Hyperparameter Tuning Cookbook

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

Thomas Bartz-Beielstein

Oct 12, 2024

Table of contents

Preface 3
Book Structure . . . 3
Software Used in this Book . . . 4
Citation . . . 4

I Optimization 6

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 . . . 16
2.7 The Big Picture: Combining all Variables . . . 17
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` **24**
3.1 Derivative-free Optimization Algorithms . . . 25
3.1.1 Nelder-Mead Simplex Algorithm . . . 25
3.1.2 Powell's Method . . . 26
3.2 Gradient-based optimization algorithms . . . 27
3.2.1 An Introductory Example: Broyden-Fletcher-Goldfarb-Shanno Algorithm (BFGS) . . . 27
3.2.2 Background and Basics for Gradient-based Optimization . . . 29
3.2.3 Gradient . . . 29
3.2.4 Jacobian Matrix . . . 29
3.2.5 Hessian Matrix . . . 30
3.2.6 Gradient for Optimization . . . 31
3.2.7 Newton Method . . . 34
3.2.8 BFGS-Algorithm . . . 35
3.2.9 Procedure: . . . 36
3.2.10 Visualization BFGS for Rosenbrock . . . 37
3.3 Gradient- and Hessian-based Optimization Algorithms . . . 37
3.3.1 Newton-Conjugate-Gradient Algorithm . . . 38
3.3.2 Trust-Region Newton-Conjugate-Gradient Algorithm . . . 38
3.3.3 Trust-Region Truncated Generalized Lanczos / Conjugate Gradient Algorithm . . . 38
3.4 Global Optimization . . . 38
3.4.1 Local vs Global Optimization . . . 38
3.4.2 Global Optimizers in SciPy . . . 39
3.4.3 Dual Annealing Optimization . . . 43
3.4.4 Differential Evolution . . . 48
3.4.5 Procedure . . . 48
3.4.6 DIRECT . . . 51
3.4.7 SHGO . . . 51
3.4.8 Basin-hopping . . . 52
3.5 Project: One-Mass Oscillator Optimization . . . 52
3.5.1 Introduction . . . 52
3.5.2 One-Mass Oscillator Model . . . 52
3.5.3 The Real-World Data . . . 53
3.5.4 Objective Function . . . 55
3.5.5 Results . . . 56
3.5.6 Tasks . . . 57
3.5.7 Task for the Project Work . . . 58
3.6 Jupyter Notebook . . . 58

4 Sequential Parameter Optimization: Using `scipy` Optimizers **59**
4.1 The Objective Function Branin . . . 59

4.2 The Optimizer . . . 60
4.2.1 TensorBoard . . . 61
4.3 Print the Results . . . 62
4.4 Show the Progress . . . 63
4.5 Exercises . . . 64
4.5.1 `dual_annealing` . . . 64
4.5.2 `direct` . . . 65
4.5.3 `shgo` . . . 67
4.5.4 `basinhopping` . . . 69
4.5.5 Performance Comparison . . . 71
4.6 Jupyter Notebook . . . 72

II Numerical Methods **73**

5 Introduction: Numerical Methods **74**
5.1 Response Surface Methods: What is RSM? . . . 74
5.1.1 Visualization: Problems in Practice . . . 77
5.1.2 RSM: Strategies . . . 77
5.1.3 RSM: Noise in the Empirical Model . . . 78
5.1.4 RSM: Natural and Coded Variables . . . 78
5.1.5 RSM Low-order Polynomials . . . 79
5.2 First-Order Models (Main Effects Model) . . . 79
5.2.1 First-Order Model Properties . . . 80
5.2.2 First-order Model with Interactions in python . . . 81
5.2.3 Observations: First-Order Model with Interactions . . . 82
5.3 Second-Order Models . . . 82
5.3.1 Second-Order Models: Properties . . . 83
5.3.2 Example: Stationary Ridge . . . 83
5.3.3 Observations: Second-Order Model (Ridge) . . . 84
5.3.4 Example: Rising Ridge . . . 85
5.3.5 Summary: Rising Ridge . . . 86
5.3.6 Falling Ridge . . . 86
5.3.7 Saddle Point . . . 86
5.3.8 Interpretation: Saddle Points . . . 87
5.3.9 Summary: Ridge Analysis . . . 87
5.4 General RSM Models . . . 88
5.4.1 Ordinary Least Squares . . . 88
5.5 Designs . . . 88
5.5.1 Different Designs . . . 88
5.6 RSM Experimentation . . . 89
5.6.1 First Step . . . 89
5.6.2 Second Step . . . 89

5.6.3 Third Step . . . 89
5.7 RSM: Review and General Considerations . . . 89
5.7.1 Historical Considerations about RSM . . . 90
5.7.2 Status Quo . . . 90
5.7.3 The Role of Statistics . . . 91
5.7.4 New RSM is needed: DACE . . . 91
5.8 Exercises . . . 92
5.9 Jupyter Notebook . . . 92

6 Kriging (Gaussian Process Regression) 93
6.1 DACE and RSM . . . 93
6.2 Background: Expectation, Mean, Standard Deviation . . . 94
6.2.1 Sample Mean . . . 94
6.2.2 Variance and Standard Deviation . . . 94
6.2.3 Standard Deviation . . . 95
6.2.4 Calculation of the Standard Deviation with Python . . . 95
6.2.5 The Empirical Standard Deviation . . . 95
6.2.6 The Argument "axis" . . . 96
6.3 Data Types and Precision in Python . . . 96
6.4 Distributions and Random Numbers in Python . . . 98
6.4.1 The Uniform Distribution . . . 99
6.4.2 The Normal Distribution . . . 100
6.4.3 Visualization of the Standard Deviation . . . 102
6.4.4 Standardization of Random Variables . . . 102
6.4.5 Realizations of a Normal Distribution . . . 102
6.4.6 The Multivariate Normal Distribution . . . 103
6.4.7 The Bivariate Normal Distribution with Mean Zero and Zero Covariances $\sigma_{12} = \sigma_{21} = 0$ . . . 107
6.4.8 The Bivariate Normal Distribution with Mean Zero and Negative Covariances $\sigma_{12} = \sigma_{21} = -4$ . . . 107
6.5 Cholesky Decomposition and Positive Definite Matrices . . . 107
6.6 Maximum Likelihood Estimation: Multivariate Normal Distribution . . . 110
6.7 Introduction to Gaussian Processes . . . 110
6.7.1 Gaussian Process Prior . . . 111
6.7.2 Covariance Function . . . 111
6.7.3 Construction of the Covariance Matrix . . . 113
6.7.4 Generation of Random Samples and Plotting the Realizations of the Random Function . . . 115
6.7.5 Properties of the 1d Example . . . 117
6.8 Kriging: Modeling Basics . . . 120
6.8.1 The Kriging Idea in a Nutshell . . . 120
6.8.2 The Kriging Basis Function . . . 120
6.8.3 The Correlation Coefficient . . . 121

6.8.4 Covariance Matrix and Correlation Matrix . . . 122
6.8.5 The Kriging Model . . . 123
6.8.6 Correlations . . . 124
6.8.7 The Condition Number . . . 128
6.8.8 MLE to estimate θ and p . . . 129
6.8.9 Tuning θ and p . . . 130
6.9 Kriging Prediction . . . 130
6.9.1 The Augmented Correlation Matrix . . . 130
6.9.2 Properties of the Predictor . . . 131
6.10 Kriging Example: Sinusoid Function . . . 131
6.10.1 Calculating the Correlation Matrix Ψ . . . 131
6.10.2 Computing the ψ Vector . . . 132
6.10.3 Predicting at New Locations . . . 133
6.10.4 Visualization . . . 133
6.11 Cholesky Example With Two Points . . . 134
6.11.1 Cholesky Decomposition . . . 134
6.11.2 Computation of the Inverse Matrix . . . 135
6.12 Jupyter Notebook . . . 136

7 Introduction to spotpython 137
7.1 Example: `Spot` and the Sphere Function . . . 137
7.1.1 The Objective Function: Sphere . . . 138
7.1.2 The `Spot` Method as an Optimization Algorithm Using a Surrogate Model 139
7.2 `Spot` Parameters: `fun_evals`, `init_size` and `show_models` . . . 141
7.3 Print the Results . . . 142
7.4 Show the Progress . . . 142
7.5 Visualizing the Optimization and Hyperparameter Tuning Process with TensorBoard . . . 143
7.6 Jupyter Notebook . . . 146

8 Multi-dimensional Functions 147
8.1 Example: `Spot` and the 3-dim Sphere Function . . . 147
8.1.1 The Objective Function: 3-dim Sphere . . . 147
8.1.2 Results . . . 148
8.1.3 A Contour Plot . . . 149
8.1.4 TensorBoard . . . 151
8.1.5 Conclusion . . . 152
8.2 Factorial Variables . . . 153
8.3 Exercises . . . 156
8.3.1 1. The Three Dimensional `fun_cubed` . . . 156
8.3.2 2. The Ten Dimensional `fun_wing_wt` . . . 157
8.3.3 3. The Three Dimensional `fun_runge` . . . 157
8.3.4 4. The Three Dimensional `fun_linear` . . . 157

8.3.5 5. The Two Dimensional Rosenbrock Function fun_rosen 158
8.4 Selected Solutions 158
8.4.1 Solution to Exercise Section 8.3.5: The Two-dimensional Rosenbrock Function fun_rosen 158
8.5 Jupyter Notebook 162

9 Isotropic and Anisotropic Kriging 163
9.1 Example: Isotropic Spot Surrogate and the 2-dim Sphere Function 163
9.1.1 The Objective Function: 2-dim Sphere 163
9.1.2 Results 164
9.2 Example With Anisotropic Kriging 165
9.2.1 Taking a Look at the theta Values 167
9.3 Exercises 168
9.3.1 1. The Branin Function fun_branin 168
9.3.2 2. The Two-dimensional Sin-Cos Function fun_sin_cos 169
9.3.3 3. The Two-dimensional Runge Function fun_runge 169
9.3.4 4. The Ten-dimensional Wing-Weight Function fun_wingwt 169
9.3.5 5. The Two-dimensional Rosenbrock Function fun_rosen 170
9.4 Selected Solutions 170
9.4.1 Solution to Exercise Section 9.3.5: The Two-dimensional Rosenbrock Function fun_rosen 170
9.5 Jupyter Notebook 176

10 Using sklearn Surrogates in spotpython 177
10.1 Example: Branin Function with spotpython's Internal Kriging Surrogate . . . 177
10.1.1 The Objective Function Branin 177
10.1.2 Running the surrogate model based optimizer Spot: 178
10.1.3 TensorBoard 179
10.1.4 Print the Results 180
10.1.5 Show the Progress and the Surrogate 180
10.2 Example: Using Surrogates From scikit-learn 181
10.2.1 GaussianProcessRegressor as a Surrogate 182
10.3 Example: One-dimensional Sphere Function With spotpython's Kriging 184
10.3.1 Results 189
10.4 Example: Sklearn Model GaussianProcess 190
10.5 Exercises 196
10.5.1 1. A decision tree regressor: DecisionTreeRegressor 196
10.5.2 2. A random forest regressor: RandomForestRegressor 196
10.5.3 3. Ordinary least squares Linear Regression: LinearRegression 196
10.5.4 4. Linear least squares with l2 regularization: Ridge 197
10.5.5 5. Gradient Boosting: HistGradientBoostingRegressor 197
10.5.6 6. Comparison of Surrogates 197

10.6 Selected Solutions . . . 198
10.6.1 Solution to Exercise Section 10.5.5: Gradient Boosting . . . 198
10.7 Jupyter Notebook . . . 213

11 Sequential Parameter Optimization: Gaussian Process Models 214
11.1 Gaussian Processes Regression: Basic Introductory `scikit-learn` Example . . 214
11.1.1 Train and Test Data . . . 214
11.1.2 Building the Surrogate With `Sklearn` . . . 215
11.1.3 Plotting the `Sklearn`Model . . . 215
11.1.4 The `spotpython` Version . . . 216
11.1.5 Visualizing the Differences Between the `spotpython` and the `sklearn` Model Fits . . . 217
11.2 Exercises . . . 218
11.2.1 `Schonlau Example Function` . . . 218
11.2.2 `Forrester Example Function` . . . 218
11.2.3 `fun_runge Function (1-dim)` . . . 219
11.2.4 `fun_cubed (1-dim)` . . . 220
11.2.5 The Effect of Noise . . . 220

12 Expected Improvement 221
12.1 Example: `Spot` and the 1-dim Sphere Function . . . 221
12.1.1 The Objective Function: 1-dim Sphere . . . 221
12.1.2 Results . . . 222
12.2 Same, but with EI as infill_criterion . . . 223
12.3 Non-isotropic Kriging . . . 225
12.4 Using `sklearn` Surrogates . . . 227
12.4.1 The spot Loop . . . 227
12.4.2 spot: The Initial Model . . . 228
12.4.3 Init: Build Initial Design . . . 229
12.4.4 Evaluate . . . 232
12.4.5 Build Surrogate . . . 232
12.4.6 A Simple Predictor . . . 232
12.5 Gaussian Processes regression: basic introductory example . . . 232
12.6 The Surrogate: Using scikit-learn models . . . 235
12.7 Additional Examples . . . 238
12.7.1 Optimize on Surrogate . . . 240
12.7.2 Evaluate on Real Objective . . . 240
12.7.3 Impute / Infill new Points . . . 240
12.8 Tests . . . 240
12.9 EI: The Famous Schonlau Example . . . 242
12.10EI: The Forrester Example . . . 244
12.11Noise . . . 247
12.12Cubic Function . . . 250

12.13 Modifying Lambda Search Space . . . 256

13 Handling Noise **258**
13.1 Example: `Spot` and the Noisy Sphere Function . . . 258
13.1.1 The Objective Function: Noisy Sphere . . . 258
13.1.2 Reproducibility: Noise Generation and Seed Handling . . . 260
13.2 spotpython's Noise Handling Approaches . . . 262
13.3 Print the Results . . . 267
13.4 Noise and Surrogates: The Nugget Effect . . . 268
13.4.1 The Noisy Sphere . . . 268
13.5 Exercises . . . 271
13.5.1 Noisy `fun_cubed` . . . 271
13.5.2 `fun_runge` . . . 271
13.5.3 `fun_forrester` . . . 271
13.5.4 `fun_xsin` . . . 272

14 Optimal Computational Budget Allocation in `Spot` **273**
14.1 Example: `Spot`, OCBA, and the Noisy Sphere Function . . . 273
14.1.1 The Objective Function: Noisy Sphere . . . 273
14.2 Print the Results . . . 279
14.3 Noise and Surrogates: The Nugget Effect . . . 280
14.3.1 The Noisy Sphere . . . 280
14.4 Exercises . . . 283
14.4.1 Noisy `fun_cubed` . . . 283
14.4.2 `fun_runge` . . . 283
14.4.3 `fun_forrester` . . . 283
14.4.4 `fun_xsin` . . . 284

15 Kriging with Varying Correlation-p **285**
15.1 Example: `Spot` Surrogate and the 2-dim Sphere Function . . . 285
15.1.1 The Objective Function: 2-dim Sphere . . . 285
15.1.2 Results . . . 286
15.2 Example With Modified p . . . 287
15.2.1 Taking a Look at the `p` Values . . . 289
15.3 Optimization of the `p` Values . . . 290
15.4 Optimization of Multiple `p` Values . . . 291
15.5 Exercises . . . 293
15.5.1 `fun_branin` . . . 293
15.5.2 `fun_sin_cos` . . . 294
15.5.3 `fun_runge` . . . 294
15.5.4 `fun_wingwt` . . . 294
15.6 Jupyter Notebook . . . 294

III Data-Driven Modeling and Optimization 295

16 Data-Driven Modeling and Optimization 296
16.1 StatQuest Videos . . . 296
16.1.1 June, 11th 2024 . . . 296
16.1.2 Mathematical Models . . . 297
16.1.3 Hypothesis Testing and the Null-Hypothesis . . . 298
16.1.4 June, 18th 2024 . . . 300
16.1.5 June, 25th 2024 . . . 303
16.1.6 t-SNE . . . 304
16.1.7 K-means clustering . . . 304
16.1.8 DBSCAN . . . 304
16.1.9 K-nearest neighbors . . . 304
16.1.10 Naive Bayes . . . 305
16.1.11 Gaussian Naive Bayes . . . 305
16.1.12 July, 2nd 2024 . . . 305
16.1.13 Additional Videos . . . 306
16.2 Introduction to Statistical Learning . . . 307
16.2.1 Opening Remarks and Examples . . . 307
16.3 Basics . . . 309
16.3.1 Histograms . . . 309
16.3.2 Probability Distributions . . . 312
16.3.3 Discrete Distributions . . . 314
16.4 Continuous Distributions . . . 317
16.4.1 Distribution functions: PDFs and CDFs . . . 317
16.4.2 Expectation (Continuous) . . . 317
16.4.3 Variance and Standard Deviation (Continuous) . . . 317
16.4.4 Uniform Distribution . . . 318
16.4.5 Normal Distribution . . . 318
16.4.6 The Mean, the Median, and the Mode . . . 321
16.4.7 The Exponential Distribution . . . 321
16.4.8 Population and Estimated Parameters . . . 322
16.4.9 Calculating the Mean, Variance, and Standard Deviation . . . 322
16.4.10 What is a Mathematical Model? . . . 322
16.4.11 Sampling from a Distribution . . . 322
16.4.12 Hypothesis Testing and the Null Hypothesis . . . 322
16.4.13 Alternative Hypotheses . . . 322
16.4.14 p-values: What They Are and How to Interpret Them . . . 322
16.4.15 How to Calculate p-values . . . 322
16.4.16 p-hacking: What It Is and How to Avoid It . . . 322
16.4.17 Covariance . . . 322
16.4.18 Pearson's Correlation . . . 322
16.4.19 Boxplots . . . 322

16.4.20 R-squared . . . 322
16.4.21 The Main Ideas of Fitting a Line to Data . . . 322
16.4.22 Linear Regression . . . 322
16.4.23 Multiple Regression . . . 322
16.5 Supervised Learning . . . 322
16.5.1 Statistical Learning and Regression . . . 324
16.5.2 Optimal Predictor . . . 325
16.5.3 Curse of Dimensionality and Parametric Models . . . 326
16.5.4 Assessing Model Accuracy and Bias-Variance Trade-off . . . 330
16.5.5 Classification Problems and K-Nearest Neighbors . . . 336
16.5.6 k-Nearest Neighbor Classification . . . 338
16.5.7 Minkowski Distance . . . 341
16.5.8 Unsuperivsed Learning: Classification . . . 341

IV Machine Learning and AI 346

17 Machine Learning and Artificial Intelligence 347
17.1 Jupyter Notebooks . . . 347
17.2 Videos . . . 347
17.2.1 June, 11th 2024 . . . 347
17.2.2 June, 18th 2024 . . . 347
17.2.3 June, 25th 2024 . . . 348
17.2.4 CNNs . . . 348
17.2.5 RNN . . . 349
17.2.6 LSTM . . . 349
17.2.7 Pytorch/Lightning . . . 349
17.2.8 July, 2nd 2024 . . . 350
17.2.9 Additional Lecture (July, 9th 2024)? . . . 350
17.2.10 Additional Videos . . . 351
17.2.11 All Videos in a Playlist . . . 351
17.3 The StatQuest Introduction to PyTorch . . . 352
17.3.1 Build a Simple Neural Network in PyTorch . . . 352
17.3.2 Use the Neural Network and Graph the Output . . . 354
17.3.3 Optimize (Train) a Parameter in the Neural Network and Graph the Output . . . 355
17.4 Build a Long Short-Term Memory unit by hand using PyTorch + Lightning . . 361
17.4.1 Train the LSTM unit and use Lightning and TensorBoard to evaluate: Part 1 - Getting Started . . . 366
17.4.2 Optimizing (Training) the Weights and Biases in the LSTM that we made by hand: Part 2 - Adding More Epochs without Starting Over . . 368
17.5 Using and optimzing the PyTorch LSTM, nn.LSTM() . . . 371

V Introduction to Hyperparameter Tuning 375

18 Hyperparameter Tuning 376
18.1 Structure of the Hyperparameter Tuning Chapters . . . 376
18.2 Goals of Hyperparameter Tuning . . . 376

VI Hyperparameter Tuning with Sklearn 378

19 HPT: sklearn 379
19.1 Introduction to sklearn . . . 379

20 HPT: sklearn SVC on Moons Data 380
20.1 Step 1: Setup . . . 380
20.2 Step 2: Initialization of the Empty `fun_control` Dictionary . . . 380
20.3 Step 3: SKlearn Load Data (Classification) . . . 381
20.4 Step 4: Specification of the Preprocessing Model . . . 383
20.5 Step 5: Select Model (`algorithm`) and `core_model_hyper_dict` . . . 384
20.6 Step 6: Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model` . . . 386
20.6.1 Modify hyperparameter of type numeric and integer (boolean) . . . 387
20.6.2 Modify hyperparameter of type factor . . . 387
20.6.3 Optimizers . . . 388
20.7 Step 7: Selection of the Objective (Loss) Function . . . 388
20.7.1 Predict Classes or Class Probabilities . . . 388
20.8 Step 8: Calling the SPOT Function . . . 389
20.8.1 The Objective Function . . . 389
20.8.2 Run the `Spot` Optimizer . . . 389
20.8.3 TensorBoard . . . 390
20.9 Step 9: Results . . . 392
20.10 Get Default Hyperparameters . . . 393
20.11 Get SPOT Results . . . 394
20.11.1 Plot: Compare Predictions . . . 394
20.11.2 Detailed Hyperparameter Plots . . . 396
20.11.3 Parallel Coordinates Plot . . . 402
20.11.4 Plot all Combinations of Hyperparameters . . . 402

VII Hyperparameter Tuning with River 403

21 HPT: River 404
21.1 Introduction to River . . . 404

22 Simplifying Hyperparameter Tuning in Online Machine Learning—The spotRiver-GUI 405
22.1 Introduction . . . 405
22.2 Installation and Starting . . . 406
22.2.1 Installation . . . 406
22.2.2 Starting the GUI . . . 407
22.3 Binary Classification . . . 407
22.3.1 Binary Classification Options . . . 407
22.3.2 Experiment Options . . . 410
22.3.3 Evaluation Options . . . 411
22.3.4 Online Machine Learning Model Options . . . 413
22.4 Regression . . . 415
22.5 Showing the Data . . . 415
22.6 Saving and Loading . . . 420
22.6.1 Saving the Experiment . . . 420
22.6.2 Loading an Experiment . . . 420
22.7 Running a New Experiment . . . 421
22.7.1 Starting and Stopping Tensorboard . . . 421
22.8 Performing the Analysis . . . 422
22.9 Summary and Outlook . . . 426
22.10 Appendix . . . 427
22.10.1 Adding new Tasks . . . 427

23 `river` Hyperparameter Tuning: Hoeffding Tree Regressor with Friedman Drift Data 428
23.1 The Friedman Drift Data Set . . . 428
23.2 Setup . . . 429
23.2.1 General Experiment Setup . . . 429
23.2.2 Data Setup . . . 430
23.2.3 Evaluation Setup . . . 431
23.2.4 River-Specific Setup . . . 431
23.2.5 Model Setup . . . 432
23.2.6 Objective Function Setup . . . 432
23.2.7 Surrogate Model Setup . . . 433
23.2.8 Summary: Setting up the Experiment . . . 433
23.2.9 Run the `Spot` Optimizer . . . 434
23.3 Using the `spotgui` . . . 435
23.4 Results . . . 436
23.5 Performance of the Model with Default Hyperparameters . . . 438
23.5.1 Get Default Hyperparameters and Fit the Model . . . 438
23.5.2 Evaluate the Model with Default Hyperparameters . . . 438
23.5.3 Show Predictions of the Model with Default Hyperparameters . . . 439
23.6 Get SPOT Results . . . 440

