neural Ordinary Differential Equations." *arXiv e-Prints*, June, arXiv:1806.07366.

Chollet, Francoise, and J. J. Allaire. 2018. *Deep Learning with Python.* Manning.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." *arXiv e-Prints*, October, arXiv:1810.04805.

Domingos, Pedro M., and Geoff Hulten. 2000. "Mining High-Speed Data Streams." In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Boston, MA, USA, August 20-23, 2000*, edited by Raghu Ramakrishnan, Salvatore J. Stolfo, Roberto J. Bayardo, and Ismail Parsa, 71–80. ACM.

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, et al. 2020. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale." *arXiv e-Prints*, October, arXiv:2010.11929.

Dredze, Mark, Tim Oates, and Christine Piatko. 2010. "We're Not in Kansas Anymore: Detecting Domain Changes in Streams." In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, 585–95.

Forrester, Alexander, András Sóbester, and Andy Keane. 2008. *Engineering Design via Surrogate Modelling.* Wiley.

Friedman, Jerome H. 1991. "Multivariate Adaptive Regression Splines." *The Annals of Statistics* 19 (1): 1–67.

Gaber, Mohamed Medhat, Arkady Zaslavsky, and Shonali Krishnaswamy. 2005. "Mining Data Streams: A Review." *SIGMOD Rec.* 34: 18–26.

Gama, João, Pedro Medas, Gladys Castillo, and Pedro Rodrigues. 2004. "Learning with Drift Detection." In *Advances in Artificial Intelligence – SBIA 2004*, edited

by Ana L. C. Bazzan and Sofiane Labidi, 286–95. Berlin, Heidelberg: Springer Berlin Heidelberg.

Gama, João, Raquel Sebastião, and Pedro Pereira Rodrigues. 2013. "On Evaluating Stream Learning Algorithms." *Machine Learning* 90 (3): 317–46.

Gramacy, Robert B. 2020. *Surrogates*. CRC press.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2017. *The Elements of Statistical Learning*. Second. Springer.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. "Deep Residual Learning for Image Recognition."

———. 2016. "Identity Mappings in Deep Residual Networks." *arXiv e-Prints*, March, arXiv:1603.05027.

Hoeglinger, Stefan, and Russel Pears. 2007. "Use of Hoeffding Trees in Concept Based Data Stream Mining." *2007 Third International Conference on Information and Automation for Sustainability*, 57–62.

Ikonomovska, Elena. 2012. "Algorithms for Learning Regression Trees and Ensembles on Evolving Data Streams." PhD thesis, Jozef Stefan International Postgraduate School.

Ikonomovska, Elena, João Gama, and Sašo Džeroski. 2011. "Learning Model Trees from Evolving Data Streams." *Data Mining and Knowledge Discovery* 23 (1): 128–68.

Jain, Sarthak, and Byron C. Wallace. 2019. "Attention is not Explanation." *arXiv e-Prints*, February, arXiv:1902.10186.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2014. *An Introduction to Statistical Learning with Applications in R*. 7th ed. Springer.

Keller-McNulty, Sallie, ed. 2004. *Statistical Analysis of Massive Data Streams: Proceedings of a Workshop*. Washington, DC: Committee on Applied; Theoretical Statistics, National Research Council; National Academies Press.

Kidger, Patrick. 2022. "On Neural Differential Equations." *arXiv e-Prints*, February, arXiv:2202.02435.

Kohavi, Ron. 1995. "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection." In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, 1137–43. IJCAI'95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Lewis, R M, V Torczon, and M W Trosset. 2000. "Direct search methods: Then and now." *Journal of Computational and Applied Mathematics* 124 (1–2): 191–207.

Li, Lisha, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2016. "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization." *arXiv e-Prints*, March, arXiv:1603.06560.

Lippe, Phillip. 2022. "UvA Deep Learning Tutorials." https://github.com/phlippe/uvadlc_notebooks/tree/master.

Liu, Liyuan, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. 2019. "On the Variance of the Adaptive Learning Rate and Beyond." *arXiv e-Prints*, August, arXiv:1908.03265.

Manapragada, Chaitanya, Geoffrey I. Webb, and Mahsa Salehi. 2018. "Extremely Fast Decision Tree." In *KDD' 2018 - Proceedings of the 24th ACM SIGKDD*

International Conference on Knowledge Discovery and Data Mining, edited by Chih-Jen Lin and Hui Xiong, 1953–62. United States of America: Association for Computing Machinery (ACM). https://doi.org/10.1145/3219819.3220005.

Masud, Mohammad, Jing Gao, Latifur Khan, Jiawei Han, and Bhavani M Thuraisingham. 2011. "Classification and Novel Class Detection in Concept-Drifting Data Streams Under Time Constraints." *IEEE Transactions on Knowledge and Data Engineering* 23 (6): 859–74.

Meignan, David, Sigrid Knust, Jean-Marc Frayet, Gilles Pesant, and Nicolas Gaud. 2015. "A Review and Taxonomy of Interactive Optimization Methods in Operations Research." *ACM Transactions on Interactive Intelligent Systems*, September.

Montiel, Jacob, Max Halford, Saulo Martiello Mastelini, Geoffrey Bolmier, Raphael Sourty, Robin Vaysse, Adil Zouitine, et al. 2021. "River: Machine Learning for Streaming Data in Python."

Mourtada, Jaouad, Stephane Gaiffas, and Erwan Scornet. 2019. "AMF: Aggregated Mondrian Forests for Online Learning." *arXiv e-Prints*, June, arXiv:1906.10529. https://doi.org/10.48550/arXiv.1906.10529.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, et al. 2011. "Scikit-Learn: Machine Learning in Python." *Journal of Machine Learning Research* 12: 2825–30.

Pontryagin. 1987. *Mathematical Theory of Optimal Processes*. Routledge.

Putatunda, Sayan. 2021. *Practical Machine Learning for Streaming Data with Python*. Springer.

Santner, T J, B J Williams, and W I Notz. 2003. *The Design and Analysis of Computer Experiments*. Berlin, Heidelberg, New York: Springer.

Street, W. Nick, and YongSeog Kim. 2001. "A Streaming Ensemble Algorithm (SEA) for Large-Scale Classification." In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 377–82. KDD '01. New York, NY, USA: Association for Computing Machinery.

Tay, Yi, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020. "Efficient Transformers: A Survey." *arXiv e-Prints*, September, arXiv:2009.06732.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. "Attention Is All You Need." *arXiv e-Prints*, June, 1–15.

Wiegreffe, Sarah, and Yuval Pinter. 2019. "Attention is not not Explanation." *arXiv e-Prints*, August, arXiv:1908.04626.

Index

batch_size, 563

correlation coefficient, 115

DataLoaders, 563

fun_sphere, 179

load_from_checkpoint(), 556
log(), 552

Maximum Likelihood Estimation, 361

on_train_epoch_end(), 552
on_validation_epoch_end(), 553

Pearson correlation coefficient, 115
predict_step(), 554

R-squared, 123

sample standard deviation, 363
sample variance, 363
save_hyperparameters(), 556
Sphere function, 179

test_step(), 554
training_step(), 551

validate(), 553
validation_step(), 553