23.7 Visualize Regression Trees . . . 443
23.7.1 Spot Model . . . 444
23.8 Detailed Hyperparameter Plots . . . 445
23.9 Parallel Coordinates Plots . . . 446

24 The Friedman Drift Data Set 448
24.1 Setup . . . 449
24.1.1 Select a User Hyperdictionary . . . 449
24.2 Modify `hyper_dict` Hyperparameters for the Selected Algorithm aka `core_model` 450
24.2.1 Run the `Spot` Optimizer . . . 451
24.3 Results . . . 452
24.4 Performance of the Model with Default Hyperparameters . . . 454
24.4.1 Get Default Hyperparameters and Fit the Model . . . 454
24.4.2 Evaluate the Model with Default Hyperparameters . . . 454
24.4.3 Show Predictions of the Model with Default Hyperparameters . . . 455
24.5 Get SPOT Results . . . 456
24.6 Detailed Hyperparameter Plots . . . 459
24.7 Parallel Coordinates Plots . . . 460

VIIIHyperparameter Tuning with PyTorch Lightning 461

25 HPT PyTorch Lightning: Data 462
25.1 Setup . . . 462
25.2 Initialization of the `fun_control` Dictionary . . . 463
25.3 Loading the Diabetes Data Set . . . 463
25.3.1 Data Set and Data Loader . . . 463
25.3.2 Preparing Training, Validation, and Test Data . . . 464
25.3.3 Dataset for spotpython . . . 466
25.4 The LightDataModule . . . 467
25.4.1 The `prepare_data()` Method . . . 467
25.4.2 The `setup()` Method . . . 467
25.4.3 The `train_dataloader()` Method . . . 471
25.4.4 The `val_dataloader()` Method . . . 472
25.4.5 The `test_dataloader()` Method . . . 473
25.4.6 The `predict_dataloader()` Method . . . 474
25.5 Using the `LightDataModule` in the `train_model()` Method . . . 475
25.6 Further Information . . . 477
25.6.1 Preprocessing . . . 477

26 Hyperparameter Tuning with `spotpython` and `PyTorch` Lightning for the Diabetes Data Set **478**
26.1 Looking at the Results . . . 480
26.1.1 Tuning Progress . . . 480
26.1.2 Tuned Hyperparameters and Their Importance . . . 480
26.1.3 Get the Tuned Architecture . . . 480
26.2 Cross Validation With Lightning . . . 482
26.3 Extending the Basic Setup . . . 482
26.3.1 General Experiment Setup . . . 482
26.3.2 Data Setup . . . 483
26.3.3 Objective Function `fun` . . . 483
26.3.4 Core-Model Setup . . . 483
26.3.5 Hyperdict Setup . . . 483
26.3.6 Other Settings . . . 483
26.4 Tensorboard . . . 484
26.5 Loading the Saved Experiment and Getting the Hyperparameters of the Tuned Model . . . 484
26.6 Using the `spotgui` . . . 484
26.7 Summary . . . 484

27 Hyperparameter Tuning with PyTorch Lightning and User Data Sets **486**
27.1 Loading a User Specified Data Set . . . 486
27.2 Summary . . . 491

28 Hyperparameter Tuning with PyTorch Lightning and User Models **492**
28.1 Using a User Specified Model . . . 492
28.2 Details . . . 495
28.2.1 Model Setup . . . 495
28.2.2 The `my_hyper_dict.py` File . . . 495
28.2.3 The `my_hyper_dict.json` File . . . 496
28.2.4 The `my_regressor.py` File . . . 498
28.3 Connection with the LightDataModule . . . 503
28.3.1 The `prepare_data()` Method . . . 503
28.3.2 The `setup()` Method . . . 503
28.3.3 The `train_dataloader()` Method . . . 508
28.3.4 The `val_dataloader()` Method . . . 509
28.3.5 The `test_dataloader()` Method . . . 511
28.3.6 The `predict_dataloader()` Method . . . 512
28.4 Using the `LightDataModule` in the `train_model()` Method . . . 513
28.5 The Last Connection: The `HyperLight` Class . . . 514
28.6 Further Information . . . 515
28.6.1 Preprocessing . . . 515

29 Hyperparameter Tuning with PyTorch Lightning: Physics Informed Neural Networks 516
29.1 The Ground Truth Model . . . 516
29.2 Required Files . . . 518
29.2.1 The New `pinn_hyperdict.py` File . . . 519
29.2.2 The New `pinn_regressor.py` File . . . 519
29.2.3 The New `pinn_hyperdict.json` File . . . 527

30 Explainable AI with SpotPython and Pytorch 528

31 HPT PyTorch Lightning Transformer: Introduction 560
31.1 Transformer Basics . . . 560
31.1.1 Embedding . . . 560
31.1.2 Attention . . . 561
31.1.3 Self-Attention . . . 563
31.1.4 Masked Self-Attention . . . 563
31.1.5 Generation of Outputs . . . 563
31.1.6 End-Of-Sequence-Token . . . 564
31.2 Details of the Implementation . . . 564
31.2.1 Dot Product Attention . . . 566
31.2.2 Scaled Dot Product Attention . . . 567
31.3 Example: Transformer in Lightning . . . 568
31.3.1 Downloading the Pretrained Models . . . 569
31.3.2 The Transformer Architecture . . . 570
31.3.3 Attention Mechanism . . . 570
31.3.4 Multi-Head Attention . . . 572
31.3.5 Permutation Equivariance . . . 574
31.3.6 Transformer Encoder . . . 575
31.3.7 Layer Normalization and Feed-Forward Network . . . 576
31.3.8 Positional Encoding . . . 579
31.3.9 Learning Rate Warm-up . . . 582
31.3.10 PyTorch Lightning Module . . . 584
31.4 Experiment: Sequence to Sequence . . . 587
31.4.1 Dataset and Data Loaders . . . 587
31.4.2 The Reverse Predictor Class . . . 588
31.4.3 Gradient Clipping . . . 589
31.4.4 Implementation of the Lightning Trainer . . . 590
31.4.5 Training the Model . . . 590
31.5 Visualizing Attention Maps . . . 591
31.6 Conclusion . . . 593
31.7 Additional Considerations . . . 594
31.7.1 Complexity and Path Length . . . 594
31.8 Further Reading . . . 595

32 Hyperparameter Tuning of a Transformer Network with PyTorch Lightning 596
32.1 Basic Setup . . . 596
32.2 Looking at the Results . . . 600
32.2.1 Tuning Progress . . . 600
32.2.2 Tuned Hyperparameters and Their Importance . . . 601
32.3 Hyperparameter Considerations . . . 601
32.3.1 Important: Constraints and Interconnections: . . . 602
32.3.2 Practical Considerations: . . . 603
32.4 Summary . . . 604

33 Saving and Loading 605
33.1 Overview . . . 605
33.2 spotpython: Saving and Loading Optimization Experiments . . . 605
33.3 spotpython as an Optimizer . . . 605
33.4 spotpython as a Hyperparameter Tuner . . . 609
33.4.1 The Diabetes Data Set . . . 609
33.5 Saving and Loading PyTorch Lightning Models . . . 616
33.5.1 Get the Tuned Architecture . . . 616
33.5.2 Load a Model from Checkpoint . . . 617
33.6 Converting a Lightning Model to a Plain Torch Model . . . 626
33.6.1 The Function `get_removed_attributes_and_base_net` . . . 626
33.6.2 An Example how to use the Plain Torch Net . . . 629

Appendices 632

A Introduction to Jupyter Notebook 632
A.1 Different Notebook cells . . . 632
A.1.1 Code cells . . . 632
A.1.2 Markdown cells . . . 632
A.1.3 Raw cells . . . 633
A.2 Install Packages . . . 633
A.3 Load Packages . . . 634
A.4 Functions in Python . . . 634
A.5 List of Useful Jupyter Notebook Shortcuts . . . 635

B Git Introduction 637
B.1 Learning Objectives . . . 637
B.2 Basics of Git . . . 637
B.2.1 Initializing a Repository: `git init` . . . 637
B.2.2 Ignoring Files: `.gitignore` . . . 638
B.2.3 Adding Changes to the Staging Area: `git add` . . . 638
B.2.4 Transferring Changes to Memory: `git commit` . . . 639

B.2.5 Check the Status of Your Repository: `git status` . . . 640
B.2.6 Review Your Repository's History: `git log` . . . 641
B.3 Branches (Timelines) . . . 641
B.3.1 Creating an Alternative Timeline: `git branch` . . . 641
B.3.2 The Pointer to the Current Branch: `HEAD` . . . 642
B.3.3 Switching to an Alternative Timeline: `git switch` . . . 642
B.3.4 Switching to an Alternative Timeline and Making Changes: `git checkout` 642
B.3.5 The Difference Between `checkout` and `switch` . . . 643
B.4 Merging Branches and Resolving Conflicts . . . 645
B.4.1 git `merge`: Merging Two Timelines . . . 645
B.4.2 Resolving Conflicts When Merging . . . 646
B.4.3 git `revert`: Undoing Something . . . 647
B.5 Downloading from GitLab . . . 649
B.6 Advanced . . . 650
B.6.1 git `rebase`: Moving the Base of a Branch . . . 650
B.7 Exercises . . . 651
B.7.1 Create project folder . . . 652
B.8 Initialize repo . . . 652
B.8.1 Do not upload / ignore certain file types . . . 652
B.8.2 Create file and stage it . . . 652
B.8.3 Create another file and check status . . . 652
B.8.4 Commit changes . . . 652
B.8.5 Create a new branch and switch to it . . . 653
B.8.6 Commit changes in the new branch . . . 653
B.8.7 Merge branch into main . . . 653
B.8.8 Resolve merge conflict . . . 653

C Python Introduction **654**
C.1 Recommendations . . . 654

D Documentation of the Sequential Parameter Optimization **655**
D.1 An Initial Example . . . 655
D.2 Organization . . . 657
D.3 The Spot Object . . . 658
D.4 Run . . . 658
D.5 Print the Results . . . 658
D.6 Show the Progress . . . 659
D.7 Visualize the Surrogate . . . 659
D.8 Run With a Specific Start Design . . . 660
D.9 Init: Build Initial Design . . . 661
D.10 Replicability . . . 662
D.11 Surrogates . . . 663
D.11.1 A Simple Predictor . . . 663

D.12 Tensorboard Setup . . . 664
D.12.1 Tensorboard Configuration . . . 664
D.12.2 Starting TensorBoard . . . 664
D.13 Demo/Test: Objective Function Fails . . . 664
D.14 Handling Results: Printing, Saving, and Loading . . . 666
D.15 `spotpython` as a Hyperparameter Tuner . . . 666
D.15.1 Modifying Hyperparameter Levels . . . 666

E Datasets 671
E.1 The Diabetes Data Set . . . 671
E.1.1 Data Exploration of the sklearn Diabetes Data Set . . . 671
E.1.2 Generating the PyTorch Data Set . . . 672
E.2 The Friedman Drift Dataset . . . 673
E.2.1 The Friedman Drift Dataset as Implemented in `river` . . . 673
E.2.2 The Friedman Drift Data Set from `spotpython` . . . 677

F Using Slurm 680
F.1 Introduction . . . 680
F.2 Prepare the Slurm Scripts on the Remote Machine . . . 680
F.3 Generate a `spotpython` Configuration . . . 681
F.4 Copy the Configuration to the Remote Machine . . . 684
F.5 Run the `spotpython` Code on the Remote Machine . . . 684
F.6 Copy the Results to the Local Machine . . . 684
F.7 Analyze the Results on the Local Machine . . . 684
F.7.1 Visualizing the Tuning Progress . . . 684
F.7.2 Design Table with Default and Tuned Hyperparameters . . . 685
F.7.3 Plotting Important Hyperparameters . . . 685

G Solutions to Selected Exercises 686
G.1 Data-Driven Modeling and Optimization . . . 686
G.1.1 Histograms . . . 686
G.1.2 The Normal Distribution . . . 686
G.1.3 The mean, the media, and the mode . . . 687
G.1.4 The exponential distribution . . . 687
G.1.5 Population and Estimated Parameters . . . 687
G.1.6 Calculating the Mean, Variance and Standard Deviation . . . 687
G.1.7 Hypothesis Testing and the Null-Hypothesis . . . 687
G.1.8 Alternative Hypotheses, Main Ideas . . . 688
G.1.9 p-values: What they are and how to interpret them . . . 688
G.1.10 How to calculate p-values . . . 688
G.1.11 p-hacking: What it is and how to avoid it . . . 688
G.1.12 Covariance . . . 689
G.1.13 Pearson's Correlation . . . 689

G.1.14 Boxplots . . . 689
G.1.15 Power Analysis . . . 690
G.1.16 The Central Limit Theorem . . . 690
G.1.17 Boxplots . . . 690
G.1.18 R-squared . . . 690
G.1.19 Linear Regression . . . 691
G.1.20 Multiple Regression . . . 691
G.1.21 A Gentle Introduction to Machine Learning . . . 691
G.1.22 Maximum Likelihood . . . 691
G.1.23 Probability is not Likelihood . . . 691
G.1.24 Cross Validation . . . 691
G.1.25 The Confusion Matrix . . . 691
G.1.26 Sensitivity and Specificity . . . 692
G.1.27 Bias and Variance . . . 692
G.1.28 Mutual Information . . . 692
G.1.29 Principal Component Analysis (PCA) . . . 692
G.1.30 t-SNE . . . 693
G.1.31 K-means clustering . . . 693
G.1.32 DBSCAN . . . 694
G.1.33 K-nearest neighbors . . . 694
G.1.34 Naive Bayes . . . 694
G.1.35 Gaussian Naive Bayes . . . 694
G.1.36 Trees . . . 694
G.2 Machine Learning and Artificial Intelligence . . . 695
G.2.1 Backpropagation . . . 695
G.2.2 Gradient Descent . . . 695
G.2.3 ReLU . . . 695
G.2.4 CNNs . . . 696
G.2.5 RNN . . . 696
G.2.6 LSTM . . . 696
G.2.7 Pytorch/Lightning . . . 697
G.2.8 Embeddings . . . 697
G.2.9 Sequence to Sequence Models . . . 697
G.2.10 Transformers . . . 697

References **699**

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)
```

```
[(0.0, 0.0),
 (0.5, 0.0),
 (1.0, 0.0),
 (0.0, 0.5),
 (0.5, 0.5),
 (1.0, 0.5),
 (0.0, 1.0),
 (0.5, 1.0),
 (1.0, 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

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

```
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.

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 $b = 1$:

```python
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 3.1 (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 3.2 (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:

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

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

Example 3.3 (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 for Optimization

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 3.4 (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='Optim

    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', length=grad

    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 and consi
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 with resp`

   - $\nabla^2 f(x_k)$: 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 3.5 (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]])
```

```python
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 3.6 (BFGS for Rosenbrock)**.**

```
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])
```

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

```
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 2.843987518235081e-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

> **Tasks**
>
> - In which situations is it possible to use algorithms like BFGS, but not the classical Newton method?
> - Investigate the Newton-CG method
> - Use an objective function of your choice and apply Newton-CG
> - Compare the Newton-CG method with the BFGS. What are the similarities and differences between the two algorithms?

3.3 Gradient- and Hessian-based Optimization Algorithms

Section 3.3.1 presents an optimization algorithm that uses gradient and Hessian information to find the minimum. Section 3.3.2 presents an optimization algorithm that uses gradient and Hessian information to find the minimum. Section 3.3.3 presents an optimization algorithm that uses gradient and Hessian information to find the minimum.

The methods Newton-CG, trust-ncg and trust-krylov are suitable for dealing with large-scale problems (problems with thousands of variables). That is because the conjugate gradient algorithm approximately solve the trust-region subproblem (or invert the Hessian) by iterations without the explicit Hessian factorization. Since only the product of the Hessian with an

arbitrary vector is needed, the algorithm is specially suited for dealing with sparse Hessians, allowing low storage requirements and significant time savings for those sparse problems.

3.3.1 Newton-Conjugate-Gradient Algorithm

Newton-Conjugate Gradient algorithm is a modified Newton's method and uses a conjugate gradient algorithm to (approximately) invert the local Hessian.

3.3.2 Trust-Region Newton-Conjugate-Gradient Algorithm

3.3.3 Trust-Region Truncated Generalized Lanczos / Conjugate Gradient Algorithm

3.4 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.

3.4.1 Local vs Global Optimization

3.4.1.1 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

3.4.1.2 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

3.4.2 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: -959.6406627208489
       x: [ 5.120e+02  4.042e+02]
     nit: 1000
    nfev: 4028
    njev: 9
    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: -935.3379515584882
                   x: [ 4.395e+02  4.540e+02]
                 nit: 23
                nfev: 741
          population: [[ 4.393e+02  4.539e+02]
                       [ 4.392e+02  4.542e+02]
                       ...
                       [ 4.459e+02  4.608e+02]
                       [ 4.391e+02  4.538e+02]]
 population_energies: [-9.353e+02 -9.351e+02 ... -9.264e+02 -9.353e+02]
                 jac: [ 2.274e-05  4.547e-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()
```

3.4.3 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.

3.4.3.1 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.

3.4.3.2 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.

3.4.3.3 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.pi * x

# 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
```

```
result
```

```
 message: ['Maximum number of iteration reached']
 success: True
  status: 0
     fun: 7.105427357601002e-15
       x: [-4.601e-09 -4.319e-09]
     nit: 1000
    nfev: 4088
    njev: 29
    nhev: 0
```

3.4.3.4 Example of Comparison Table:

Function	Algorithm	Best Objective Function Value	Number of Iterations	Number of Evaluations	Local/Global Optimizer	Gradient-Based/Free Algorithm	Other Comments
Rastrigin	Algorithm 1	1.23	1000	5000	Global	Free	?
Rastrigin	Algorithm 2	0.95	800	4000	Local	Gradient-Based	?
Rastrigin	Algorithm 3	1.45	1200	6000	Global	Free	?
Rosenbrock	Algorithm 1	2.56	1500	7500	Local	Gradient-Based	?
Rosenbrock	Algorithm 2	2.12	1200	6000	Global	Free	?
Rosenbrock	Algorithm 3	2.73	1800	9000	Local	Gradient-Based	?

3.4.4 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.

3.4.5 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.

```python
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='Optimal

# 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
```

3.4.6 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

3.4.7 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).

3.4.8 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

3.5 Project: One-Mass Oscillator Optimization

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

3.5.1 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.

3.5.2 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.

3.5.3 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 frequencies
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)]
```

3.5.4 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)
```

3.5.5 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,  5.354e+01
```

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,
)
```

3.5.6 Tasks

- Investigate the trust region Conjugate Gradient method
- Investigate the Differential Evolution Algorithm

- Compare all the methods we introduced for the Rosenbrock (4d) and Rastrigin (2d) function

3.5.7 Task for the Project Work

> **Task**
>
> - Experiment with various optimizers to identify the optimal parameter setup for the one-mass oscillator model and both data frames.
> - Please explain your thoughts and ideas, and subsequently compare the results obtained from different optimizers.

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_in
```

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

```
spotpython tuning: 3.8004550038787155 [######----] 55.00%
spotpython tuning: 3.8004550038787155 [######----] 60.00%
spotpython tuning: 3.1588579885698627 [######----] 65.00%
spotpython tuning: 3.1342382932317037 [#######---] 70.00%
spotpython tuning: 2.8956615907630585 [########--] 75.00%
spotpython tuning: 0.42052429574482275 [########--] 80.00%
spotpython tuning: 0.4013351867835322 [########--] 85.00%
spotpython tuning: 0.399265616254338 [#########-] 90.00%
spotpython tuning: 0.399265616254338 [##########] 95.00%
spotpython tuning: 0.399265616254338 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x167256900>
```

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: 0.399265616254338
x0: 3.151170754781285
x1: 2.2981660114765448

[['x0', 3.151170754781285], ['x1', 2.2981660114765448]]
```

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.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: 3.8004480172281534 [######----] 55.00%
spotpython tuning: 3.8004480172281534 [######----] 60.00%
spotpython tuning: 3.158996247273234 [######----] 65.00%
spotpython tuning: 3.134218255713952 [#######---] 70.00%
spotpython tuning: 2.8926591957342467 [########--] 75.00%
spotpython tuning: 0.4189006494820333 [########--] 80.00%
spotpython tuning: 0.4019392204560983 [########--] 85.00%
spotpython tuning: 0.39922543271904765 [#########-] 90.00%
spotpython tuning: 0.39922543271904765 [##########] 95.00%
spotpython tuning: 0.39922543271904765 [##########] 100.00% Done...

min y: 0.39922543271904765
x0: 3.1506699177492252
x1: 2.298631597428197
```

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.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...

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.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: 3.8004562736456844 [######----] 55.00%
spotpython tuning: 3.8004562736456844 [######----] 60.00%
spotpython tuning: 3.158996879015902 [######----] 65.00%
spotpython tuning: 3.1341298968229996 [#######---] 70.00%
spotpython tuning: 2.8919915800445954 [########--] 75.00%
spotpython tuning: 0.4173165753511867 [########--] 80.00%
spotpython tuning: 0.40097732409794773 [########--] 85.00%
spotpython tuning: 0.3993020098909934 [#########-] 90.00%
spotpython tuning: 0.3993020098909934 [##########] 95.00%
spotpython tuning: 0.3993020098909934 [##########] 100.00% Done...

min y: 0.3993020098909934
x0: 3.1510894339439672
x1: 2.298936853041466
```

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.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: 3.800453600053931 [######----] 55.00%
spotpython tuning: 3.800453600053931 [######----] 60.00%
spotpython tuning: 3.1590141837294237 [######----] 65.00%
spotpython tuning: 3.1341341806066314 [#######---] 70.00%
spotpython tuning: 2.8914331943522242 [########--] 75.00%
spotpython tuning: 0.41214245125719984 [########--] 80.00%
spotpython tuning: 0.40113843843078634 [########--] 85.00%
spotpython tuning: 0.3992327747775164 [#########-] 90.00%
spotpython tuning: 0.3992327747775164 [##########] 95.00%
spotpython tuning: 0.3992327747775164 [##########] 100.00% Done...

min y: 0.3992327747775164
x0: 3.15016404734246
x1: 2.2998320162156896
```

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 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

5.5.1 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

5.6 RSM Experimentation

5.6.1 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

5.6.2 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

5.6.3 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

5.7 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

5.7.1 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

5.7.2 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

5.7.3 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

5.7.4 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

5.8 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.

5.9 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).

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.

6.2 Background: Expectation, Mean, Standard Deviation

The distribution of a random vector is characterized by some indexes. One of them is the expected value, which is defined as

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

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

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: Expectation**
>
> Let X denote the number produced by rolling a fair die. Then
>
> $$E(X) = 1 \times 1/6 + 2 \times 1/6 + 3 \times 1/6 + 4 \times 1/6 + 5 \times 1/6 + 6 \times 1/6 = 3.5.$$

6.2.1 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

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

6.2.2 Variance and Standard Deviation

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:

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

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

6.2.3 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.

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

6.2.4 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

$$\sqrt{\frac{1}{N}\sum_i (x_i - \bar{x})^2} \qquad \text{with} \qquad \bar{x} = \sum_{i=1}^{N} x_i/N. \tag{6.3}$$

Example: 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)
```

```
0.816496580927726
```

6.2.5 The Empirical Standard Deviation

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 as follows:

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

```
1.0
```

6.2.6 The Argument "axis"

> **Axes along which the standard deviation is computed**
>
> - 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.

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

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

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

```
1.118033988749895
```

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

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

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

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

6.3 Data Types and Precision in Python

We consider single versus double precision in Python. In single precision, `std()` can be inaccurate:

```python
a = np.zeros((2, 4*4), dtype=np.float32)
a[0, :] = 1.0
a[1, :] = 0.1
a
```

```
array([[1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. ,
        1. , 1. , 1. ],
       [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1,
        0.1, 0.1, 0.1]], dtype=float32)
```

```python
np.std(a, axis=0)
```

```
array([0.45, 0.45, 0.45, 0.45, 0.45, 0.45, 0.45, 0.45, 0.45, 0.45, 0.45,
       0.45, 0.45, 0.45, 0.45, 0.45], dtype=float32)
```

```python
np.std(a, axis=1)
```

```
array([0., 0.], dtype=float32)
```

```python
abs(0.45 - np.std(a))
```

```
1.7881393421514957e-08
```

> **Float data types**
>
> - float32 and float64 are data types in numpy that specify the precision of floating point numbers.
> - float32 is a single-precision floating point number that occupies 32 bits of memory. It has a precision of about 7 decimal digits.
> - float64 is a double-precision floating point number that occupies 64 bits of memory. It has a precision of about 15 decimal digits.
> - The main difference between float32 and float64 is the precision and memory usage. float64 provides a higher precision but uses more memory, while float32 uses less memory but has a lower precision.

Computing the standard deviation in float64 is more accurate (result may vary), see https://numpy.org/devdocs/reference/generated/numpy.std.html.

```
abs(0.45 - np.std(a, dtype=np.float64))
```

```
7.450580707946131e-10
```

Example: 32 versus 64 bit

```
import numpy as np

# Define a number
num = 0.123456789123456789

# Convert to float32 and float64
num_float32 = np.float32(num)
num_float64 = np.float64(num)

# Print the number in both formats
print("float32: ", num_float32)
print("float64: ", num_float64)
```

```
float32:  0.12345679
float64:  0.12345678912345678
```

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 float32 shows 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.

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

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

The probability density function of the normal distribution is defined as:

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right), \tag{6.4}$$

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.

```
# generate 10 random numbers between from a normal distribution
import numpy as np
rng = np.random.default_rng()
n = 10
mu, sigma = 2, 0.1
x = rng.normal(mu, sigma, n)
x
```

```
array([1.95638919, 1.96402437, 1.87085487, 2.14292207, 2.09659183,
       2.00431349, 1.99292669, 2.02882354, 1.95938996, 2.0611441 ])
```

Verify the mean:

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

```
0.007738009244959709
```

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

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

```
0.021599493496973043
```

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.

```
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. 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.

For example, 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's a Python example:

```
import numpy as np

# Define the parameters of the normal distribution
mu = 0
sigma = 1

# Draw 10 samples (realizations) from the normal distribution
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 10 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 6.4.

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.

> **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:

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

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. 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. For discrete random variables, covariance can be written as:

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

Figure 6.1 shows draws from a bivariate normal distribution with $\mu = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and $\Sigma = \begin{pmatrix} 9 & 4 \\ 4 & 9 \end{pmatrix}$.

```
import numpy as np
rng = np.random.default_rng()
import matplotlib.pyplot as plt
mean = [0, 0]
cov = [[9, 4], [4, 9]]  # diagonal covariance
x, y = rng.multivariate_normal(mean, cov, 1000).T
# Create a scatter plot of the numbers
plt.scatter(x, y, s=2)
plt.axis('equal')
plt.grid()
plt.title(f"Bivariate Normal. Mean zero and positive covariance: {cov}")
plt.show()
```

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

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 $y = x$.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal

# Parameters
```

```
mu = np.array([0, 0])
cov = np.array([[9, 4], [4, 9]])

# Create grid and multivariate normal
x = np.linspace(-10,10,100)
y = np.linspace(-10,10,100)
X, Y = np.meshgrid(x,y)
pos = np.empty(X.shape + (2,))
pos[:, :, 0] = X; pos[:, :, 1] = Y
rv = multivariate_normal(mu, cov)

fig = plt.figure()
ax = plt.axes(projection='3d')
surf=ax.plot_surface(X, Y, rv.pdf(pos),cmap='viridis',linewidth=0)
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
ax.set_title('Bivariate Normal Distribution')
fig.colorbar(surf, shrink=0.5, aspect=10)
plt.show()
```

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

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

6.4.8 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 Cholesky Decomposition and Positive Definite Matrices

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

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

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.

```
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
```

More effficent (and check if symmetric) is based on Cholesky decomposition.

```
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
```

Example: Cholesky decomposition.

`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.]]
```

6.6 Maximum Likelihood Estimation: 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., $X^{(j)} \sim N(\mu, \Sigma)$, $j = 1, 2, \ldots$. 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 $x_1, \ldots x_n$, i.e., the realizations of the first n random vectors in the sequence, to estimate the two unknown parameters μ and Σ.

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 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:

$$\begin{aligned} L(\mu, \Sigma) &= \prod_{j=1}^{n} f_X(x_j) = \prod_{j=1}^{n} \frac{1}{\sqrt{(2\pi)^k \det(\Sigma)}} \exp\left(-\frac{1}{2}(x_j - \mu)^T \Sigma^{-1}(x_j - \mu)\right) \\ &= \frac{1}{(2\pi)^{nk/2} \det(\Sigma)^{n/2}} \exp\left(-\frac{1}{2}\sum_{j=1}^{n}(x_j - \mu)^T \Sigma^{-1}(x_j - \mu)\right). \end{aligned}$$

The log-likelihood function is

$$\ell(\mu, \Sigma) = -\frac{nk}{2}\ln(2\pi) - \frac{n}{2}\ln(\det(\Sigma)) - \frac{1}{2}\sum_{j=1}^{n}(x_j - \mu)^T \Sigma^{-1}(x_j - \mu).$$

The likelihood function is well-defined only if $\det(\Sigma) > 0$.

6.7 Introduction to Gaussian Processes

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.

6.7.1 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.

6.7.2 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 6.5.

```
visualize_inverse_exp_squared_distance(5, 0.0, [0.5, 1, 2.0])
```

Figure 6.5: One-dim inverse exponentiated squared Euclidean distance

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

$$\Sigma(\vec{x}, \vec{x}') = \exp\{-||\vec{x} - \vec{x}'||^2/(2\sigma^2)\}. \tag{6.5}$$

The covariance decays exponentially fast as $\vec{x}$ and $\vec{x}'$ become farther apart. Observe that

$$\Sigma(\vec{x}, \vec{x}) = 1$$

and

$$\Sigma(\vec{x}, \vec{x}') < 1$$

for $\vec{x} \neq \vec{x}'$. The function $\Sigma(\vec{x}, \vec{x}')$ must be positive definite.

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 $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n$. 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.

6.7.3 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, mat, spacing, sqr
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.   ],
       [0.995, 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.   ],
       [0.98 , 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.   ]])
```

```python
import matplotlib.pyplot as plt
plt.imshow(Sigma, cmap='hot', interpolation='nearest')
plt.colorbar()
plt.show()
```

6.7.4 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").resh
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(sigma2[0]))
plt.show()
```

Figure 6.6: 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(sigma
plt.show()
```

Figure 6.7: Realization of three random functions under a GP prior. sigma2: 1.0

6.7.5 Properties of the 1d Example

6.7.5.1 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.

6.7.5.2 Smoothness:

The function plotted in Figure 6.6 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.

6.7.5.3 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, mat, spacing, sqrt
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
    if show:
        plt.show()
```

```
plot_mvn(a=0, b=10, sigma2=10.0, size=3, n=250)
```

Figure 6.8: Realization of Random Functions under a GP prior. sigma2: 10

```
plot_mvn(a=0, b=10, sigma2=0.1, size=3, n=250)
```

Figure 6.9: Realization of Random Functions under a GP prior. sigma2: 0.1

6.8 Kriging: Modeling Basics

6.8.1 The Kriging Idea in a Nutshell

We consider observed data of an unknown function f at n points $x_1, \dots, x_n$, see Figure 6.10. 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 6.7, Figure 6.8 or Figure 6.9. In Kriging, a more general covariance matrix (or equivalently, a correlation matrix Ψ) is used, see Equation 6.6. 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.

6.8.2 The Kriging Basis Function

k-dimensional basis functions of the form

$$\psi(\vec{x}^{(i)}, \vec{x}^{(j)}) = \exp\left(-\sum_{l=1}^{k} \theta_l |x_l^{(i)} - x_l^{(j)}|^{p_l}\right) \tag{6.6}$$

Figure 6.10: Eight measurements of an unknown function

are used in a method known as Kriging. Note, $\vec{x}^{(i)}$ denotes the k-dim vector $\vec{x}^{(i)} = (x_1^{(i)}, \dots, x_k^{(i)})^T$.

The Kriging basis function is related to the 1-dim Gaussian basis function (Equation 6.5), which is defined as

$$\Sigma(\vec{x}^{(i)}, \vec{x}^{(j)}) = \exp\{-||\vec{x}^{(i)} - \vec{x}^{(j)}||^2/(2\sigma^2)\}. \tag{6.7}$$

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).

6.8.3 The Correlation Coefficient

In a bivariate normal distribution, the covariance matrix and the correlation coefficient are closely related. The covariance matrix Σ for a bivariate normal distribution is a 2×2 matrix that looks like this:

$$\Sigma = \begin{pmatrix} \sigma_1^2 & \sigma_{12} \\ \sigma_{21} & \sigma_2^2 \end{pmatrix},$$

where σ_1^2 and σ_2^2 are the variances of X_1 and X_2, and $\sigma_{12} = \sigma_{21}$ is the covariance between X_1 and X_2.

The correlation coefficient, often denoted as ρ, is a normalized measure of the linear relationship between two variables. It is calculated from the covariance and the standard deviations σ_1 and σ_2 (or the square roots of the variances) of X_1 and X_2 as follows:

$$\rho = \sigma_{12}/(\sqrt{\sigma_1^2} \times \sqrt{\sigma_2^2}) = \sigma_{12}/(\sigma_1 \times \sigma_2).$$

So we can express the correlation coefficient ρ in terms of the elements of the covariance matrix Σ. It can be interpreted as follows: The correlation coefficient ranges from -1 to 1. A value of 1 means that X_1 and X_2 are perfectly positively correlated, a value of -1 means they are perfectly negatively correlated, and a value of 0 means they are uncorrelated. This gives the same information as the covariance, but on a standardized scale that does not depend on the units of X_1 and X_2.

6.8.4 Covariance Matrix and Correlation Matrix

Covariance and Correlation (taken from @Forr08a)

Covariance is a measure of the correlation between two or more sets of random variables.

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

From the covariance, we can derive the correlation

$$\mathrm{Corr}(X, Y) = \frac{\mathrm{Cov}(X, Y)}{\sqrt{\mathrm{Var}(X)\mathrm{Var}(Y)}} = \frac{\mathrm{Cov}(X, Y)}{\sigma_X \sigma_Y}. \tag{6.8}$$

For a vector of random variables

$$Y = \left((Y^{(l)}, \dots, Y^{(n)})\right)^T$$

the covariance matrix is a matrix of covariances between the random variables

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

and from Equation 6.8

$$\mathrm{Cov}(Y, Y) = \sigma_Y^2 \mathrm{Cor}(Y, Y).$$

You can compute the correlation matrix Ψ from a covariance matrix Σ in Python using the numpy library. The correlation matrix is computed by dividing each element of the covariance matrix by the product of the standard deviations of the corresponding variables.

The function `covariance_to_correlation` first computes the standard deviations of the variables with `np.sqrt(np.diag(cov))`. It then computes the correlation matrix by dividing each element of the covariance matrix by the product of the standard deviations of the corresponding variables with `cov / np.outer(std_devs, std_devs)`.

```
import numpy as np

def covariance_to_correlation(cov):
    # Compute standard deviations
    std_devs = np.sqrt(np.diag(cov))

    # Compute correlation matrix
    corr = cov / np.outer(std_devs, std_devs)

    return corr

cov = np.array([[9, -4], [-4, 9]])
print(covariance_to_correlation(cov))
```

```
[[ 1.         -0.44444444]
 [-0.44444444  1.        ]]
```

6.8.5 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.

The observed responses $\vec{y}$ are considered as if they are from a stochastic process, which will be denoted as

$$\begin{pmatrix} \vec{Y}(\vec{x}^{(1)}) \\ \vdots \\ \vec{Y}(\vec{x}^{(n)}) \end{pmatrix}.$$

The set of random vectors (also referred to as a *random field*) has a mean of $\vec{1}\mu$, which is a $(n \times 1)$ vector.

6.8.6 Correlations

The random vectors are correlated with each other using the basis function expression from Equation 6.6:

$$\text{cor}\left(\vec{Y}(\vec{x}^{(i)}), \vec{Y}(\vec{x}^{(l)})\right) = \exp\left\{-\sum_{j=1}^{k} \theta_j |x_j^{(i)} - x_j^{(l)}|^{p_j}\right\}.$$

The $(n \times n)$ correlation matrix of the observed sample data is

$$\vec{\Psi} = \begin{pmatrix} \text{cor}\left(\vec{Y}(\vec{x}^{(i)}), \vec{Y}(\vec{x}^{(l)})\right) & \dots & \text{cor}\left(\vec{Y}(\vec{x}^{(i)}), \vec{Y}(\vec{x}^{(l)})\right) \\ \vdots & \vdots & \vdots \\ \text{cor}\left(\vec{Y}(\vec{x}^{(i)}), \vec{Y}(\vec{x}^{(l)})\right) & \dots & \text{cor}\left(\vec{Y}(\vec{x}^{(i)}), \vec{Y}(\vec{x}^{(l)})\right) \end{pmatrix}.$$

Note: correlations depend on the absolute distances between sample points $|x_j^{(n)} - x_j^{(n)}|$ and the parameters p_j and θ_j.

Correlation is intuitive, because when two points move close together, then $|x_l^{(i)} - x_l| \to 0$ and $\exp(-|x_l^{(i)} - x_l| \to 1$, points show very close correlation and $Y(x_l^{(i)}) = Y(x_l)$.

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 6.11.

```
visualize_inverse_exp_squared_distance(5, 0, theta_values=[0.5, 1, 2.0])
```

Figure 6.11: Theta set to 1/2, 1, and 2

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:

$$\begin{pmatrix} 0 & \theta_1(x_{11} - x_{21})^2 + \theta_2(x_{12} - x_{22})^2 + \theta_3(x_{13} - x_{23})^2 & \theta_1(x_{11} - x_{31})^2 + \theta_2(x_{12} - x_{32})^2 + \theta_3(x_{13} - x_{33})^2 & \theta_1(\\ 0 & 0 & \theta_1(x_{21} - x_{31})^2 + \theta_2(x_{22} - x_{32})^2 + \theta_3(x_{23} - x_{33})^2 & \theta_1 \\ 0 & 0 & 0 & \theta_1(\\ 0 & 0 & 0 & \end{pmatrix}$$

Finally,

$$\Psi = \exp(-D)$$

is computed.
Next, we will demonstrate how this computation can be implemented in Python.

```
from numpy import (array, zeros, power, ones, exp, multiply,
                   eye, linspace, mat, 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: 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.        ]])
```

6.8.7 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:  200000000.53159264
```

```
np.linalg.cond(Psi)
```

```
2.163953413738652
```

6.8.8 MLE to estimate θ and p

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

$$L\left(\vec{Y}(\vec{x}^{(1)}), \ldots, \vec{Y}(\vec{x}^{(n)})|\mu, \sigma\right) = \frac{1}{(2\pi\sigma)^{n/2}|\vec{\Psi}|^{1/2}} \exp\left\{\frac{-(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)}{2\sigma^2}\right\},$$

and formulated as the log-likelihood:

$$\ln(L) = -\frac{n}{2}\ln(2\pi\sigma) - \frac{1}{2}\ln|\vec{\Psi}|\frac{-(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)}{2\sigma^2}.$$

Optimization of the log-likelihood by taking derivatives with respect to μ and σ results in

$$\hat{\mu} = \frac{\vec{1}^T\vec{\Psi}^{-1}\vec{y}^T}{\vec{1}^T\vec{\Psi}^{-1}\vec{1}^T}$$

and

$$\hat{\sigma} = \frac{(\vec{y}-\vec{1}\mu)^T\vec{\Psi}^{-1}(\vec{y}-\vec{1}\mu)}{n}.$$

Combining the equations leads to the concentrated log-likelihood:

$$\ln(L) = -\frac{n}{2}\ln(\hat{\sigma}) - \frac{1}{2}\ln|\vec{\Psi}|. \tag{6.9}$$

> **Note: The Concentrated Log-Likelihood**
>
> - The first term in Equation 6.9 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 equation is not differentiable.

6.8.9 Tuning 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.

6.9 Kriging Prediction

6.9.1 The Augmented Correlation Matrix

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 $\tilde{\vec{y}} = (\vec{y}^T, \hat{y})^T$. A vector of correlations between the observed data and the new prediction is defined as

$$\vec{\psi} = \begin{pmatrix} \operatorname{cor}\left(\vec{Y}(\vec{x}^{(1)}), \vec{Y}(\vec{x})\right) \\ \vdots \\ \operatorname{cor}\left(\vec{Y}(\vec{x}^{(n)}), \vec{Y}(\vec{x})\right) \end{pmatrix} = \begin{pmatrix} \vec{\psi}^{(1)} \\ \vdots \\ \vec{\psi}^{(n)} \end{pmatrix}.$$

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

$$\ln(L) = -\frac{n}{2}\ln(2\pi) - \frac{n}{2}\ln(\hat{\sigma}^2) - \frac{1}{2}\ln|\vec{\hat{\Psi}}| - \frac{(\vec{\tilde{y}} - \vec{1}\hat{\mu})^T \vec{\tilde{\Psi}}^{-1} (\vec{\tilde{y}} - \vec{1}\hat{\mu})}{2\hat{\sigma}^2}.$$

The MLE for $\hat{y}$ can be calculated as

$$\hat{y}(\vec{x}) = \hat{\mu} + \vec{\psi}^T \vec{\tilde{\Psi}}^{-1} (\vec{y} - \vec{1}\hat{\mu}). \tag{6.10}$$

6.9.2 Properties of the Predictor

Equation 6.10 reveals two important properties of the Kriging predictor:

1. 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 $\vec{w} = \vec{\tilde{\Psi}}^{(-1)}(\vec{y} - \vec{1}\hat{\mu})$.
2. 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, $\hat{y}(\vec{x}^{(i)}) = y^{(i)}$.

6.10 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.

6.10.1 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)
# 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)
```

Evaluate at sample points

```
y = np.sin(X)
```

```
import matplotlib.pyplot as plt
plt.plot(X, y, "bo")
plt.title(f"Sin(x) evaluated at {n} points")
plt.show()
```

6.10.2 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)
    print(D.shape)
    psi = exp(-D)
    # return psi transpose to be consistent with the literature
    return(psi.T)
```

6.10.3 Predicting at New Locations

We would like to predict at $m = 100$ new 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)
psi = build_psi(X, x, theta)
```

```
(100, 8)
```

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)))
```

To compute f, Equation 6.10 is used.

6.10.4 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()
```

6.11 Cholesky Example With Two Points

6.11.1 Cholesky Decomposition

We consider $k = 1$ 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 6.6, we can compute the correlation matrix Ψ:

$$\Psi = \begin{pmatrix} 1 & e^{-1} \\ e^{-1} & 1 \end{pmatrix}.$$

To determine MLE as in Equation 6.10, 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:

$$LDL^T = \begin{pmatrix} 1 & 0 \\ l_{21} & 1 \end{pmatrix} \begin{pmatrix} d_{11} & 0 \\ 0 & d_{22} \end{pmatrix} \begin{pmatrix} 1 & l_{21} \\ 0 & 1 \end{pmatrix} =$$

$$\begin{pmatrix} d_{11} & 0 \\ d_{11}l_{21} & d_{22} \end{pmatrix} \begin{pmatrix} 1 & l_{21} \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} d_{11} & d_{11}l_{21} \\ d_{11}l_{21} & d_{11}l_{21}^2 + d_{22} \end{pmatrix}. \tag{6.11}$$

Using Equation 6.11, 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

$$\Psi = \begin{pmatrix} 1 & 0 \\ e^{-1} & \sqrt{1 - e^{-2}} \end{pmatrix} \begin{pmatrix} 1 & e^{-1} \\ 0 & \sqrt{1 - e^{-2}} \end{pmatrix} = U^T U.$$

6.11.2 Computation of the Inverse Matrix

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]]
```

6.12 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.

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.

7.1 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
```

7.1.1 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()
```

7.1.2 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.Spot(fun=fun,
                   fun_control=fun_control)
spot_0.run()
```

```
spotpython tuning: 4.959603317754042e-09 [#######---] 73.33%
spotpython tuning: 4.959603317754042e-09 [########--] 80.00%
spotpython tuning: 4.959603317754042e-09 [#########-] 86.67%
spotpython tuning: 4.959603317754042e-09 [#########-] 93.33%
spotpython tuning: 1.866838525968143e-10 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x3786c2900>
```

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', 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.

7.2 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.Spot(fun=fun,
                   fun_control=fun_control,
                   design_control=design_control)
spot_1.run()
```

```
spotpython tuning: 9.642507519254602e-09 [##########] 100.00% Done...
```

7.3 Print the Results

```
spot_1.print_results()
```

```
min y: 9.642507519254602e-09
x0: -9.819627039381181e-05

[['x0', -9.819627039381181e-05]]
```

7.4 Show the Progress

```
spot_1.plot_progress()
```

7.5 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]))
design_control = design_control_init(init_size=5)
```

Since the `PREFIX` is not None, `spotpython` will log the optimization process in the TensorBoard files.

```
spot_tuner = spot.Spot(fun=fun,
                   fun_control=fun_control,
                   design_control=design_control)
spot_tuner.run()
spot_tuner.print_results()
```

```
spotpython tuning: 2.4876122402648534e-05 [####------] 40.00%
spotpython tuning: 8.406820960103693e-07 [#####-----] 46.67%
spotpython tuning: 7.414780214595436e-07 [#####-----] 53.33%
```

```
spotpython tuning: 3.807407090267134e-07 [######----] 60.00%
spotpython tuning: 5.01557230424484e-09 [#######---] 66.67%
spotpython tuning: 5.01557230424484e-09 [#######---] 73.33%
spotpython tuning: 5.01557230424484e-09 [########--] 80.00%
spotpython tuning: 5.01557230424484e-09 [#########-] 86.67%
spotpython tuning: 5.01557230424484e-09 [#########-] 93.33%
spotpython tuning: 5.01557230424484e-09 [##########] 100.00% Done...

min y: 5.01557230424484e-09
x0: -7.082070533569148e-05


[['x0', -7.082070533569148e-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.

7.6 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.

8.1 Example: Spot and the 3-dim Sphere Function

```
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
```

8.1.1 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:

$$f(x) = \sum_{i}^{k} x_i^2.$$

It is avaliable 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 7.5), 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`. 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 interpreable 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"],
              show_progress=True)
surrogate_control = surrogate_control_init(n_theta=3)
spot_3 = spot.Spot(fun=fun,
                  fun_control=fun_control,
                  surrogate_control=surrogate_control)
spot_3.run()
```

```
spotpython tuning: 0.03443452603483929 [#######---] 73.33%
spotpython tuning: 0.031343357117474185 [########--] 80.00%
spotpython tuning: 0.0009629583911652714 [#########-] 86.67%
spotpython tuning: 8.541680652934052e-05 [#########-] 93.33%
spotpython tuning: 6.65563988471339e-05 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x35f41b3e0>
```

> **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/
> ```

8.1.2 Results

```
_ = spot_3.print_results()
```

```
min y: 6.65563988471339e-05
Pressure: 0.005332620051379623
Temp: 0.001932437769368857
Lambda: 0.0058638934593215975
```

```
spot_3.plot_progress()
```

8.1.3 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_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:  95.16050385357518
Temp:  100.0
Lambda:  86.66610935794161
```

```
spot_3.plot_importance()
```

8.1.4 TensorBoard

Figure 8.1: 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 8.2: TensorBoard visualization of the spotpython surrogate model.

8.1.5 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 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  2.        ]
 [-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   2.          32.09388125  22.09388125]
 [ -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=["num"
S.fit(X, y)
Sf = Kriging(name='kriging',  seed=123, log_level=50, n_theta=3, noise=False, var_type=["num
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	6.684749	17.660408	8.981430
1	95.865258	90.509501	94.789595
2	49.811774	31.120556	50.354535
3	8.177150	5.917583	8.441051
4	10.968377	14.164812	4.820856

```
df.tail()
```

	y	Prediction	Prediction_factor
9995	73.620503	82.887200	73.604506
9996	76.187178	92.365607	76.894275
9997	29.494401	27.820944	29.928268
9998	15.390268	15.671179	3.957986
9999	26.261264	13.626484	25.011336

```
s=np.sum(np.abs(y_pred - y_true))
sf=np.sum(np.abs(y_factor_pred - y_true))
res = (sf - s)
print(res)
```

```
-93783.37196523952
```

```
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")
```

8.3 Exercises

8.3.1 1. The Three Dimensional `fun_cubed`

- The input dimension is 3. The search range is $-1 \leq x \leq 1$ 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?

8.3.2 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?

8.3.3 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?

8.3.4 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?

8.3.5 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?

8.4 Selected Solutions

8.4.1 Solution to Exercise Section 8.3.5: 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
```

8.4.1.1 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),
              show_progress=True,
              fun_evals=25)
surrogate_control = surrogate_control_init(n_theta=2)
spot_rosen = spot.Spot(fun=fun_rosen,
                  fun_control=fun_control,
                  surrogate_control=surrogate_control)
spot_rosen.run()
```

```
spotpython tuning: 90.78805160669465 [####------] 44.00%
spotpython tuning: 1.0171930962259077 [#####-----] 48.00%
spotpython tuning: 1.0171930962259077 [#####-----] 52.00%
spotpython tuning: 1.0171930962259077 [######----] 56.00%
spotpython tuning: 1.0171930962259077 [######----] 60.00%
spotpython tuning: 1.0171930962259077 [######----] 64.00%
spotpython tuning: 1.0171930962259077 [#######---] 68.00%
spotpython tuning: 0.9722891367911908 [#######---] 72.00%
spotpython tuning: 0.9722891367911908 [########--] 76.00%
spotpython tuning: 0.9722891367911908 [########--] 80.00%
spotpython tuning: 0.9722891367911908 [########--] 84.00%
spotpython tuning: 0.8058112323775151 [#########-] 88.00%
spotpython tuning: 0.8058112323775151 [#########-] 92.00%
spotpython tuning: 0.8058112323775151 [##########] 96.00%
spotpython tuning: 0.726650609991151 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x3614fb230>
```

> **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/
> ```

8.4.1.2 Results

```
_ = spot_rosen.print_results()
```

```
min y: 0.726650609991151
x0: 0.17237438431069813
x1: 0.09427797327939783
```

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

8.4.1.3 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()
```

8.4.1.4 TensorBoard

TBD

8.5 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.Spot(fun=fun,
                   fun_control=fun_control,
                   surrogate_control=surrogate_control)

spot_2.run()
```

```
spotpython tuning: 2.217297132344163e-05 [#######---] 73.33%
spotpython tuning: 2.217297132344163e-05 [########--] 80.00%
spotpython tuning: 2.217297132344163e-05 [#########-] 86.67%
spotpython tuning: 2.217297132344163e-05 [#########-] 93.33%
spotpython tuning: 2.217297132344163e-05 [##########] 100.00% Done...

<spotpython.spot.spot.Spot at 0x3561b9e80>
```

9.1.2 Results

```
spot_2.print_results()
```

```
min y: 2.217297132344163e-05
x0: 0.001637745619980198
x1: 0.004414834154039912

[['x0', 0.001637745619980198], ['x1', 0.004414834154039912]]
```

```
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.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: 1.4886245843852055e-05 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x356c986b0>
```

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: 1.4886245843852055e-05
x0: -0.003850983818095356
x1: -0.00023700100552480994

[['x0', -0.003850983818095356], ['x1', -0.00023700100552480994]]
```

```
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
```

```
array([-0.32858908, -0.13977864])
```

- 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
```

```
array([-0.17881419])
```

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.Spot(fun=fun_rosen,
                   fun_control=fun_control,
                   surrogate_control=surrogate_control)
spot_rosen.run()
```

```
spotpython tuning: 52.876358299143206 [#######---] 73.33%
spotpython tuning: 52.36204755139067 [########--] 80.00%
spotpython tuning: 52.36204755139067 [#########-] 86.67%
spotpython tuning: 43.44294819789333 [#########-] 93.33%
spotpython tuning: 12.275982031887748 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x35d5cc3e0>
```

> **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: 12.275982031887748
x0: -2.3708319530278255
x1: 5.923081469551754
```

```
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.Spot(fun=fun_rosen,
                   fun_control=fun_control,
                   surrogate_control=surrogate_control)
spot_rosen.run()
```

```
spotpython tuning: 90.78805160669465 [#######---] 73.33%
spotpython tuning: 1.0171930962259077 [########--] 80.00%
spotpython tuning: 1.0171930962259077 [#########-] 86.67%
spotpython tuning: 1.0171930962259077 [#########-] 93.33%
spotpython tuning: 1.0171930962259077 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x35d965b50>
```

> **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: 1.0171930962259077
x0: 0.002781302621691875
x1: -0.047687091984116046
```

```
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:  100.0
x1:  2.2275258776369804
```

```
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 7.5, 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.Spot(fun=fun,
                   fun_control=fun_control,
                   design_control=design_control)
```

```
spot_2.run()
```

```
spotpython tuning: 3.8004550038787155 [######----] 55.00%
spotpython tuning: 3.8004550038787155 [######----] 60.00%
spotpython tuning: 3.1588579885698627 [######----] 65.00%
spotpython tuning: 3.1342382932317037 [#######---] 70.00%
spotpython tuning: 2.8956615907630585 [########--] 75.00%
spotpython tuning: 0.42052429574482275 [########--] 80.00%
spotpython tuning: 0.4013351867835322 [########--] 85.00%
spotpython tuning: 0.399265616254338 [#########-] 90.00%
spotpython tuning: 0.399265616254338 [##########] 95.00%
spotpython tuning: 0.399265616254338 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x12c5593d0>
```

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: 0.399265616254338
x0: 3.151170754781285
x1: 2.2981660114765448

[['x0', 3.151170754781285], ['x1', 2.2981660114765448]]
```

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.Spot(fun=fun,
                      fun_control=fun_control,
                      design_control=design_control,
                      surrogate = S_GP)
spot_2_GP.run()
```

```
spotpython tuning: 18.865120626024897 [######----] 55.00%
spotpython tuning: 4.067035571881624 [######----] 60.00%
spotpython tuning: 3.4619135134152215 [######----] 65.00%
spotpython tuning: 3.4619135134152215 [#######---] 70.00%
spotpython tuning: 1.328248854169006 [########--] 75.00%
spotpython tuning: 0.9548473238361144 [########--] 80.00%
spotpython tuning: 0.9362788065438252 [########--] 85.00%
spotpython tuning: 0.40009463636571496 [#########-] 90.00%
spotpython tuning: 0.39824274152672423 [##########] 95.00%

spotpython tuning: 0.39824274152672423 [##########] 100.00% Done...

<spotpython.spot.spot.Spot at 0x3744084a0>
```

```
spot_2_GP.plot_progress()
```

```
spot_2_GP.print_results()
```

```
min y: 0.39824274152672423
x0: 3.1501084171824973
x1: 2.2710556115184426

[['x0', 3.1501084171824973], ['x1', 2.2710556115184426]]
```

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.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: 0.014772665252290174 [######----] 60.00%
```

```
spotpython tuning: 0.00020571435112063172 [#######---] 70.00%
```

```
spotpython tuning: 5.943319903709223e-08 [########--] 80.00%
```

```
spotpython tuning: 5.943319903709223e-08 [#########-] 90.00%
```

```
spotpython tuning: 5.943319903709223e-08 [##########] 100.00% Done...
```

10.3.1 Results

```
spot_1.print_results()
```

```
min y: 5.943319903709223e-08
x0: -0.0002437892512747275

[['x0', -0.0002437892512747275]]
```

```
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.Spot(fun=fun,
                      fun_control=fun_control,
                      design_control=design_control,
                      surrogate = S_GP)
spot_1_GP.run()
```

```
spotpython tuning: 0.00492567138682192 [####------] 40.00%
```

```
spotpython tuning: 0.0026120626229886095 [#####-----] 50.00%
```

```
spotpython tuning: 3.1732324790463464e-07 [######----] 60.00%
```

```
spotpython tuning: 3.291512456526943e-08 [#######---] 70.00%
```

```
spotpython tuning: 1.8087882373554217e-08 [########--] 80.00%
```

```
spotpython tuning: 2.4792327258527864e-09 [#########-] 90.00%
```

```
spotpython tuning: 2.4792327258527864e-09 [##########] 100.00% Done...
```

```
spot_1_GP.print_results()
```

```
min y: 2.4792327258527864e-09
x0: 4.979189417819718e-05

[['x0', 4.979189417819718e-05]]
```

```
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.Spot(fun=fun,
                   fun_control=fun_control,
                   design_control=design_control)
spot_2.run()
```

```
spotpython tuning: 3.1468336273020228 [######----] 55.00%
spotpython tuning: 3.1468336273020228 [######----] 60.00%
spotpython tuning: 3.1468336273020228 [######----] 65.00%
spotpython tuning: 3.1468336273020228 [#######---] 70.00%
spotpython tuning: 1.1486878851267175 [########--] 75.00%
spotpython tuning: 1.0238265839035492 [########--] 80.00%
spotpython tuning: 0.42056072017865986 [########--] 85.00%
spotpython tuning: 0.4019421180007683 [#########-] 90.00%
```

```
spotpython tuning: 0.39920553705190365 [##########] 95.00%
spotpython tuning: 0.39920553705190365 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x3773d1700>
```

- Print the results

```
spot_2.print_results()
```

```
min y: 0.39920553705190365
x0: 3.154894449957061
x1: 2.286298891852929
```

```
[['x0', 3.154894449957061], ['x1', 2.286298891852929]]
```

- 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.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...
```

```
<spotpython.spot.spot.Spot at 0x375c46600>
```

- Print the Results

```
spot_2_XGB.print_results()
```

```
min y: 1.3263745845108854
x0: -2.872730773493426
x1: 10.874313833535739
```

```
[['x0', -2.872730773493426], ['x1', 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.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: 0.014772665252290174 [######----] 60.00%
```

```
spotpython tuning: 0.00020571435112063172 [#######---] 70.00%
```

```
spotpython tuning: 5.943319903709223e-08 [########--] 80.00%
```

```
spotpython tuning: 5.943319903709223e-08 [#########-] 90.00%
```

```
spotpython tuning: 5.943319903709223e-08 [##########] 100.00% Done...
```

- Print the Results

```
spot_1.print_results()
```

```
min y: 5.943319903709223e-08
x0: -0.0002437892512747275
```

```
[['x0', -0.0002437892512747275]]
```

- 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.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...
```

```
spot_1_XGB.print_results()
```

```
min y: 0.008730885505764131
x0: 0.09343920754032609

[['x0', 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/pl
- 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_in
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 7.5, 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.Spot(
            fun=fun,
            fun_control=fun_control,
            design_control=design_control)
spot_1.run()
```

```
spotpython tuning: 4.959603317754042e-09 [####------] 44.00%
spotpython tuning: 4.959603317754042e-09 [#####-----] 48.00%
spotpython tuning: 4.959603317754042e-09 [#####-----] 52.00%
spotpython tuning: 4.959603317754042e-09 [######----] 56.00%
spotpython tuning: 1.7538785665210774e-10 [######----] 60.00%
spotpython tuning: 5.299979470384224e-12 [######----] 64.00%
spotpython tuning: 5.299979470384224e-12 [#######---] 68.00%
spotpython tuning: 5.299979470384224e-12 [#######---] 72.00%
spotpython tuning: 5.299979470384224e-12 [########--] 76.00%
spotpython tuning: 5.299979470384224e-12 [########--] 80.00%
spotpython tuning: 5.299979470384224e-12 [########--] 84.00%
spotpython tuning: 5.299979470384224e-12 [#########-] 88.00%
spotpython tuning: 5.299979470384224e-12 [#########-] 92.00%
spotpython tuning: 5.299979470384224e-12 [##########] 96.00%
spotpython tuning: 5.299979470384224e-12 [##########] 100.00% Done...

<spotpython.spot.spot.Spot at 0x340c84410>
```

12.1.2 Results

```
spot_1.print_results()
```

```
min y: 5.299979470384224e-12
x0: 2.302168427892326e-06

[['x0', 2.302168427892326e-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.Spot(fun=fun,
                      fun_control=fun_control)
spot_1_ei.run()
```

```
spotpython tuning: 7.188987729200596e-08 [####------] 44.00%
spotpython tuning: 7.188987729200596e-08 [#####-----] 48.00%
spotpython tuning: 7.188987729200596e-08 [#####-----] 52.00%
spotpython tuning: 7.188987729200596e-08 [######----] 56.00%
spotpython tuning: 5.939954122907979e-08 [######----] 60.00%
spotpython tuning: 1.3947524785731501e-08 [######----] 64.00%
spotpython tuning: 1.3947524785731501e-08 [#######---] 68.00%
spotpython tuning: 1.3947524785731501e-08 [#######---] 72.00%
spotpython tuning: 1.3947524785731501e-08 [########--] 76.00%
spotpython tuning: 1.3947524785731501e-08 [########--] 80.00%
spotpython tuning: 4.854409561808151e-10 [########--] 84.00%
spotpython tuning: 4.854409561808151e-10 [#########-] 88.00%
spotpython tuning: 2.611463411033466e-11 [#########-] 92.00%
spotpython tuning: 2.611463411033466e-11 [##########] 96.00%
spotpython tuning: 2.611463411033466e-11 [##########] 100.00% Done...

<spotpython.spot.spot.Spot at 0x343ace270>
```

```
spot_1_ei.plot_progress(log_y=True)
```

```
spot_1_ei.print_results()
```

```
min y: 2.611463411033466e-11
x0: 5.110247949985857e-06
```

```
[['x0', 5.110247949985857e-06]]
```

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.Spot(fun=fun,
                   fun_control=fun_control,
                   surrogate_control=surrogate_control)
spot_2_ei_noniso.run()
```

```
spotpython tuning: 1.8084153639415482e-05 [####------] 44.00%
spotpython tuning: 1.8084153639415482e-05 [#####-----] 48.00%
spotpython tuning: 1.8084153639415482e-05 [#####-----] 52.00%
spotpython tuning: 1.8084153639415482e-05 [######----] 56.00%
spotpython tuning: 1.8084153639415482e-05 [######----] 60.00%
spotpython tuning: 1.8084153639415482e-05 [######----] 64.00%
```

```
spotpython tuning: 1.8084153639415482e-05 [#######---] 68.00%
spotpython tuning: 1.8084153639415482e-05 [#######---] 72.00%
spotpython tuning: 1.8084153639415482e-05 [########--] 76.00%
spotpython tuning: 1.7065306452997463e-05 [########--] 80.00%
spotpython tuning: 1.7065306452997463e-05 [########--] 84.00%
spotpython tuning: 7.056740725219606e-06 [#########-] 88.00%
spotpython tuning: 7.056740725219606e-06 [#########-] 92.00%
spotpython tuning: 7.056740725219606e-06 [##########] 96.00%
spotpython tuning: 7.056740725219606e-06 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x343c969c0>
```

```
spot_2_ei_noniso.plot_progress(log_y=True)
```

```
spot_2_ei_noniso.print_results()
```

```
min y: 7.056740725219606e-06
x0: -0.0025062845492088196
x1: -0.0008804989969425007

[['x0', -0.0025062845492088196], ['x1', -0.0008804989969425007]]
```

```
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.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: 0.13771720111978272 [####------] 40.00%
spotpython tuning: 0.008762260544617942 [#####-----] 46.67%
spotpython tuning: 0.0028383641218116835 [#####-----] 53.33%
spotpython tuning: 0.0008109493206433335 [######----] 60.00%
spotpython tuning: 0.0003652470205973824 [#######---] 66.67%
spotpython tuning: 0.00036032551596159646 [#######---] 73.33%
spotpython tuning: 0.00035888041070709844 [########--] 80.00%
spotpython tuning: 0.00032677794812133767 [#########-] 86.67%
spotpython tuning: 0.0002726988496483528 [#########-] 93.33%
spotpython tuning: 0.00015304178060680248 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x34225b500>
```

```
spot_ei.plot_progress()
```

```
np.min(spot_1.y), np.min(spot_ei.y)
```

```
(5.299979470384224e-12, 0.00015304178060680248)
```

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.

```python
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.Spot(fun=fun,
                    fun_control=fun_control,
                    surrogate=S,
                    design_control=design_control)
spot_GP.run()
```

```
spotpython tuning: 24.51465459019188 [####------] 40.00%
spotpython tuning: 11.003101704408273 [#####-----] 46.67%
spotpython tuning: 11.003101704408273 [#####-----] 53.33%
```

```
spotpython tuning: 7.281515308693262 [######----] 60.00%
spotpython tuning: 7.281515308693262 [#######---] 66.67%
spotpython tuning: 7.281515308693262 [#######---] 73.33%
spotpython tuning: 2.9519942803779493 [########--] 80.00%
spotpython tuning: 2.9519942803779493 [#########-] 86.67%
spotpython tuning: 2.1049767654946914 [#########-] 93.33%
spotpython tuning: 1.9431498675678949 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x3893333b0>
```

```
spot_GP.y
```

```
array([ 69.32459936, 152.38491454, 107.92560483,  24.51465459,
        76.73500031,  86.30429136,  11.0031017 ,  16.11757292,
         7.28151531,  21.82295775,  10.96088904,   2.95199428,
         3.0290972 ,   2.10497677,   1.94314987])
```

```
spot_GP.plot_progress()
```

```
spot_GP.print_results()
```

```
min y: 1.9431498675678949
x0: 10.0
x1: 2.9999226833344648

[['x0', 10.0], ['x1', 2.9999226833344648]]
```

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.Spot(fun=fun,
                     fun_control=fun_control,
                     surrogate=S_K,
                     design_control=design_control,
                     surrogate_control=surrogate_control)
spot_S_K.run()
```

```
spotpython tuning: 0.13771716844457255 [##--------] 24.00%
spotpython tuning: 0.008763992988881779 [###-------] 28.00%
spotpython tuning: 0.0028324931031249094 [###-------] 32.00%
spotpython tuning: 0.0008161627041911008 [####------] 36.00%
spotpython tuning: 0.00036337249944152984 [####------] 40.00%
spotpython tuning: 0.0003620634598515755 [####------] 44.00%
spotpython tuning: 0.00036019126988579237 [#####-----] 48.00%
spotpython tuning: 0.0003310003348891555 [#####-----] 52.00%
spotpython tuning: 0.0002796810486585005 [######----] 56.00%
spotpython tuning: 0.00016668209586740844 [######----] 60.00%
spotpython tuning: 2.105184213458755e-05 [######----] 64.00%
spotpython tuning: 2.33449020805854e-06 [#######---] 68.00%
spotpython tuning: 7.112087478830758e-07 [#######---] 72.00%
spotpython tuning: 4.3581016990329405e-07 [########--] 76.00%
spotpython tuning: 3.9965568597157324e-07 [########--] 80.00%
spotpython tuning: 1.9740797347254742e-07 [########--] 84.00%
spotpython tuning: 1.6627705746993065e-07 [#########-] 88.00%
spotpython tuning: 1.6627705746993065e-07 [#########-] 92.00%
spotpython tuning: 1.6627705746993065e-07 [##########] 96.00%
spotpython tuning: 1.6627705746993065e-07 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x347b8e540>
```

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

```
spot_S_K.surrogate.plot()
```

```
spot_S_K.print_results()
```

```
min y: 1.6627705746993065e-07
x0: 0.0003245075243596407
x1: 0.00024692493617273215

[['x0', 0.0003245075243596407], ['x1', 0.00024692493617273215]]
```

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.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  ]
[[0.00163775 0.00441483]
 [0.00163816 0.00398502]]
```

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([-0.99002505]),
 '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="norm")
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([-1.10547468]),
 '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([4.89634048e-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
```

```python
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]]
[ 2.56406437e-01 -6.93071067e+01 -8.56027124e-02  8.93405931e+02
 -5.82561927e+02 -4.76028022e+01  7.16445311e+01  4.09512920e+01
  2.57319028e+02 -4.70871982e+02]
```

```
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]]
[0.712453   0.05595118 0.83735691 0.0106654  0.01413372 0.07074765
 0.05479457 0.07763503 0.02412205 0.01625354]
```

```
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: 0.1
```

```
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 Lambda sear
```

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_in

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.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: 0.034752873669989026 [####------] 40.00%
```

```
spotpython tuning: 0.03230817945928789 [#####-----] 50.00%
```

```
spotpython tuning: 0.015578418800855254 [######----] 60.00%
```

```
spotpython tuning: 0.0009550994714026289 [#######---] 70.00%
```

```
spotpython tuning: 5.561590542963861e-05 [########--] 80.00%
```

```
spotpython tuning: 7.181090066713707e-07 [#########-] 90.00%
```

```
spotpython tuning: 4.5254143126895086e-07 [##########] 100.00% Done...
```

13.3 Print the Results

```
spot_1_noisy.print_results()
```

```
min y: 4.5254143126895086e-07
min mean y: 4.5254143126895086e-07
x0: 0.0006727119972684825

[['x0', 0.0006727119972684825]]
```

```
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
```

```
0.0005592051705322895
```

- 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_in

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.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: 0.00891934134603014 [####------] 40.00%
```

```
spotpython tuning: 0.00891934134603014 [#####-----] 50.00%
```

```
spotpython tuning: 6.820812967131544e-05 [######----] 60.00%
```

```
spotpython tuning: 3.692698488681066e-06 [#######---] 70.00%
```

```
spotpython tuning: 1.4235007051487162e-06 [########--] 80.00%
```

```
spotpython tuning: 1.4235007051487162e-06 [#########-] 90.00%
```

```
spotpython tuning: 1.4235007051487162e-06 [##########] 100.00% Done...
```

14.2 Print the Results

```
spot_1_noisy.print_results()
```

```
min y: 1.4235007051487162e-06
min mean y: 1.4235007051487162e-06
x0: -0.0011931054878545804
```

```
[['x0', -0.0011931054878545804]]
```

```
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
```

```
9.867760027597887e-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.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: 1.4886245843852055e-05 [##########] 100.00% Done...

<spotpython.spot.spot.Spot at 0x16cc5c620>
```

15.1.2 Results

```
spot_2.print_results()
```

```
min y: 1.4886245843852055e-05
x0: -0.003850983818095356
x1: -0.00023700100552480994

[['x0', -0.003850983818095356], ['x1', -0.00023700100552480994]]
```

```
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.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: 1.4886245843852055e-05 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x16c80c320>
```

- 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: 1.4886245843852055e-05
x0: -0.003850983818095356
x1: -0.00023700100552480994

[['x0', -0.003850983818095356], ['x1', -0.00023700100552480994]]
```

```
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.Spot(fun=fun,
                     fun_control=fun_control,
                     surrogate_control=surrogate_control)
spot_2_pm.run()
```

```
spotpython tuning: 2.0901803896651035e-05 [#######---] 73.33%
spotpython tuning: 2.0901803896651035e-05 [########--] 80.00%
spotpython tuning: 2.0901803896651035e-05 [#########-] 86.67%
spotpython tuning: 2.0901803896651035e-05 [#########-] 93.33%
spotpython tuning: 9.862120336177949e-06 [##########] 100.00% Done...

<spotpython.spot.spot.Spot at 0x36f2fade0>
```

```
spot_2_pm.plot_progress(log_y=True)
```

```
spot_2_pm.print_results()
```

```
min y: 9.862120336177949e-06
x0: -0.002974641180991332
x1: -0.0010067920244660482

[['x0', -0.002974641180991332], ['x1', -0.0010067920244660482]]
```

```
spot_2_pm.surrogate.plot()
```

```
spot_2_pm.surrogate.p
```

```
array([1.66198193])
```

15.4 Optimization of Multiple p Values

```
surrogate_control = surrogate_control_init(n_p=2,
                                           optim_p=True)
spot_2_pmo= spot.Spot(fun=fun,
                      fun_control=fun_control,
                      surrogate_control=surrogate_control)
spot_2_pmo.run()
```

```
spotpython tuning: 1.90602728663205e-05 [#######---] 73.33%
spotpython tuning: 1.90602728663205e-05 [########--] 80.00%
spotpython tuning: 1.90602728663205e-05 [#########-] 86.67%
spotpython tuning: 1.90602728663205e-05 [#########-] 93.33%
spotpython tuning: 5.741913594811217e-06 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x375138470>
```

```
spot_2_pmo.plot_progress(log_y=True)
```

```
spot_2_pmo.print_results()
```

```
min y: 5.741913594811217e-06
x0: -0.002070805151396577
x1: -0.001205686368737997

[['x0', -0.002070805151396577], ['x1', -0.001205686368737997]]
```

```
spot_2_pmo.surrogate.plot()
```

```
spot_2_pmo.surrogate.p
```

```
array([1.35351306, 1.87234959])
```

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

Part III

Data-Driven Modeling and Optimization

16 Data-Driven Modeling and Optimization

16.1 StatQuest Videos

16.1.1 June, 11th 2024

16.1.1.1 Histograms

- Video: Histograms, Clearly Explained

Exercise 16.1 (Histograms)**.** Problems with histograms?

16.1.1.2 Probability Distributions

- Video: The Main Ideas behind Probability Distributions

Exercise 16.2 (Smaller Bins)**.** What happens when we use smaller bins in a histogram?

Exercise 16.3 (Density Curve)**.** Why plot a curve to approximate a histogram?

16.1.1.3 Normal Distribution

- Video: The Normal Distribution, Clearly Explained!!!

Exercise 16.4 (TwoSDQuestion)**.** How many samples are plus/minus two SD around the mean?

Exercise 16.5 (OneSDQuestion)**.** How many samples are plus/minus one SD around the mean?

Exercise 16.6 (ThreeSDQuestion)**.** How many samples are plus/minus three SD around the mean?

Exercise 16.7 (DataRangeQuestion)**.** You have a mean at 100 and a SD of 10. Where are 95% of the data?

Exercise 16.8 (PeakHeightQuestion)**.** If the peak is very high, is the SD low or high?

16.1.1.4 The mean, the media, and the mode

16.1.1.5 The exponential distribution

16.1.1.6 Population and Estimated Parameters

- Video: Population and Estimated Parameters, Clearly Explained

Exercise 16.9 (ProbabilityQuestion)**.** If we have a certain curve and want to calculate the probability of values equal to 20 if the mean is 20.

16.1.1.7 Mean, Variance, and Standard Deviation

- Video: Calculating the Mean, Variance, and Standard Deviation

Exercise 16.10 (MeanDifferenceQuestion)**.** The difference between μ and x-bar?

Exercise 16.11 (EstimateMeanQuestion)**.** How do you calculate the sample mean?

Exercise 16.12 (SigmaSquaredQuestion)**.** What is sigma squared?

Exercise 16.13 (EstimatedSDQuestion)**.** What is the formula for the estimated standard deviation?

Exercise 16.14 (VarianceDifferenceQuestion)**.** Difference between the variance and the estimated variance?

16.1.2 Mathematical Models

- Video: What is a mathematical model?

Exercise 16.15 (ModelBenefitsQuestion)**.** What are the benefits of using models?

16.1.2.1 Sampling from a Distribution

- Vidoe: Sampling from a Distribution, Clearly Explained!!!

Exercise 16.16 (SampleDefinitionQuestion)**.** What is a sample in statistics?

16.1.3 Hypothesis Testing and the Null-Hypothesis

Exercise 16.17 (RejectHypothesisQuestion)**.** What does it mean to reject a hypothesis?

Exercise 16.18 (NullHypothesisQuestion)**.** What is a null hypothesis?

Exercise 16.19 (BetterDrugQuestion)**.** How can you show that you have found a better drug?

16.1.3.1 Alternative Hypotheses, Main Ideas

16.1.3.2 p-values: What they are and how to interpret them

Exercise 16.20 (PValueIntroductionQuestion)**.** What is the reason for introducing the p-value?

Exercise 16.21 (PValueRangeQuestion)**.** Is there any range for p-values? Can it be negative?

Exercise 16.22 (PValueRangeQuestion)**.** Is there any range for p-values? Can it be negative?

Exercise 16.23 (TypicalPValueQuestion)**.** What are typical values of the p-value and what does it mean? 5%?

Exercise 16.24 (FalsePositiveQuestion)**.** What is a false-positive?

16.1.3.3 How to calculate p-values

Exercise 16.25 (CalculatePValueQuestion)**.** How to calculate p-value?

Exercise 16.26 (SDCalculationQuestion)**.** What is the SD if the mean is 155 and in the range from 142 - 169 there are 95% of the data?

Exercise 16.27 (SidedPValueQuestion)**.** When do we need the two-sided p-value and when the one-sided?

Exercise 16.28 (CoinTestQuestion)**.** Test a coin with Tail-Head-Head. What is the p-value?

Exercise 16.29 (BorderPValueQuestion)**.** If you get exactly the 0.05 border value, can you reject?

Exercise 16.30 (OneSidedPValueCautionQuestion)**.** Why should you be careful with a one-sided p-test?

Exercise 16.31 (BinomialDistributionQuestion)**.** What is the binomial distribution?

16.1.3.4 p-hacking: What it is and how to avoid it

Exercise 16.32 (PHackingWaysQuestion)**.** Name two typical ways of p-hacking.

Exercise 16.33 (AvoidPHackingQuestion)**.** How can p-hacking be avoided?

Exercise 16.34 (MultipleTestingProblemQuestion)**.** What is the multiple testing problem?

16.1.3.5 Covariance

Exercise 16.35 (CovarianceDefinitionQuestion)**.** What is covariance?

Exercise 16.36 (CovarianceMeaningQuestion)**.** What is the meaning of covariance?

Exercise 16.37 (CovarianceVarianceRelationshipQuestion)**.** What is the relationship between covariance and variance?

Exercise 16.38 (HighCovarianceQuestion)**.** If covariance is high, is there a strong relationship?

Exercise 16.39 (ZeroCovarianceQuestion)**.** What if the covariance is zero?

Exercise 16.40 (NegativeCovarianceQuestion)**.** Can covariance be negative?

Exercise 16.41 (NegativeVarianceQuestion)**.** Can variance be negative?

16.1.3.6 Pearson's Correlation

Video: [Pearson's Correlation, Clearly Explained]

Exercise 16.42 (CorrelationValueQuestion)**.** What do you do if the correlation value is 10?

Exercise 16.43 (CorrelationRangeQuestion)**.** What is the possible range of correlation values?

Exercise 16.44 (CorrelationFormulaQuestion)**.** What is the formula for correlation?

16.1.3.7 Boxplots

16.1.4 June, 18th 2024

16.1.4.1 Statistical Power

- Video: Statistical Power, Clearly Explained

Exercise 16.45 (UnderstandingStatisticalPower)**.** What is the definition of power in a statistical test?

Exercise 16.46 (DistributionEffectOnPower)**.** What is the implication for power analysis if the samples come from the same distribution?

Exercise 16.47 (IncreasingPower)**.** How can you increase the power if the distributions are very similar?

Exercise 16.48 (PreventingPHacking)**.** What should be done to avoid p-hacking when the distributions are close to each other?

Exercise 16.49 (SampleSizeAndPower)**.** If there is overlap and the sample size is small, will the power be high or low?

16.1.4.2 Power Analysis

- Video: Power Analysis, Clearly Explained!!!

Exercise 16.50 (FactorsAffectingPower)**.** Which are the two main factors that affect power?

Exercise 16.51 (PurposeOfPowerAnalysis)**.** What does power analysis tell us?

Exercise 16.52 (ExperimentRisks)**.** What are the two risks faced when performing an experiment?

Exercise 16.53 (PerformingPowerAnalysis)**.** How do you perform a power analysis?

16.1.4.3 The Central Limit Theorem

- Video: The Central Limit Theorem, Clearly Explained!!!

Exercise 16.54 (CentralLimitTheoremExplanation)**.** What does the Central Limit Theorem state?

16.1.4.4 Boxplots

- Video: Boxplots are Awesome

Exercise 16.55 (MedianInBoxplot)**.** What is represented by the middle line in a boxplot?

Exercise 16.56 (BoxContentInBoxplot)**.** What does the box in a boxplot represent?

16.1.4.5 R-squared

- Video: R-squared, Clearly Explained

Exercise 16.57 (RSquaredDefinition)**.** What is R-squared? Show the formula.

Exercise 16.58 (NegativeRSquared)**.** Can the R-squared value be negative?

Exercise 16.59 (RSquaredCalculation)**.** Perform a calculation involving R-squared.

16.1.4.6 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 16.60 (LeastSquaresMeaning)**.** What is the meaning of the least squares method?

16.1.4.7 Linear Regression

- Video: Linear Regression, Clearly Explained

16.1.4.8 Multiple Regression

- Video: Multiple Regression, Clearly Explained

16.1.4.9 A Gentle Introduction to Machine Learning

- Video: A Gentle Introduction to Machine Learning

Exercise 16.61 (RegressionVsClassification)**.** What is the difference between regression and classification?

16.1.4.10 Maximum Likelihood

- Video: Maximum Likelihood, clearly explained!!!

Exercise 16.62 (LikelihoodConcept)**.** What is the idea of likelihood?

- Video: Probability is not Likelihood. Find out why!!!

Exercise 16.63 (ProbabilityVsLikelihood)**.** What is the difference between probability and likelihood?

16.1.4.11 Cross-Validation

- Video: Machine Learning Fundamentals: Cross Validation

Exercise 16.64 (TrainVsTestData)**.** What is the difference between training and testing data?

Exercise 16.65 (SingleValidationIssue)**.** What is the problem if you validate the model only once?

Exercise 16.66 (FoldDefinition)**.** What is a fold in cross-validation?

Exercise 16.67 (LeaveOneOutValidation)**.** What is leave-one-out cross-validation?

16.1.4.12 The Confusion Matrix

- Video: Machine Learning Fundamentals: The Confusion Matrix

Exercise 16.68 (DrawingConfusionMatrix)**.** Draw the confusion matrix.

16.1.4.13 Sensitivity and Specificity

- Video: Machine Learning Fundamentals: Sensitivity and Specificity

Exercise 16.69 (SensitivitySpecificityCalculation1)**.** Calculate the sensitivity and specificity for a given confusion matrix.

Exercise 16.70 (SensitivitySpecificityCalculation2)**.** Calculate the sensitivity and specificity for a given confusion matrix.

16.1.4.14 Bias and Variance

- Video: Machine Learning Fundamentals: Bias and Variance

Exercise 16.71 (BiasAndVariance)**.** What are bias and variance?

16.1.4.15 Mutual Information

- Video: Mutual Information, Clearly Explained

Exercise 16.72 (MutualInformationExample)**.** Provide an example and calculate if mutual information is high or low.

16.1.5 June, 25th 2024

16.1.5.1 Principal Component Analysis (PCA)

- Video: Principal Component Analysis (PCA), Step-by-Step

Exercise 16.73 (WhatIsPCA)**.** What is PCA?

Exercise 16.74 (ScreePlotExplanation)**.** What is a scree plot?

- Vidoe: PCA - Practical Tips

Exercise 16.75 (LeastSquaresInPCA)**.** Does PCA use least squares?

Exercise 16.76 (PCASteps)**.** Which steps are performed by PCA?

Exercise 16.77 (EigenvaluePC1)**.** What is the eigenvalue of the first principal component?

Exercise 16.78 (DifferencesBetweenPoints)**.** Are the differences between red and yellow the same as the differences between red and blue points?

- Video: PCA in Python

Exercise 16.79 (ScalingInPCA)**.** How to scale data in PCA?

Exercise 16.80 (DetermineNumberOfComponents)**.** How to determine the number of principal components?

Exercise 16.81 (LimitingNumberOfComponents)**.** How is the number of principal components limited?

16.1.6 t-SNE

- Video: t-SNE, Clearly Explained

Exercise 16.82 (WhyUseTSNE)**.** Why use t-SNE?

Exercise 16.83 (MainIdeaOfTSNE)**.** What is the main idea of t-SNE?

Exercise 16.84 (BasicConceptOfTSNE)**.** What is the basic concept of t-SNE?

Exercise 16.85 (TSNESteps)**.** What are the steps in t-SNE?

16.1.7 K-means clustering

- Video: K-means clustering

Exercise 16.86 (HowKMeansWorks)**.** How does K-means clustering work?

Exercise 16.87 (QualityOfClusters)**.** How can the quality of the resulting clusters be calculated?

Exercise 16.88 (IncreasingK)**.** Why is it not a good idea to increase k too much?

16.1.8 DBSCAN

- Video: Clustering with DBSCAN, Clearly Explained!!!

Exercise 16.89 (CorePointInDBSCAN)**.** What is a core point in DBSCAN?

Exercise 16.90 (AddingVsExtending)**.** What is the difference between adding and extending in DBSCAN?

Exercise 16.91 (OutliersInDBSCAN)**.** What are outliers in DBSCAN?

16.1.9 K-nearest neighbors

- Video: StatQuest: K-nearest neighbors, Clearly Explained

Exercise 16.92 (AdvantagesAndDisadvantagesOfK)**.** What are the advantages and disadvantages of $k = 1$ and $k = 100$ in K-nearest neighbors?

16.1.10 Naive Bayes

- Video: Naive Bayes, Clearly Explained!!!

Exercise 16.93 (NaiveBayesFormula)**.** What is the formula for Naive Bayes?

Exercise 16.94 (CalculateProbabilities)**.** Calculate the probabilities for a given example using Naive Bayes.

16.1.11 Gaussian Naive Bayes

- Video: Gaussian Naive Bayes, Clearly Explained!!!

Exercise 16.95 (UnderflowProblem)**.** Why is underflow a problem in Gaussian Naive Bayes?

16.1.12 July, 2nd 2024

16.1.12.1 Decision and Classification Trees, Clearly Explained

16.1.12.2 StatQuest: Decision Trees, Part 2 - Feature Selection and Missing Data

16.1.12.3 Regression Trees, Clearly Explained!!!

16.1.12.4 How to Prune Regression Trees, Clearly Explained!!!

16.1.12.5 Trees

Exercise 16.96 (Tree Usage)**.** For what can we use trees?

16.1.12.6 Decision Trees

Exercise 16.97 (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 16.98 (Tree Feature Importance)**.** Is the most or least important feature on top?

Exercise 16.99 (Tree Feature Imputation)**.** How can you fill a gap/missing data?

Solution 16.1 (Tree Feature Imputation)*.*

- Mean
- Median
- Comparing to column with high correlation

16.1.12.7 Regression Trees

Exercise 16.100 (Regression Tree Limitations)**.** What are limitations?

Exercise 16.101 (Regression Tree Score)**.** How is the tree score calculated?

Exercise 16.102 (Regression Tree Alpha Value Small)**.** What can we say about the tree if the alpha value is small?

Exercise 16.103 (Regression Tree Increase Alpha Value)**.** What happens if you increase alpha?

Exercise 16.104 (Regression Tree Pruning)**.** What is the meaning of pruning?

16.1.13 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!

16.2 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.

16.2.1 Opening Remarks and Examples

- Artificial Intelligence (AI)
- Machine learning (ML)
- Deep Learning (DL)

Figure 16.1: 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.

16.2.1.1 Supervised and Unsupervised Learning

Two important types: supervised and unsupervised learning. There is even more, e.g., semi-supervised learning.

16.2.1.1.1 Starting point

- Outcome measurement Y (dependent variable, response, target).
- Vector of p predictor measurements X (inputs, regressors, covariates, features, independent variables).
- Training data $(x_1, y1), \ldots, (x_N, y_N)$. 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).

16.2.1.1.2 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.

16.3 Basics

16.3.1 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 boundaries
probabilities = n * bin_width  # n is already normalized to form a probability density if `de

# 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 16.2

```python
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: 0.0050
Bin 2 Probability: 0.0320
Bin 3 Probability: 0.0610
Bin 4 Probability: 0.1320
Bin 5 Probability: 0.2050
Bin 6 Probability: 0.2560
Bin 7 Probability: 0.1710
Bin 8 Probability: 0.0950
Bin 9 Probability: 0.0320
Bin 10 Probability: 0.0090
```

```
Bin 11 Probability: 0.0020
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.

16.3.2 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.

16.3.2.1 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.

16.3.2.2 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), colo
plt.xlabel('Height (cm)')
plt.ylabel('Probability Density')
plt.show()
```

Figure 16.3: 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.

16.3.3 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.

16.3.3.1 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:

$$P(X = 1) = 1 - P(X = 0) = p = 1 - q.$$

16.3.3.2 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.

16.3.3.3 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:.3f}")
```

```
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 16.1 (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

$$p(x) = \binom{n}{k}p^x(1-p)^{n-x} \qquad x = 0, 1, \ldots, n. \tag{16.1}$$

The mean μ and the variance σ^2 of the binomial distribution are

$$\mu = np \tag{16.2}$$

and

$$\sigma^2 = np(1-p). \tag{16.3}$$

Note, the Bernoulli distribution is simply Binomial(1,p).

16.4 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.

16.4.1 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.

$$f(x) = \frac{d}{dx}F(x) \tag{16.4}$$

16.4.2 Expectation (Continuous)

Definition 16.2 (Expectation (Continuous))**.**

$$\mathrm{E}(X) = \int_{-\infty}^{\infty} x f(x)\, dx \tag{16.5}$$

16.4.3 Variance and Standard Deviation (Continuous)

Definition 16.3 (Variance (Continuous))**.** Variance can be calculated with $\mathrm{E}(X)$ and

$$\mathrm{E}(X^2) = \int_{-\infty}^{\infty} x^2 f(x)\, dx \tag{16.6}$$

as

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

□

Definition 16.4 (Standard Deviation (Continuous))**.** Standard deviation can be calculated as

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

□

16.4.4 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

$$f(x) = \begin{cases} 1/(b-a) & \text{if } a < x < b, \\ 0 & \text{otherwise} \end{cases} \tag{16.7}$$

16.4.5 Normal Distribution

Definition 16.5 (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*

$$\varphi(z) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z^2}{2}\right). \tag{16.8}$$

An important application of the standardization introduced in Equation 16.8 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

$$X = \mu + \sigma Z \text{ where } Z \sim N(0,1). \tag{16.9}$$

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 16.4.

```
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 16.4: 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 16.5.

```
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
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 plo
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 16.5: Normal Distribution Cumulative Distribution Function

16.4.6 The Mean, the Median, and the Mode

16.4.7 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.

16.4.8 Population and Estimated Parameters

16.4.9 Calculating the Mean, Variance, and Standard Deviation

16.4.10 What is a Mathematical Model?

16.4.11 Sampling from a Distribution

16.4.12 Hypothesis Testing and the Null Hypothesis

16.4.13 Alternative Hypotheses

16.4.14 p-values: What They Are and How to Interpret Them

16.4.15 How to Calculate p-values

16.4.16 p-hacking: What It Is and How to Avoid It

16.4.17 Covariance

16.4.18 Pearson's Correlation

16.4.19 Boxplots

16.4.20 R-squared

16.4.21 The Main Ideas of Fitting a Line to Data

16.4.22 Linear Regression

16.4.23 Multiple Regression

16.5 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 16.105.

- Do children learn supervised?
- When do you learn supervised?
- Can learning be unsupervised?

16.5.0.0.1 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.

16.5.0.0.2 Statistical Learning

We consider supervised learning first.

Figure 16.6: Sales as a function of TV, radio and newspaper. Taken from James et al. (2014)

Sales figures from a marketing campaign, see Figure 16.6. 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.

16.5.1 Statistical Learning and Regression

16.5.1.1 Regression Function

Consider Figure 16.7. 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 16.7: 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)

16.5.2 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 $X = x$.

16.5.2.1 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 16.7.

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.

16.5.2.2 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 16.8.

Figure 16.8: 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.

16.5.3 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 16.9. 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 16.9: 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 16.10: A linear model $\hat{f}_L$ gives a reasonable fit. Taken from James et al. (2014)

Figure 16.11: 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 16.10. 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 16.11.

Figure 16.12 shows a simulated example. Red points are simulated values for income from the model

$$income = f(education, seniority) + \epsilon$$

Figure 16.12: The true model. Red points are simulated values for income from the model, f is the blue surface. Taken from James et al. (2014)

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**.

16.5.3.1 Trade-offs

- Prediction accuracy versus interpretability: Linear models are easy to interpret; thin-plate splines are not.

Figure 16.13: Linear regression fit to the simulated data (red points). Taken from James et al. (2014)

- Good fit versus over-fit or under-fit: How do we know when the fit is just right?
- 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 16.16.

16.5.4 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 16.14: Thin-plate spline models $\hat{f}_S(education, seniority)$ fitted to the model from Figure 16.12. Taken from James et al. (2014)

16.5.4.1 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

$$E\left(y_0 - \hat{f}(x_0)\right)^2 = \text{var}(\hat{f}(x_0)) + [Bias(\hat{f}(x_0))]^2 + \text{var}(\epsilon). \tag{16.10}$$

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

$$Bias(\hat{f}(x_0)) = E[\hat{f}(x_0)] - f(x_0).$$

Figure 16.15: Thin-plate spline models $\hat{f}_S(education, seniority)$ fitted to the model from Figure 16.12. The model makes no errors on the training data (overfitting). Taken from James et al. (2014)

Figure 16.16: Interpretability versus flexibility. Flexibility corresponds with the number of model parameters. Taken from James et al. (2014)

Figure 16.17: 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 16.18: 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 16.19: 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)

Figure 16.20: Bias-variance trade-off for the three examples. Taken from James et al. (2014)

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 16.20.

If we add the two components (reducible and irreducible error), we get the MSE in Figure 16.20 as can be seen in Equation 16.10.

16.5.5 Classification Problems and K-Nearest Neighbors

In classification we have a qualitative response variable.

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

Figure 16.21: Classification. Taken from James et al. (2014)

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 $1, 2, \ldots, K$. Let

$$p_k(x) = Pr(Y = k|X = x), k = 1, 2, \ldots, K.$$

These are the **conditional class probabilities** at x; e.g. see little barplot at $x = 5$. Then the **Bayes optimal classifier** at x is

$$C(x) = j \qquad \text{if } p_j(x) = \max\{p_1(x), p_2(x), \ldots, p_K(x)\}.$$

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 $\hat{p}_k(x)$, $k = 1, \ldots, K$.

16.5.5.1 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).

Figure 16.22: Classification. Taken from James et al. (2014)

16.5.6 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.

K large means higher bias, so $1/K$ is chosen, because we go from low to high complexity on the x-error, see Figure 16.26. Horizontal dotted line is the base error.

Figure 16.23: K-nearest neighbors in two dimensions. Taken from James et al. (2014)

Figure 16.24: K-nearest neighbors in two dimensions. Taken from James et al. (2014)

Figure 16.25: K-nearest neighbors in two dimensions. Taken from James et al. (2014)

Figure 16.26: K-nearest neighbors classification error. Taken from James et al. (2014)

16.5.7 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:

$$D(X, Y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}.$$

16.5.8 Unsuperivsed Learning: Classification

16.5.8.1 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:

- 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 16.27, Figure 16.28, Figure 16.29, and Figure 16.30.

Figure 16.27: 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 16.28: 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 16.29: 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 16.30: 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

17 Machine Learning and Artificial Intelligence

17.1 Jupyter Notebooks

- The Jupyter-Notebook version of this file can be found here: malai.ipynb

17.2 Videos

17.2.1 June, 11th 2024

- Happy Halloween (Neural Networks Are Not Scary)
- The Essential Main Ideas of Neural Networks

17.2.2 June, 18th 2024

- The Chain Rule
- Gradient Descent, Step-by-Step
- Neural Networks Pt. 2: Backpropagation Main Ideas

17.2.2.1 Gradient Descent

Exercise 17.1 (GradDescStepSize)**.** How is the step size calculated?

Exercise 17.2 (GradDescIntercept)**.** How to calculate the new intercept?

Exercise 17.3 (GradDescIntercept)**.** When does the gradient descend stop?

17.2.2.2 Backpropagation

Exercise 17.4 (ChainRuleAndGradientDescent)**.** What are the key components involved in backpropagation?

Exercise 17.5 (BackpropagationNaming)**.** Why is it called backpropagation?

17.2.2.3 ReLU

Exercise 17.6 (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

17.2.2.4 PyTorch Links

- StatQuest: Introduction to Coding Neural Networks with PyTorch
- ML-AI Pytorch Introduction

17.2.3 June, 25th 2024

17.2.4 CNNs

17.2.4.1 Neural Networks Part 8: Image Classification with Convolutional Neural Networks (CNNs)

Exercise 17.7 (CNNImageRecognition)**.** Why are classical neural networks poor at image recognition?

Exercise 17.8 (CNNFiltersInitialization)**.** How are the filter values in CNNs initialized and optimized?

Exercise 17.9 (CNNFilterInitialization)**.** How are the filter values determined in Convolutional Neural Networks (CNNs)?

Exercise 17.10 (GenNNStockPrediction)**.** What is a limitation of using classical neural networks for stock market prediction?

17.2.5 RNN

17.2.5.1 Recurrent Neural Networks (RNNs), Clearly Explained!!!

Exercise 17.11 (RNNUnrolling)**.** How does the unrolling process work in Recurrent Neural Networks (RNNs)?

Exercise 17.12 (RNNReliability)**.** Why do Recurrent Neural Networks (RNNs) sometimes fail to work reliably?

17.2.6 LSTM

17.2.6.1 Long Short-Term Memory (LSTM), Clearly Explained

Exercise 17.13 (LSTMSigmoidTanh)**.** What are the differences between the sigmoid and tanh activation functions?

Exercise 17.14 (LSTMSigmoidTanh)**.** What is the ?

Exercise 17.15 (LSTMGates)**.** What are the gates in an LSTM network and their functions?

Exercise 17.16 (LSTMLongTermInfo)**.** In which gate is long-term information used in an LSTM network?

Exercise 17.17 (LSTMUpdateGates)**.** In which Gates is it updated in an LSTM?

17.2.7 Pytorch/Lightning

17.2.7.1 Introduction to Coding Neural Networks with PyTorch and Lightning

Exercise 17.18 (PyTorchRequiresGrad)**.** What does `requires_grad` mean in PyTorch?

17.2.8 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!!!

17.2.8.1 Embeddings

Exercise 17.19 (NN Strings)**.** Can neural networks process strings?

Exercise 17.20 (Embedding Definition)**.** What is the meaning of word embedding?

Exercise 17.21 (Embedding Dimensions)**.** Why do we need high dimension in word embedding?

17.2.8.2 Sequence to Sequence

Exercise 17.22 (LSTM)**.** Why are LSTMs used?

Exercise 17.23 (Teacher Forcing)**.** Why is teacher forcing used?

Exercise 17.24 (Attention)**.** What is the idea of attention?

17.2.9 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

17.2.9.1 Transformers

Exercise 17.25 (ChatGPT)**.** What kind of transformer does ChatGPT use?

Exercise 17.26 (Translation)**.** What kind of NN are used for translation?

Exercise 17.27 (Difference Encoder-Decoder and Decoder Only.)**.** What is the encoder-decoder transformer and the decoder only transformer?

Exercise 17.28 (Weights)**.** How are the weights initialized (a) and trained (b)?

Exercise 17.29 (Order of Words)**.** How is the word order preserved?

Exercise 17.30 (Relationship Between Words)**.** How is the relationship between words modeled?

Exercise 17.31 (Masked Self Attention)**.** What is masked self-attention?

Exercise 17.32 (Softmax)**.** Why is Softmax used to calculate percentage of similarities?

Exercise 17.33 (Softmax Output)**.** How is the percentage output of softmax in Transformers used?

Exercise 17.34 (V´s)**.** What is done with the scaled V´s that we get for each token so far (example: "is","what")?

Exercise 17.35 (Residual Connections)**.** What are residual connections?

Exercise 17.36 (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 17.37 (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"))

17.2.10 Additional Videos

- The SoftMax Derivative, Step-by-Step!!!
- Neural Networks Part 6: Cross Entropy
- Neural Networks Part 7: Cross Entropy Derivatives and Backpropagation

17.2.11 All Videos in a Playlist

- Full Playlist ML-AI

17.3 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 reproducabili
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 fu
from torch.optim import SGD # optim contains many optimizers. Here, we're using SGD, stochas

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.

17.3.1 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 to ini

        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 optimize the
        ## requires_grad, which is short for "requires gradient", to True. Since we don't ne
        ## parameters now, we set requires_grad=False.
        ##
        ## NOTE: Because our neural network is already fit to the data, we will input specif
        ## for each weight and bias. In contrast, if we had not already fit the neural netwo
        ## 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
                              ## illustrated at the top of this notebook.

        ## the next three lines implement the top of the neural network (using the top node i
        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
        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 fina
        input_to_final_relu = scaled_top_relu_output + scaled_bottom_relu_output + self.final

        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.)
```

17.3.2 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 as a l
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.

17.3.3 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 initia

        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 backpropagatio
        ## 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 gradient
        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 + self.final
```

```
        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 P
sns.lineplot(x=input_doses,
             y=output_values.detach(), ## NOTE: because final_bias has a gradident, we call d
                                       ## to return a new tensor that only has the value and
             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 train the
                                            ## NOTE: There are a bunch of different ways to c
                                            ## In this example, we'll use Stochastic Gradient
                                            ## another popular algortihm is Adam (which will

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 data is ca
for epoch in range(100):

    ## we create and initialize total_loss for each epoch so that we can evaluate how well mo
```

```
    ## training data. At first, when the model doesn't fit the training data very well, total
    ## will be large. However, as gradient descent improves the fit, total_loss will get smal
    ## If total_loss gets really small, we can decide that the model fits the data well enoug
    ## optimizing the fit. Otherwise, we can just keep optimizing until we reach the maximum
    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

        output_i = model(input_i) ## calculate the neural network output for the input (the s

        loss = (output_i - label_i)**2 ## calculate the loss for the single value.
                                       ## NOTE: Because output_i = model(input_i), "loss" has
                                       ## and the derivative (calculated in the next step) is
                                       ## in "model".

        loss.backward() # backward() calculates the derivative for that single value and adds

        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
                          ## and we took advantage of this process to calculate the derivativ
                          ## NOTE: "optimizer" has access to "model" because of how it was cr
                          ## (made earlier): optimizer = SGD(model.parameters(), lr=0.1).
                          ## ALSO NOTE: Alternatively, we can zero out the gradient with mode
    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 as a 
sns.lineplot(x=input_doses,
     y=output_values.detach(), ## NOTE: we call detach() because final_bias has a gradient
     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.

17.4 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 fu
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 da
```

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 initialize the t
        ## Here we can do it 2 different ways 1) Normal Distribution and 2) Uniform Distribut
        ## 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 percent
        ## 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 new
        ## 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 i
        ## 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 memory
        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 determines what
        ##    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 each
        ## that we have. forward() also keeps track of the long and short-term memories afte
        ## 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 with c0,
        short_memory = 0 # short term memory is also called "hidden state" and indexed with
        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 backpr
        # return Adam(self.parameters(), lr=0.1) # NOTE: Setting the learning rate to 0.1 tra
                                                 # using the default learning rate, lr=0.001
                                                 # training. However, if we use the default v
                                                 # the exact same Weights and Biases that I u
                                                 # the LSTM Clearly Explained StatQuest video
                                                 # 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 C
        ## For Company A, the goal is to predict that the value on Day 5 = 0, and for Company
        ## the goal is to predict that the value on Day 5 = 1. We use label_i, the value we w
        ## 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.

17.4.1 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
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.])).detac
print("Company B: Observed = 1, Predicted =", model(torch.tensor([1., 0.5, 0.25, 1.])).detac
```

```
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.

17.4.2 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" = "mos
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=3000) # Before, max_epochs=2000, so, by setting it to 3000, we
## 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/
```

```
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" = "mos
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
## 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/
```

```
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.])).detac
print("Company B: Observed = 1, Predicted =", model(torch.tensor([1., 0.5, 0.25, 1.])).detac
```

```
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)
```

17.5 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 ini

        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 the
        ##               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 t
        prediction = lstm_out[-1]
        return prediction


    def configure_optimizers(self): # this configures the optimizer we want to use for backpr
        return Adam(self.parameters(), lr=0.1) ## we'll just go ahead and set the learning ra
```

```
    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.])).detac
print("Company B: Observed = 1, Predicted =", model(torch.tensor([1., 0.5, 0.25, 1.])).detac
```

```
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 faster.
## Before, with the hand made LSTM and the default learning rate, 0.001, it took about 5000 e
## the model. Now, with the learning rate set to 0.1, we only need 300 epochs. Now, because w
## we have to tell the trainer add stuff to the log files every 2 steps (or epoch, since we h
## because the default, updating the log files every 50 steps, will result in a terrible look
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

18 Hyperparameter Tuning

18.1 Structure of the Hyperparameter Tuning Chapters

The first part is structured as follows:

The concept of the hyperparameter tuning is described in Section 18.2.

Hyperparameter tuning with sklearn in Python is described in Chapter 19.

Hyperparameter tuning with river in Python is described in Chapter 21.

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 **?@sec-hpt-pytorch**. This is considered as the most effective, efficient, and flexible way to integrate `spotpython` into the `PyTorch` training workflow.

Figure 18.1 shows the graphical user interface of `spotpython` that is used in this book.

Figure 18.1: spot GUI

18.2 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

19 HPT: sklearn

19.1 Introduction to sklearn

20 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.

20.1 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"
```

20.2 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 gen_design_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_2024_10_12_23_48_41
```

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.

20.3 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})
```

20.4 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,
     )
```

20.5 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
```

20.6 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.

20.6.1 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}
```

20.6.2 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}
```

20.6.3 Optimizers

Optimizers are described in Section 4.2.

20.7 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, lo
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`.

20.7.1 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,
               })
```

20.8 Step 8: Calling the SPOT Function

20.8.1 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)
```

20.8.2 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.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 [----------] 4.96%
spotpython tuning: 6.436366676628063 [#---------] 7.38%
spotpython tuning: 6.436366676628063 [#---------] 10.33%
spotpython tuning: 6.436366676628063 [#---------] 13.77%
spotpython tuning: 6.436366676628063 [##--------] 16.77%
spotpython tuning: 6.436366676628063 [##--------] 19.19%
spotpython tuning: 6.436366676628063 [##--------] 22.55%
spotpython tuning: 6.436366676628063 [###-------] 27.13%
spotpython tuning: 6.436366676628063 [###-------] 32.46%
spotpython tuning: 6.436366676628063 [####------] 37.52%
spotpython tuning: 6.436366676628063 [#####-----] 45.97%
spotpython tuning: 6.436366676628063 [######----] 60.60%
spotpython tuning: 6.436366676628063 [######----] 64.85%
spotpython tuning: 6.436366676628063 [#######---] 73.06%
spotpython tuning: 6.436366676628063 [########--] 75.68%
spotpython tuning: 6.436366676628063 [########--] 82.55%
spotpython tuning: 6.436366676628063 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x33e46a240>
```

20.8.3 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 20.1: TensorBoard visualization of the spotpython optimization process and the surrogate model.

20.9 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+"_progress.pdf")
```

Results can also be printed in tabular form.

```
print(gen_design_table(fun_control=fun_control, spot=spot_tuner))
```

```
| name        | type   | default   |   lower |   upper | tuned                  | transform
|-------------|--------|-----------|---------|---------|------------------------|-------------
| 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+"_impor
```

20.10 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)
```

20.11 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)
```

20.11.1 Plot: Compare Predictions

```
from spotpython.plot.validation import plot_roc
plot_roc(model_list=[model_default, model_spot], fun_control= fun_control, model_names=["Defa
```

```
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)
```

```
(6.436366676628063, 10.813096016735146)
```

20.11.2 Detailed Hyperparameter Plots

```
spot_tuner.plot_important_hyperparameter_contour(filename=None)
```

```
C:  43.21937437869265
kernel:  100.0
gamma:  0.028895927097448246
shrinking:  24.81890091438799
tol:  0.06432027777032988
cache_size:  0.029077677771141282
```

```
break_ties:  0.03796473642110418
impo: [['C', 43.21937437869265], ['kernel', 100.0], ['gamma', 0.028895927097448246], ['shrinl
indices: [1, 0, 3, 4, 6, 5, 2]
indices after max_imp selection: [1, 0, 3, 4, 6, 5, 2]
```

x5: cache_size
x1: kernel
x2: gamma
x1: kernel
x1: kernel
x0: C
x3: shrinking
x0: C

x4: tol
x0: C
x6: break_ties
x5: cache_size
x2: gamma

x4: tol
x3: shrinking
8.48608
8.48596
8.48584
8.48572
8.48560
8.48548
8.48536

x3: shrinking
x4: tol

x6: break_ties
x3: shrinking
8.48608
8.48596
8.48584
8.48572
8.48560
8.48548
8.48536

x3: shrinking
x6: break_ties

x5: cache_size
x3: shrinking
8.54
8.50
8.46
8.42
8.38
8.34

x3: shrinking
x5: cache_size

1e−6+8.4854
x6: break_ties
x4: tol
15.2
13.6
12.0
10.4
8.8
7.2

x4: tol
x6: break_ties

x5: cache_size
x4: tol
x6: break_ties
x5: cache_size
x3: shrinking
x2: gamma
x4: tol
x2: gamma
1e−6+8.4854
x2: gamma
x4: tol

20.11.3 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
```

20.11.4 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)
```

Part VII

Hyperparameter Tuning with River

21 HPT: River

21.1 Introduction to River

22 Simplifying Hyperparameter Tuning in Online Machine Learning—The spotRiverGUI

22.1 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 22.2 describes how to install the software. It also explains how the `spotRiverGUI` can be started. Section 22.3 describes the binary classification task and the options available in the `spotRiverGUI`. Section 22.4 provides information about the planned regression task. Section 22.5 describes how the data can be visualized in the `spotRiverGUI`. Section 22.6 provides information about saving and loading experiments. Section 22.7 describes how to start an experiment and how the associated tensorboard process can be started and stopped. Section 22.8 provides information about the analysis of the results from the hyperparameter tuning process. Section 22.9 concludes the article and provides an outlook.

22.2 Installation and Starting

22.2.1 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
```

22.2.2 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 22.1.

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.

22.3 Binary Classification

22.3.1 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 22.1: spotriver GUI

22.3.1.1 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/

22.3.1.2 User Data Sets

Besides the `river` data sets described in Section 22.3.1.1, 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.

22.3.1.3 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).

22.3.1.4 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.

22.3.2 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.

22.3.3 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.

22.3.4 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/l 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 22.2 shows the `spotRiverGUI` when the `forest.AMFClassifier` is selected and Figure 22.3 shows the `spotRiverGUI` when the `tree.HoeffdingTreeClassifier` is selected.

Numerical and categorical hyperparameters are treated differently in the `spotRiverGUI`:

- 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 22.3. 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

Figure 22.2: `spotRiverGUI` when `forest.AMFClassifier` is selected

Figure 22.3: `spotRiverGUI` when `tree.HoeffdingAdaptiveTreeClassifier` is selected

`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.

22.4 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.

22.5 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 22.4) shows histograms of the target variables in the train and test data sets. The second figure (Figure 22.5) shows scatter plots of the features in the train data set. The third figure (Figure 22.6) shows the corresponding scatter plots of the features in the test data set.

> **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 22.7, Figure 22.8, and Figure 22.9: 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.

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:

Figure 22.4: Output from the `spotRiverGUI` when `Bananas` data is selected for the `Show Data` option

Figure 22.5: Visualization of the train data. Output from the `spotRiverGUI` when `Bananas` data is selected for the `Show Data` option

Figure 22.6: Visualization of the test data. Output from the `spotRiverGUI` when `Bananas` data is selected for the `Show Data` option

Figure 22.7: Output from the `spotRiverGUI` when `HTTP` data is selected for the `Show Data` option. The target variable is biased.

Figure 22.8: 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 22.9: 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.

```
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
```

22.6 Saving and Loading

22.6.1 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 22.1.

Table 22.1: 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

22.6.2 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.

22.7 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.

22.7.1 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 22.10 and Figure 22.11 show the tensorboard page when the `spotRiverGUI` is performing the tuning process.

Figure 22.10: Tensorboard visualization of the hyperparameter tuning process

Figure 22.11: Tensorboard. Parallel coordinates plot

`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.

22.8 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 22.12 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 in Figure 22.13. The default versus tuned hyperparameters are shown in Figure 22.14. The surrogate plot is shown in Figure 22.15, Figure 22.16, and Figure 22.17.

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| .   |
```

Figure 22.12: Progress plot of the hyperparameter tuning process

Figure 22.13: Confusion matrix

Figure 22.14: Default versus tuned hyperparameters

Figure 22.15: Surrogate plot based on the Kriging model. `x0` and `x1` plotted against each other.

Figure 22.16: Surrogate plot based on the Kriging model. `x1` and `x2` plotted against each other.

Figure 22.17: Surrogate plot based an the Kriging model. `x0` and `x2` plotted against each other.

```
|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 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.

22.9 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.

22.10 Appendix

22.10.1 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.

23 `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.

23.1 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.

23.2 Setup

23.2.1 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`.

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
```

23.2.2 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 23.1. Therefore, the `target_column` is set to `y` (as above).

Summary: Data Setup

The following parameters are used to specify the data setup:

```
prep_model_name = "StandardScaler"
test = test
train = train
target_column = "y"
```

23.2.3 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"
> ```

23.2.4 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
> ```

23.2.5 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
> ```

23.2.6 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 23.2.4.

Summary: Objective Function Setup

The following parameters are used:

```
fun = HyperRiver().fun_oml_horizon
```

23.2.7 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
```

23.2.8 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_control_in

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()
```

23.2.9 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.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: 3.197687314293912 [----------] 1.50%
spotpython tuning: 2.237682473123044 [#---------] 8.27%
spotpython tuning: 2.237682473123044 [#---------] 11.45%
spotpython tuning: 2.237682473123044 [##--------] 18.07%
spotpython tuning: 2.237682473123044 [##--------] 22.26%
spotpython tuning: 2.237682473123044 [##--------] 23.81%
spotpython tuning: 2.1977049143222906 [###-------] 27.07%
spotpython tuning: 2.196826338992795 [###-------] 32.55%
spotpython tuning: 2.1953262054488545 [####------] 38.89%
spotpython tuning: 2.1953262054488545 [#####-----] 47.81%
spotpython tuning: 2.181207274570926 [#######---] 73.83%
spotpython tuning: 2.181207274570926 [##########] 100.00% Done...
```

23.3 Using the `spotgui`

The `spotgui` [github] provides a convenient way to interact with the hyperparameter tuning process. To obtain the settings from Section 23.2.8, the `spotgui` can be started as shown in Figure 26.1.

Figure 23.1: spotgui

23.4 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 gen_design_table
print(gen_design_table(fun_control=fun_control, spot=spot_tuner))
```

```
| name                    | type   | default           |   lower |   upper | tuned
|-------------------------|--------|-------------------|---------|---------|-------------------
| grace_period            | int    | 200               |    10.0 |  1000.0 | 458.0
| max_depth               | int    | 20                |     2.0 |    20.0 | 2.0
| delta                   | float  | 1e-07             |   1e-08 |   1e-06 | 1e-08
| tau                     | float  | 0.05              |    0.01 |     0.1 | 0.1
| leaf_prediction         | factor | mean              |     0.0 |     2.0 | adaptive
| leaf_model              | factor | LinearRegression  |     0.0 |     2.0 | LinearRegression
| model_selector_decay    | float  | 0.95              |     0.9 |    0.99 | 0.99
| splitter                | factor | EBSTSplitter      |     0.0 |     2.0 | TEBSTSplitter
| 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 | 503.2323556340199
| memory_estimate_period  | int    | 6                 |     3.0 |     8.0 | 5.0
| stop_mem_management     | factor | 0                 |     0.0 |     1.0 | 1
| remove_poor_attrs       | factor | 0                 |     0.0 |     1.0 | 1
| merit_preprune          | factor | 1                 |     0.0 |     1.0 | 1
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=10.0)
```

23.5 Performance of the Model with Default Hyperparameters

23.5.1 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)
```

23.5.2 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_horizon_
df_labels=["default"]
plot_bml_oml_horizon_metrics(df_eval = [df_eval_default], log_y=False, df_labels=df_labels, r
```

23.5.3 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_colu
```

23.6 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, df_label
```

```
plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b], df_true_spot[a:b]], target_
```

```
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["Predi
plot_actual_vs_predicted(y_test=df_true_spot[target_column], y_pred=df_true_spot["Prediction"
```

23.7 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}
```

23.7.1 Spot Model

```python
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.

```python
# model_spot.draw()
```

```python
model_spot.summary
```

```
{'n_nodes': 13,
 'n_branches': 6,
 'n_leaves': 7,
 'n_active_leaves': 5,
 'n_inactive_leaves': 2,
 'height': 5,
 'total_observed_weight': 14168.0}
```

```python
from spotpython.utils.eda import compare_two_tree_models
print(compare_two_tree_models(model_default, model_spot))
```

```
| Parameter              |   Default |   Spot |
|------------------------|-----------|--------|
| n_nodes                |        23 |     13 |
| n_branches             |        11 |      6 |
| n_leaves               |        12 |      7 |
| n_active_leaves        |        12 |      5 |
| n_inactive_leaves      |         0 |      2 |
| height                 |         7 |      5 |
| total_observed_weight  |     14168 |  14168 |
```

23.8 Detailed Hyperparameter Plots

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
grace_period:  0.0013940676950504687
max_depth:  0.0013940676950504687
delta:  0.47687246213064
tau:  0.0032477549704306515
leaf_prediction:  58.73267939694902
leaf_model:  0.005381256429946626
model_selector_decay:  0.07158922462880041
splitter:  0.39166776051090396
min_samples_split:  0.0013940676950504687
binary_split:  0.01768249036012152
max_size:  0.0013940676950504687
memory_estimate_period:  0.0013940676950504687
stop_mem_management:  1.6313617746091733
remove_poor_attrs:  0.03673582413530393
merit_preprune:  100.0
impo: [['grace_period', 0.0013940676950504687], ['max_depth', 0.0013940676950504687], ['delta
indices: [14, 4, 12, 2, 7, 6, 13, 9, 5, 3, 0, 1, 8, 10, 11]
indices after max_imp selection: [14, 4, 12]
```

23.9 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
```

24 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 23.1.

```
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]
```

24.1 Setup

We will use a general experiment, data, evaluation, river-specific, objective-function, and surrogate setup similar to the setup from Section 23.2. Only the model setup differs from the setup in Section 23.2. 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
```

24.1.1 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_control_i
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()
```

24.2 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_hyperparamet` 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.

```
from spotpython.utils.eda import gen_design_table
print(gen_design_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_hyperp
set_int_hyperparameter_values(fun_control, "n_estimators", 2, 7)
set_float_hyperparameter_values(fun_control, "step", 0.1, 15)
print(gen_design_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.

24.2.1 Run the `Spot` Optimizer

```
from spotpython.spot import spot
spot_tuner = spot.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: 2.8226382496098203 [########--] 80.82%
spotpython tuning: 2.8226382496098203 [##########] 100.00% Done...
```

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/
```

24.3 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 gen_design_table
print(gen_design_table(fun_control=fun_control, spot=spot_tuner))
```

```
| name            | type    |   default |   lower |   upper |   tuned | transform
|-----------------|---------|-----------|---------|---------|---------|----------------------
| n_estimators    | int     |       3.0 |     2.0 |       7 |     5.0 | transform_power_2_int
| step            | float   |       1.0 |     0.1 |      15 |    15.0 | None
| use_aggregation | factor  |       1.0 |     0.0 |       1 |     1.0 | None
```

A histogram can be used to visualize the most important hyperparameters.

```
spot_tuner.plot_importance(threshold=10.0)
```

24.4 Performance of the Model with Default Hyperparameters

24.4.1 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)
```

24.4.2 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_
df_labels=["default"]
plot_bml_oml_horizon_metrics(df_eval = [df_eval_default], log_y=False, df_labels=df_labels, r
```

24.4.3 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=target_colu
```

24.5 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_label
```

```
plot_bml_oml_horizon_predictions(df_true = [df_true_default[a:b], df_true_spot[a:b]], target_
```

```
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["Predi
plot_actual_vs_predicted(y_test=df_true_spot[target_column], y_pred=df_true_spot["Prediction"
```

24.6 Detailed Hyperparameter Plots

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
n_estimators:  100.00000000000001
step:  0.004303362866512184
use_aggregation:  0.06163625943695855
impo: [['n_estimators', 100.00000000000001], ['step', 0.004303362866512184], ['use_aggregatio
indices: [0, 2, 1]
indices after max_imp selection: [0, 2, 1]
```

24.7 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

25 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.

25.1 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.

25.2 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,
    )
```

25.3 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
```

25.3.1 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)
```

```python
# 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.])
```

25.3.2 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.

```python
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
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}]
```

25.3.3 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
```

25.4 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.

25.4.1 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.

25.4.2 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.

25.4.2.1 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.

25.4.2.2 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], gene
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
```

25.4.2.3 Stage "test"

```python
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=generator_
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, -0.0214,
         -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.])
```

25.4.2.4 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.])
```

25.4.3 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 25.4.2.2.

```
def train_dataloader(self) -> DataLoader:
    return DataLoader(self.data_train, batch_size=self.batch_size, num_workers=self.num_worke
```

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([[ 0.0562, -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.])
```

25.4.4 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 25.4.2.2.

```
def val_dataloader(self) -> DataLoader:
    return DataLoader(self.data_val, batch_size=self.batch_size, num_workers=self.num_worker
```

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.])
```

25.4.5 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 25.4.2.3.

```
def test_dataloader(self) -> DataLoader:
    return DataLoader(self.data_test, batch_size=self.batch_size, num_workers=self.num_worke
```

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, -0.0214,
         -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.])
```

25.4.6 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 25.4.2.4.

> ⚠ Warning
>
> The `batch_size` is set to the length of the `data_predict` set.

```python
def predict_dataloader(self) -> DataLoader:
    return DataLoader(self.data_predict, batch_size=len(self.data_predict), num_workers=self
```

The `predict_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_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.])
```

25.5 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=Fal
    ],
    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"]
```

25.6 Further Information

25.6.1 Preprocessing

Preprocessing is handled by `Lightning` and `PyTorch`. It is described in the LIGHTNING-DATAMODULE documentation. Here you can find information about the `transforms` methods.

26 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.

```
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_control_i
from spotpython.utils.eda import gen_design_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
```

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", [3,5])
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(gen_design_table(fun_control))
```

Saving the experiment is optional. If `save_experiment` is set to `True`, the experiment is saved to a file.

```
fun_control.update({"save_experiment": True})
```

Finally, a `Spot` object is created. Calling the method `run()` starts the hyperparameter tuning process.

```
spot_tuner = spot.Spot(fun=fun,fun_control=fun_control, design_control=design_control)
spot_tuner.save_experiment(path="userExperiment", overwrite=False)
res = spot_tuner.run()
```

26.1 Looking at the Results

26.1.1 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()
```

26.1.2 Tuned Hyperparameters and Their Importance

Results can be printed in tabular form.

```
from spotpython.utils.eda import gen_design_table
print(gen_design_table(fun_control=fun_control, spot=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)
```

26.1.3 Get the Tuned Architecture

```
import pprint
from spotpython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(spot_tuner, fun_control)
pprint.pprint(config)
```

- Test on the full data set

```
# set the value of the key "TENSORBOARD_CLEAN" to True in the fun_control dictionary and use
fun_control.update({"TENSORBOARD_CLEAN": True})
fun_control.update({"tensorboard_log": True})
```

```
import os
import pickle

def chk_pickle(s):
    # Construct the filename
    filename = s + '.pickle'

    # Check if the file exists
    return os.path.isfile(filename)
```

```
from spotpython.light.testmodel import test_model
from spotpython.light.loadmodel import load_light_from_checkpoint
from spotpython.utils.init import get_feature_names


if chk_pickle("model_loaded_" + PREFIX):
    with open("model_loaded_" + PREFIX + ".pickle", "rb") as f:
        model_loaded = pickle.load(f)
else:
    test_model(config, fun_control)
    model_loaded = load_light_from_checkpoint(config, fun_control)
    # save model_loaded to pickle file
    with open("model_loaded_" + PREFIX + ".pickle", "wb") as f:
        pickle.dump(model_loaded, f)

get_feature_names(fun_control)
```

26.2 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
```

```
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)
```

26.3 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.

26.3.1 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`.

26.3.2 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`.

26.3.3 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`.

26.3.4 Core-Model Setup

By using `core_model_name = "light.regression.NNLinearRegressor"`, the `spotpython` model class `NetLightRegression` [SOURCE] from the `light.regression` module is selected.

26.3.5 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.

26.3.6 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.

26.4 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.

26.5 Loading the Saved Experiment and Getting the Hyperparameters of the Tuned Model

To get the tuned hyperparameters as a dictionary, the `get_experiment_from_PREFIX` function can be used.

```
from spotpython.utils.file import get_experiment_from_PREFIX
config = get_experiment_from_PREFIX("601")["config"]
config
```

26.6 Using the `spotgui`

The `spotgui` [github] provides a convenient way to interact with the hyperparameter tuning process. To obtain the settings from **?@sec-summary-setting-up-the-experiment-601**, the `spotgui` can be started as shown in Figure 26.1.

26.7 Summary

This section presented an introduction to the basic setup of hyperparameter tuning with `spotpython` and `PyTorch` Lightning.

Figure 26.1: spotgui

27 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`.

27.1 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 **?@sec-basic-setup-601**, 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.

```
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_i
from spotpython.utils.eda import gen_design_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.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(gen_design_table(fun_control=fun_control, spot=spot_tuner))
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}

train_model result: {'val_loss': 12256.3876953125, 'hp_metric': 12256.3876953125}

train_model result: {'val_loss': 23173.634765625, 'hp_metric': 23173.634765625}

train_model result: {'val_loss': nan, 'hp_metric': nan}

train_model result: {'val_loss': nan, 'hp_metric': nan}

train_model result: {'val_loss': 23810.33203125, 'hp_metric': 23810.33203125}

train_model result: {'val_loss': 13219.3115234375, 'hp_metric': 13219.3115234375}
spotpython tuning: 12256.3876953125 [###-------] 33.33%

train_model result: {'val_loss': 10988.732421875, 'hp_metric': 10988.732421875}
spotpython tuning: 10988.732421875 [####------] 41.67%

train_model result: {'val_loss': 10807.1953125, 'hp_metric': 10807.1953125}
spotpython tuning: 10807.1953125 [#####-----] 50.00%
| name           | type   | default   |   lower |   upper | tuned               | transform
|----------------|--------|-----------|---------|---------|---------------------|----------
| l1             | int    | 3         |     3.0 |     8.0 | 4.0                 | transform_
| epochs         | int    | 4         |     4.0 |     9.0 | 7.0                 | transform_
| batch_size     | int    | 4         |     1.0 |     4.0 | 2.0                 | transform_
| act_fn         | factor | ReLU      |     0.0 |     5.0 | Swish               | None
```

```
| optimizer      | factor | SGD       |     0.0 |    11.0 | Adagrad              | None
| dropout_prob   | float  | 0.01      |     0.0 |    0.25 | 0.08638404150780843 | None
| lr_mult        | float  | 1.0       |     0.1 |    10.0 | 5.9782734862852775  | None
| patience       | int    | 2         |     2.0 |     6.0 | 3.0                 | transform_p
| batch_norm     | factor | 0         |     0.0 |     1.0 | 1                   | None
| initialization | factor | Default   |     0.0 |     0.0 | Default             | None
l1:  0.3405546044830082
epochs:  100.0
batch_size:  0.3403411651664836
act_fn:  0.015686587527038564
optimizer:  83.45110370294486
dropout_prob:  0.21040471019463378
lr_mult:  3.5368197185142463
patience:  0.028632058307146952
batch_norm:  0.013408853458270809
impo: [['l1', 0.3405546044830082], ['epochs', 100.0], ['batch_size', 0.3403411651664836], ['a
indices: [1, 4, 6, 0, 2, 5, 7, 3, 8]
indices after max_imp selection: [1, 4, 6]
```

27.2 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.

28 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`.

28.1 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 28.2.4
- `my_hyperdict.json`, see Section 28.2.3
- `my_hyperdict.py`, see Section 28.2.2.

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 gen_design_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.Spot(fun=fun,fun_control=fun_control, design_control=design_control)
```

```
res = spot_tuner.run()
print(gen_design_table(fun_control=fun_control, spot=spot_tuner))
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

```
train_model result: {'val_loss': nan, 'hp_metric': nan}

train_model result: {'val_loss': 4795.224609375, 'hp_metric': 4795.224609375}

train_model result: {'val_loss': 19881.609375, 'hp_metric': 19881.609375}

train_model result: {'val_loss': nan, 'hp_metric': nan}

train_model result: {'val_loss': 22673.50390625, 'hp_metric': 22673.50390625}

train_model result: {'val_loss': 23567.240234375, 'hp_metric': 23567.240234375}

train_model result: {'val_loss': 4667.7060546875, 'hp_metric': 4667.7060546875}
spotpython tuning: 4667.7060546875 [#---------] 10.85%

train_model result: {'val_loss': 8564.896484375, 'hp_metric': 8564.896484375}
spotpython tuning: 4667.7060546875 [#######---] 72.10%
```

```
train_model result: {'val_loss': 4786.97119140625, 'hp_metric': 4786.97119140625}
spotpython tuning: 4667.7060546875 [########--] 82.56%

train_model result: {'val_loss': 4959.9033203125, 'hp_metric': 4959.9033203125}
spotpython tuning: 4667.7060546875 [##########] 100.00% Done...

| name           | type   | default   |   lower |   upper | tuned               | transform
|----------------|--------|-----------|---------|---------|---------------------|------------
| l1             | int    | 3         |     3.0 |     8.0 | 4.0                 | transform_p
| epochs         | int    | 4         |     4.0 |     9.0 | 7.0                 | transform_p
| batch_size     | int    | 4         |     1.0 |     4.0 | 2.0                 | transform_p
| act_fn         | factor | ReLU      |     0.0 |     5.0 | Swish               | None
| optimizer      | factor | SGD       |     0.0 |    11.0 | Adagrad             | None
| dropout_prob   | float  | 0.01      |     0.0 |    0.25 | 0.0859555065832669  | None
| lr_mult        | float  | 1.0       |     0.1 |    10.0 | 5.442942114443714   | None
| patience       | int    | 2         |     2.0 |     6.0 | 3.0                 | transform_p
| initialization | factor | Default   |     0.0 |     2.0 | Kaiming             | None
l1:  0.7704984104186962
epochs:  0.7704984104186962
batch_size:  92.24549821912215
act_fn:  24.93612753081475
optimizer:  0.7704984104186962
dropout_prob:  0.7704984104186962
lr_mult:  100.0
patience:  0.7704984104186962
initialization:  0.7704984104186962
impo: [['l1', 0.7704984104186962], ['epochs', 0.7704984104186962], ['batch_size', 92.2454982
indices: [6, 2, 3, 0, 1, 4, 5, 7, 8]
indices after max_imp selection: [6, 2, 3]
```

28.2 Details

28.2.1 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.

28.2.2 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 28.2.2. 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"`).

28.2.3 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
        }
    }
}
```

28.2.4 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.

28.2.4.1 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,
```

```
    ):
        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 = 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 them in sequence
        self.layers = nn.Sequential(*layers)
```

28.2.4.2 The `_get_hidden_sizes` Method

`__init__()` uses the helper method `_get_hidden_sizes()` to calculate the hidden layer sizes based on the number of neurons in the first hidden layer `l1`. The hidden layer sizes are calculated as follows:

```
def _get_hidden_sizes(self):
    # Calculate the hidden layer sizes based on the number of neurons in the first hidden lay
```

```
    hidden_sizes = [self.hparams.l1]
    while hidden_sizes[-1] > 2:
        hidden_sizes.append(hidden_sizes[-1] // 2)
    return hidden_sizes
```

28.2.4.3 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)
```

28.2.4.4 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
```

28.2.4.5 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
```

28.2.4.6 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
```

28.2.4.7 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
```

28.2.4.8 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)
```

28.2.4.9 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.hparams
    )
    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.

28.3 Connection with the LightDataModule

The steps described in Section 28.2.4 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.

28.3.1 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.

28.3.2 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 28.3.2.1. 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.

28.3.2.1 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
```

28.3.2.2 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))]).s
        # 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(targe
```

```
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 28.3.3 and Section 28.3.4, respectively.

28.3.2.3 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_
    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, -0.0764,
```

```
         -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.])
```

28.3.2.4 The "setup" Method: Stage "predict"

Prediction and testing use the same data set. The prediction data set is created based on the `test_size`.

```python
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.])
```

28.3.3 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 28.3.2.2.

```
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:

```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"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([[ 0.0562, -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.])
```

28.3.4 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 28.3.2.2.

```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, -0.0011,
          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.])
```

28.3.5 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 25.4.2.3.

```
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:

```
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.])
```

28.3.6 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 25.4.2.4.

> **Warning**
>
> The `batch_size` is set to the length of the `data_predict` set.

```python
def predict_dataloader(self) -> DataLoader:
    return DataLoader(data_predict, batch_size=len(data_predict), num_workers=num_workers)
```

> **Using the `predict_dataloader()` Method**
>
> The `predict_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_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.])
```

28.4 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", strict=Fal
    ],
    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"]
```

28.5 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
```

28.6 Further Information

28.6.1 Preprocessing

Preprocessing is handled by `Lightning` and `PyTorch`. It is described in the LIGHTNING-DATAMODULE documentation. Here you can find information about the `transforms` methods.

29 Hyperparameter Tuning with PyTorch Lightning: Physics Informed Neural Networks

In this section, we will show how to set up PINN hyperparameter tuner from scratch based on the `spotpython` programs from Chapter 28.

29.1 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')
```

29.2 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 29.2.2
- `my_hyperdict.json` ⇒ `pinn_hyperdict.py`, see Section 29.2.3
- `my_hyperdict.py` ⇒ `pinn_hyperdict.py`, see **?@sec-pinn-hyper-dict**.

29.2.1 The New `pinn_hyperdict.py` File

Modifying the `pin_hyperdict.py` file is very easy. We simply have to change the classname `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
```

29.2.2 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
```

```python
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 create the
            _L_out (int):
                The number of output classes. Not a hyperparameter, but needed to create the
            _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 them in s
        self.layers = nn.Sequential(*layers)
```

```python
    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 // current.
        No more than 4 repeats of the same value (`max_repeats` below) are added to the list

        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=True)
        return val_loss

    def validation_step(self, batch: tuple, batch_idx: int, prog_bar: bool = False) -> torch
        """
        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 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, on_step=False, on_epoch=True, prog_bar=prog_bar)
        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.Tenso
        """
        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.Ten
        """
        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
        # If you want to manually log, please consider using `self.log_dict({'pred_loss': pre
        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

        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=self.hpa
        )
        return optimizer
```

29.2.3 The New `pinn_hyperdict.json` File

30 Explainable AI with SpotPython and Pytorch

```
from torch.utils.data import DataLoader
from spotpython.utils.init import fun_control_init
from spotpython.hyperparameters.values import set_control_key_value
from spotpython.data.diabetes import Diabetes
from spotpython.light.regression.netlightregression import NetLightRegression
from spotpython.hyperdict.light_hyper_dict import LightHyperDict
from spotpython.hyperparameters.values import add_core_model_to_fun_control
from spotpython.hyperparameters.values import (
        get_default_hyperparameters_as_array, get_one_config_from_X)
from spotpython.hyperparameters.values import set_control_key_value
from spotpython.plot.xai import (get_activations, get_gradients, get_weights, plot_nn_values_
fun_control = fun_control_init(
    _L_in=10, # 10: diabetes
    _L_out=1,
    _torchmetric="mean_squared_error",
    )
dataset = Diabetes()
set_control_key_value(control_dict=fun_control,
                        key="data_set",
                        value=dataset,
                        replace=True)
add_core_model_to_fun_control(fun_control=fun_control,
                              core_model=NetLightRegression,
                              hyper_dict=LightHyperDict)
X = get_default_hyperparameters_as_array(fun_control)
config = get_one_config_from_X(X, fun_control)
_L_in = fun_control["_L_in"]
_L_out = fun_control["_L_out"]
_torchmetric = fun_control["_torchmetric"]
model = fun_control["core_model"](**config, _L_in=_L_in, _L_out=_L_out, _torchmetric=_torchme
batch_size= config["batch_size"]
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=False)
```

```python
get_activations(model, fun_control=fun_control, batch_size=batch_size, device = "cpu")
```

```
net: NetLightRegression(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=8, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.01, inplace=False)
    (3): Linear(in_features=8, out_features=4, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.01, inplace=False)
    (6): Linear(in_features=4, out_features=4, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.01, inplace=False)
    (9): Linear(in_features=4, out_features=2, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.01, inplace=False)
    (12): Linear(in_features=2, out_features=1, bias=True)
  )
)
```

```
{0: array([ 1.43207282e-01,  6.29712082e-03,  1.04200497e-01, -3.79188173e-03,
        -1.74976081e-01, -7.97475874e-02, -2.00860098e-01,  2.48444736e-01,
         1.42530382e-01, -2.86847632e-03,  3.61538231e-02, -5.21567538e-02,
        -2.15294853e-01, -1.26742452e-01, -1.79230243e-01,  2.73077697e-01,
         1.36738747e-01,  8.57900176e-03,  1.01677164e-01,  3.27536091e-03,
        -1.92429125e-01, -7.95854479e-02, -1.84092522e-01,  2.72164375e-01,
         1.51459932e-01,  3.70034538e-02,  4.94864434e-02, -6.36564642e-02,
        -1.63678646e-01, -1.26617596e-01, -2.05547154e-01,  2.25242063e-01,
         1.54910132e-01,  4.92912624e-03,  6.90693632e-02, -3.28048877e-02,
        -1.77523270e-01, -1.17699921e-01, -1.95609123e-01,  2.50784487e-01,
         1.66618377e-01,  1.22015951e-02,  2.58807316e-02, -8.16192776e-02,
        -2.00623482e-01, -1.17052853e-01, -1.86843857e-01,  2.40996510e-01,
         1.80479109e-01,  3.72159854e-02,  3.55244167e-02, -3.60636115e-02,
        -2.09616780e-01, -1.19843856e-01, -1.44335642e-01,  2.73970902e-01,
         1.46006003e-01, -1.83095373e-02,  8.83664042e-02,  2.28608586e-02,
        -1.77115664e-01, -1.37761638e-01, -1.90622538e-01,  2.85049856e-01,
         1.44436464e-01,  1.36893094e-02,  6.65568933e-02, -2.01083720e-04,
        -1.99043870e-01, -1.11171007e-01, -1.76820531e-01,  2.78549373e-01,
         1.31597325e-01,  1.31126186e-02,  5.92438355e-02, -6.50760308e-02,
        -1.55642599e-01, -1.12090096e-01, -2.32182071e-01,  2.25448400e-01,
         2.09733546e-01,  4.48576249e-02,  1.76887661e-02, -7.26176351e-02,
```

```
       -1.81560591e-01, -1.18118793e-01, -1.55840069e-01,  2.45131850e-01,
        1.57539800e-01,  4.57477495e-02,  8.64019692e-02,  1.06538832e-02,
       -2.25713193e-01, -8.36062431e-02, -1.51326194e-01,  2.42097050e-01,
        1.46130219e-01, -6.08363096e-03,  4.69235368e-02, -4.06553932e-02,
       -1.90215483e-01, -1.30105391e-01, -1.91207454e-01,  2.75829703e-01,
        1.37035578e-01,  1.32784406e-02,  8.11730623e-02, -2.83420049e-02,
       -1.72134370e-01, -1.05717532e-01, -1.93411276e-01,  2.68321246e-01,
        1.24822736e-01, -2.49985531e-02,  5.46513572e-02, -3.76938097e-02,
       -2.02080101e-01, -1.29510283e-01, -1.99880868e-01,  2.84415126e-01,
        1.36025175e-01,  2.10405551e-02,  1.25923336e-01, -1.76883545e-02,
       -1.46617338e-01, -1.00234658e-01, -2.21794963e-01,  2.05139250e-01],
      dtype=float32),
3: array([-0.09106569,  0.15831017,  0.29874575, -0.05709065, -0.07168067,
        0.13238071,  0.29310873, -0.04537551, -0.08868651,  0.15093939,
        0.29576218, -0.0508837 , -0.07256822,  0.15756649,  0.29804155,
       -0.06024086, -0.07925774,  0.15159754,  0.29655144, -0.05204485,
       -0.06510481,  0.14707124,  0.2955585 , -0.05045141, -0.05945833,
        0.15397519,  0.28643152, -0.03937227, -0.0780265 ,  0.1443048 ,
        0.2993904 , -0.04338943, -0.07745007,  0.1438258 ,  0.29152495,
       -0.04569358, -0.08201659,  0.14775375,  0.3020632 , -0.06361471,
       -0.05014775,  0.16657498,  0.28808075, -0.04191205, -0.07614301,
        0.16806594,  0.29809946, -0.05615523, -0.07369395,  0.13612927,
        0.2925982 , -0.04455032, -0.08367015,  0.14735378,  0.29441217,
       -0.05101945, -0.07929114,  0.12925598,  0.29300398, -0.04631315,
       -0.09977546,  0.1741175 ,  0.30642375, -0.07330882], dtype=float32),
6: array([ 0.02894721, -0.15329668,  0.0478624 ,  0.5073338 ,  0.03414171,
       -0.1624101 ,  0.0582981 ,  0.5058923 ,  0.0301194 , -0.15560818,
        0.05099656,  0.5068564 ,  0.02897662, -0.15344843,  0.04822758,
        0.5072659 ,  0.03012998, -0.15550745,  0.05065323,  0.5069246 ,
        0.03103478, -0.1570965 ,  0.05247599,  0.50667256,  0.02730933,
       -0.15252227,  0.0509877 ,  0.5065358 ,  0.03256607, -0.15896471,
        0.05305116,  0.5067359 ,  0.03095146, -0.15756464,  0.05418621,
        0.5063291 ,  0.03229896, -0.1581411 ,  0.05142961,  0.50702184,
        0.02454497, -0.14787357,  0.04604906,  0.50718296,  0.02638294,
       -0.14930864,  0.04427201,  0.5077407 ,  0.03309863, -0.16082475,
        0.05695035,  0.50603575,  0.03071198, -0.15675312,  0.05250891,
        0.5066291 ,  0.03489432, -0.16362445,  0.05948593,  0.50574666,
        0.02671532, -0.14859803,  0.04098557,  0.5084204 ], dtype=float32),
9: array([0.04397329, 0.23183572, 0.04112439, 0.22675759, 0.0430866 ,
       0.23046201, 0.04386139, 0.2317175 , 0.04319487, 0.23055825,
       0.04269706, 0.22967225, 0.04286424, 0.23156166, 0.04263979,
       0.2289063 , 0.0421553 , 0.2292049 , 0.04312573, 0.22948454,
       0.04418794, 0.23408437, 0.04489119, 0.23388621, 0.0414625 ,
```

```
        0.22755873, 0.0426609 , 0.22978865, 0.04081305, 0.22611658,
        0.04594607, 0.23471704], dtype=float32),
 12: array([-0.30635476, -0.30988604, -0.3073418 , -0.30644947, -0.30726004,
        -0.30787635, -0.306807  , -0.308307  , -0.3082779 , -0.3078606 ,
        -0.30507785, -0.3049926 , -0.30935943, -0.30782318, -0.3103186 ,
        -0.30425358], dtype=float32)}
```

```
get_gradients(model, fun_control=fun_control, batch_size=batch_size, device = "cpu")
```

```
{'layers.0.weight': array([ 0.10417589, -0.04161514,  0.10597268,  0.02180895,  0.12001497,
         0.0289035 ,  0.01146171,  0.08183315,  0.2495192 ,  0.5108763 ,
         0.14668097, -0.07902835,  0.00912531,  0.02640062,  0.14108549,
         0.06816658,  0.14256881, -0.00347908,  0.07373644,  0.23171763,
         0.08313344, -0.0332093 ,  0.08456729,  0.01740377,  0.09577318,
         0.0230653 ,  0.00914656,  0.0653037 ,  0.1991189 ,  0.4076846 ,
         0.04405227,  0.03805925,  0.015035  ,  0.0069457 ,  0.0094994 ,
         0.03021198, -0.01876849,  0.02160799, -0.03238906, -0.02050959,
         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.        ,
        -0.05415884,  0.02163483, -0.05509295, -0.01133801, -0.06239325,
        -0.01502632, -0.0059587 , -0.04254333, -0.12971975, -0.2655938 ],
       dtype=float32),
 'layers.3.weight': array([ 0.0000000e+00,  0.0000000e+00,  0.0000000e+00,  0.0000000e+00,
         0.0000000e+00,  0.0000000e+00,  0.0000000e+00,  0.0000000e+00,
        -5.8896484e+00, -6.3058013e-01, -2.5641673e+00, -8.9936234e-02,
         0.0000000e+00,  0.0000000e+00,  0.0000000e+00, -1.0009734e+01,
         5.1539743e-01,  5.5181440e-02,  2.2438775e-01,  7.8702327e-03,
         0.0000000e+00,  0.0000000e+00,  0.0000000e+00,  8.7594193e-01,
         0.0000000e+00,  0.0000000e+00,  0.0000000e+00,  0.0000000e+00,
         0.0000000e+00,  0.0000000e+00,  0.0000000e+00,  0.0000000e+00],
       dtype=float32),
 'layers.6.weight': array([ 0.        ,  7.6445217, 15.007772 ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ,  0.        , 11.027901 ,
        21.650045 ,  0.        ,  0.        ,  3.458755 ,  6.7902493,
         0.        ], dtype=float32),
 'layers.9.weight': array([ -2.3285942,   0.        ,  -3.9471323, -39.11015  ,  -4.6057286,
          0.        ,  -7.8070364, -77.35598  ], dtype=float32),
 'layers.12.weight': array([-12.126856, -64.91129 ], dtype=float32)}
```

```
get_weights(model)
```

```
{'Layer 0': array([-0.12895013,  0.01047491, -0.15705723,  0.11925378, -0.26944348,
         0.23180884, -0.22984707, -0.25141433, -0.19982024,  0.1432175 ,
        -0.11684369,  0.11833665, -0.2683918 , -0.19186287, -0.11611126,
        -0.06214499, -0.24123858,  0.20706302, -0.07457636,  0.10150522,
         0.22361842,  0.05891513,  0.08647271,  0.3052416 , -0.1426217 ,
         0.10016554, -0.14069483,  0.22599207,  0.25255734, -0.29155323,
         0.26994652,  0.1510033 ,  0.13780165,  0.13018303,  0.26287985,
        -0.04175457, -0.26743335, -0.09074122, -0.2227112 ,  0.02090477,
        -0.05904209, -0.16961981, -0.02875187,  0.2995954 , -0.0249426 ,
         0.01004026, -0.04931906,  0.04971322,  0.28176296,  0.19337103,
         0.11224869,  0.06871963,  0.07456426,  0.12216929, -0.04086405,
        -0.29390487, -0.19555901,  0.2699275 ,  0.01890202, -0.25616774,
         0.04987781,  0.26129004, -0.29883513, -0.21289697, -0.12594265,
         0.0126926 , -0.07375361, -0.03475064, -0.30828732,  0.14808287,
         0.2775668 ,  0.19329055, -0.22393112, -0.25491226,  0.13131432,
         0.00710202,  0.12963155, -0.3090024 , -0.01885445,  0.22301763],
       dtype=float32),
 'Layer 3': array([ 0.19455571,  0.12364562, -0.2711233 ,  0.2728095 ,  0.11085409,
         0.24458633, -0.13908438,  0.07495222,  0.34520328,  0.23782092,
         0.28354865, -0.07424083,  0.26936427, -0.2769144 ,  0.03057847,
        -0.19906998, -0.08245403, -0.09054411,  0.02645254,  0.32178298,
         0.17503859, -0.00149773,  0.2509683 , -0.1811804 ,  0.18221132,
        -0.03278595, -0.06152213,  0.0413917 , -0.27085608,  0.04085568,
         0.11887809,  0.302264  ], dtype=float32),
 'Layer 6': array([ 0.4752962 , -0.24824601,  0.22039747,  0.19587505,  0.13966405,
         0.39540154, -0.20208222,  0.13140953,  0.00280607, -0.3760708 ,
        -0.12140697, -0.33391154,  0.22107768,  0.04494798,  0.04898232,
        -0.15168536], dtype=float32),
 'Layer 9': array([ 0.07573527, -0.22145915, -0.30541402,  0.03821951, -0.3709231 ,
        -0.3758251 , -0.3254385 , -0.1698224 ], dtype=float32),
 'Layer 12': array([0.2738903, 0.5417278], dtype=float32)}
```

```
visualize_activations(model, fun_control=fun_control, batch_size=batch_size, device = "cpu",
```

```
net: NetLightRegression(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=8, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.01, inplace=False)
```

```
    (3): Linear(in_features=8, out_features=4, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.01, inplace=False)
    (6): Linear(in_features=4, out_features=4, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.01, inplace=False)
    (9): Linear(in_features=4, out_features=2, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.01, inplace=False)
    (12): Linear(in_features=2, out_features=1, bias=True)
  )
)
128 values in Layer 0.
16 padding values added.
144 values now in Layer 0.
```

```
64 values in Layer 3.
64 values now in Layer 3.
```

```
64 values in Layer 6.
64 values now in Layer 6.
```

```
32 values in Layer 9.
4 padding values added.
36 values now in Layer 9.
```

```
16 values in Layer 12.
16 values now in Layer 12.
```

```
visualize_weights_distributions(model, color=f"C{0}")
```

```
n:5
```

```
visualize_gradient_distributions(model, fun_control, batch_size=batch_size, color=f"C{0}")
```

```
n:5
```

```
visualize_weights(model, absolute=True, cmap="gray", figsize=(6, 6))
```

```
80 values in Layer Layer 0.
1 padding values added.
```

```
81 values now in Layer Layer 0.
```

```
32 values in Layer Layer 3.
4 padding values added.
36 values now in Layer Layer 3.
```

```
16 values in Layer Layer 6.
16 values now in Layer Layer 6.
```

```
8 values in Layer Layer 9.
1 padding values added.
9 values now in Layer Layer 9.
```

```
2 values in Layer Layer 12.
2 padding values added.
4 values now in Layer Layer 12.
```

```
visualize_gradients(model, fun_control, batch_size, absolute=True, cmap="BlueWhiteRed", figsi
```

```
80 values in Layer layers.0.weight.
1 padding values added.
81 values now in Layer layers.0.weight.
```

```
32 values in Layer layers.3.weight.
4 padding values added.
36 values now in Layer layers.3.weight.
```

```
16 values in Layer layers.6.weight.
16 values now in Layer layers.6.weight.
```

```
8 values in Layer layers.9.weight.
1 padding values added.
9 values now in Layer layers.9.weight.
```

```
2 values in Layer layers.12.weight.
2 padding values added.
4 values now in Layer layers.12.weight.
```

```
visualize_activations(model, fun_control=fun_control, batch_size=batch_size, device = "cpu")
```

```
net: NetLightRegression(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=8, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.01, inplace=False)
    (3): Linear(in_features=8, out_features=4, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.01, inplace=False)
    (6): Linear(in_features=4, out_features=4, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.01, inplace=False)
    (9): Linear(in_features=4, out_features=2, bias=True)
    (10): ReLU()
```

```
    (11): Dropout(p=0.01, inplace=False)
    (12): Linear(in_features=2, out_features=1, bias=True)
  )
)
128 values in Layer 0.
16 padding values added.
144 values now in Layer 0.
```

```
64 values in Layer 3.
64 values now in Layer 3.
```

```
64 values in Layer 6.
64 values now in Layer 6.
```

```
32 values in Layer 9.
4 padding values added.
36 values now in Layer 9.
```

```
16 values in Layer 12.
16 values now in Layer 12.
```

```
visualize_activations(model, fun_control=fun_control, batch_size=batch_size, device = "cpu")
```

```
net: NetLightRegression(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=8, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.01, inplace=False)
    (3): Linear(in_features=8, out_features=4, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.01, inplace=False)
    (6): Linear(in_features=4, out_features=4, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.01, inplace=False)
    (9): Linear(in_features=4, out_features=2, bias=True)
    (10): ReLU()
```

```
    (11): Dropout(p=0.01, inplace=False)
    (12): Linear(in_features=2, out_features=1, bias=True)
  )
)
128 values in Layer 0.
16 padding values added.
144 values now in Layer 0.
```

```
64 values in Layer 3.
64 values now in Layer 3.
```

```
64 values in Layer 6.
64 values now in Layer 6.
```

```
32 values in Layer 9.
4 padding values added.
36 values now in Layer 9.
```

```
16 values in Layer 12.
16 values now in Layer 12.
```

Activations Plot for 12
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0.310
0.309
0.308
0.307
0.306
0.305
Value

31 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 31.1. Section 31.2 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 31.3.

31.1 Transformer Basics

31.1.1 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.

31.1.1.1 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 31.1. 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.

31.1.1.2 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.

31.1.2 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.

Figure 31.1: 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.

Calculation of the self-attention:

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.

31.1.3 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.

31.1.4 Masked Self-Attention

Masked self-attention is a variant of the self-attention method described in Section 31.1.3. 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).

31.1.5 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.

31.1.6 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.

31.2 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 31.1 are used in the Transformer architecture.

Table 31.1: 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 31.1, 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 31.2: 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)

31.2.1 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 31.1, we considered Q_2 and K_1 as well as Q_2 and K_2). The dot product attention is calculated as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(QK^T\right)V \tag{31.1}$$

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).

31.2.2 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 31.1 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 31.3.

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:

Figure 31.3: Scaled dot product attention. Figure credit Vaswani et al. (2017)

31.3 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)
```

31.3.1 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)
```

31.3.2 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.

31.3.3 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.]])
```

31.3.4 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 $W^Q_{1...h} \in \mathbb{R}^{D\times d_k}$, $W^K_{1...h} \in \mathbb{R}^{D\times d_k}$, $W^V_{1...h} \in \mathbb{R}^{D\times d_v}$, 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 31.4.

Figure 31.4: 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 31.4, 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 heads."

        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
```

31.3.5 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 31.3.8.

31.3.6 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 31.5.

Figure 31.5: Transformer architecture. Figure credit: Vaswani et al. (2017)

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

$$\text{LayerNorm}(x + \text{Multihead}(x, x, x))$$

(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.

31.3.7 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:

$$\begin{aligned} \text{FFN}(x) &= \max(0, xW_1 + b_1)W_2 + b_2 \\ x &= \text{LayerNorm}(x + \text{FFN}(x)) \end{aligned}$$

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
```

31.3.8 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) / d_mo
        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.

31.3.9 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 31.6: 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.

31.3.10 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.

```python
class TransformerPredictor(pl.LightningModule):

    def __init__(self, input_dim, model_dim, num_classes, num_heads, num_layers, lr, warmup,
        """
        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
            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)
        )
```

```python
    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 input.
                                      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 single
        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
```

31.4 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.

31.4.1 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).

31.4.2 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.

31.4.3 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.

31.4.4 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 need

    # 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), **kwargs)
        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_acc"]}

    model = model.to(device)
    return model, result
```

31.4.5 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.

31.5 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_layers*fig
    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.

31.6 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.

31.7 Additional Considerations

31.7.1 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 31.7 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 31.7.

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 31.7: Comparison of complexity and path length of different sequence layers. Table taken from Lippe (2022)

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.

31.8 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.

32 Hyperparameter Tuning of a Transformer Network with PyTorch Lightning

32.1 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_control_i
from spotpython.utils.eda import gen_design_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.Spot(fun=fun,fun_control=fun_control, design_control=design_control, surrog
```

```
Moving TENSORBOARD_PATH: runs/ to TENSORBOARD_PATH_OLD: runs_OLD/runs_2024_10_13_00_20_12
Created spot_tensorboard_path: runs/spot_logs/603_maans14_2024-10-13_00-20-12 for SummaryWrit
module_name: light
submodule_name: regression
model_name: NNTransformerRegressor
```

We can take a look at the design table to see the initial design.

```
print(gen_design_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
```

If we want to run the hyperparameter tuning process on a remote server, we can save the setting as a `pickle` file and load it on the remote server.

```
filename = get_experiment_filename(fun_control["PREFIX"])
# if userExperimnents directory does not exist, create it
if not os.path.exists("userExperiment"):
    os.makedirs("userExperiment")
filename = os.path.join("userExperiment", filename)
if spot_tuner.spot_writer is not None:
    spot_tuner.spot_writer.close()
# remove attribute spot_writer from spot_tuner object
if hasattr(spot_tuner, "spot_writer"):
    delattr(spot_tuner, "spot_writer")
spot_tuner.save_experiment(filename=filename)
```

```
Experiment saved to userExperiment/spot_603_experiment.pickle
```

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': 23954.24609375, 'hp_metric': 23954.24609375}
d_model: 128, dim_feedforward: 256

train_model result: {'val_loss': 20689.533203125, 'hp_metric': 20689.533203125}
d_model: 32, dim_feedforward: 64

train_model result: {'val_loss': 23385.4296875, 'hp_metric': 23385.4296875}
d_model: 16, dim_feedforward: 32

train_model result: {'val_loss': 23920.986328125, 'hp_metric': 23920.986328125}
d_model: 8, dim_feedforward: 16

train_model result: {'val_loss': 23945.990234375, 'hp_metric': 23945.990234375}
d_model: 128, dim_feedforward: 256

train_model result: {'val_loss': 19936.63671875, 'hp_metric': 19936.63671875}
spotpython tuning: 19936.63671875 [#---------] 14.90%
d_model: 128, dim_feedforward: 256

train_model result: {'val_loss': 19263.798828125, 'hp_metric': 19263.798828125}
spotpython tuning: 19263.798828125 [###-------] 29.94%

d_model: 128, dim_feedforward: 256

train_model result: {'val_loss': 18752.330078125, 'hp_metric': 18752.330078125}
spotpython tuning: 18752.330078125 [#####-----] 47.89%

d_model: 128, dim_feedforward: 256

train_model result: {'val_loss': 19413.541015625, 'hp_metric': 19413.541015625}
spotpython tuning: 18752.330078125 [#######---] 73.50%
```

```
d_model: 128, dim_feedforward: 256

train_model result: {'val_loss': 20917.896484375, 'hp_metric': 20917.896484375}
spotpython tuning: 18752.330078125 [########--] 83.54%

d_model: 128, dim_feedforward: 256

train_model result: {'val_loss': 20649.611328125, 'hp_metric': 20649.611328125}
spotpython tuning: 18752.330078125 [##########] 97.92%

d_model: 32, dim_feedforward: 64

train_model result: {'val_loss': 22397.33203125, 'hp_metric': 22397.33203125}
spotpython tuning: 18752.330078125 [##########] 100.00% Done...
```

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/"
```

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.

```
spot_tuner.plot_progress(log_y=True, filename=None)
```

32.2.2 Tuned Hyperparameters and Their Importance

Results can be printed in tabular form.

```
from spotpython.utils.eda import gen_design_table
print(gen_design_table(fun_control=fun_control, spot=spot_tuner))
```

```
| name                  | type   | default        |   lower |   upper | tuned           | tra
|-----------------------|--------|----------------|---------|---------|-----------------|----
| d_model_mult          | int    | 4              |     1.0 |     5.0 | 5.0             | tra
| nhead                 | int    | 3              |     1.0 |     2.0 | 2.0             | tra
| num_encoder_layers    | int    | 1              |     1.0 |     4.0 | 1.0             | tra
| dim_feedforward_mult  | int    | 1              |     1.0 |     1.0 | 1.0             | tra
| epochs                | int    | 7              |     5.0 |     7.0 | 7.0             | tra
| batch_size            | int    | 5              |     5.0 |     8.0 | 5.0             | tra
| optimizer             | factor | Adam           |     0.0 |     4.0 | Adamax          | Non
| dropout               | float  | 0.1            |    0.01 |     0.1 | 0.1             | Non
| lr_mult               | float  | 0.1            |    0.01 |     0.3 | 0.3             | Non
| patience              | int    | 5              |     4.0 |     7.0 | 4.0             | tra
| initialization        | factor | xavier_uniform |     0.0 |     3.0 | kaiming_uniform | Non
```

32.3 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`).

32.3.1 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`.

32.3.2 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`.

32.4 Summary

This section presented an introduction to the basic setup of hyperparameter tuning of a transformer with `spotpython` and `PyTorch` Lightning.

33 Saving and Loading

33.1 Overview

This tutorial shows how to save and load objects in `spotpython`. It is split into the following parts: - Section 33.2 shows how to save and load objects in `spotpython`, if `spotpython` is used as an optimizer. - Section 33.4 shows how to save and load hyperparameter tuning experiments. - Section 33.5 shows how to save and load `PyTorch Lightning` models. - Section 33.6 shows how to convert a `PyTorch Lightning` model to a plain `PyTorch` model.

33.2 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.

33.3 spotpython 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.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}")
```

```
spotpython tuning: 4.7932399644479124 [########--] 75.00%
spotpython tuning: 2.0379524815147736 [#########-] 87.50%
spotpython tuning: 1.9863250753932071 [##########] 100.00% Done...

Experiment saved to spot_branin_experiment.pickle
filename: spot_branin_experiment.pickle
```

```
(spot_tuner_1, fun_control_1, design_control_1,
    surrogate_control_1, optimizer_control_1) = load_experiment(filename)
```

```
Loaded experiment from spot_branin_experiment.pickle
```

The progress of the original experiment is shown in **?@fig-plot-progress-37a** and the reloaded experiment in **?@fig-plot-progress-37b**.

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

Figure 33.1: Progress of the original experiment

```
spot_tuner_1.plot_progress(log_y=True)
```

Table 33.1

```
[['x0', 10.0], ['x1', 3.2107652010539627]]
```

Figure 33.2: Progress of the reloaded experiment

The results from the original experiment are shown in **?@tbl-results-37a** and the reloaded experiment in **?@tbl-results-37b**.

```
spot_tuner.print_results()
```

```
min y: 1.9863250753932071
x0: 10.0
x1: 3.2107652010539627
```

```
spot_tuner_1.print_results()
```

```
min y: 1.9863250753932071
x0: 10.0
x1: 3.2107652010539627
```

33.3.0.1 Getting the Tuned Hyperparameters

The tuned hyperparameters can be obtained as a dictionary with the following code.

Table 33.2

```
[['x0', 10.0], ['x1', 3.2107652010539627]]
```

```
from spotpython.hyperparameters.values import get_tuned_hyperparameters
get_tuned_hyperparameters(spot_tuner=spot_tuner)
```

```
{'x0': 10.0, 'x1': 3.2107652010539627}
```

> 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.

33.4 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.

```
import warnings
warnings.filterwarnings("ignore")
```

33.4.1 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
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_i
from spotpython.utils.eda import gen_design_table
from spotpython.spot import spot
from spotpython.utils.file import get_experiment_filename

PREFIX="604"

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

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(gen_design_table(fun_control))
```

```
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   |   5     | 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                  |
```

```python
fun_control.update({"save_experiment": True})
```

```python
spot_tuner = spot.Spot(fun=fun,fun_control=fun_control, design_control=design_control)
spot_tuner.save_experiment(path="userExperiment", overwrite=False)
res = spot_tuner.run()
```

```
Error: File userExperiment/spot_604_experiment.pickle already exists. Use overwrite=True to o

train_model result: {'val_loss': 23995.974609375, 'hp_metric': 23995.974609375}

train_model result: {'val_loss': 24026.373046875, 'hp_metric': 24026.373046875}

train_model result: {'val_loss': 23620.583984375, 'hp_metric': 23620.583984375}

train_model result: {'val_loss': 24180.556640625, 'hp_metric': 24180.556640625}

train_model result: {'val_loss': 23986.830078125, 'hp_metric': 23986.830078125}

train_model result: {'val_loss': 24038.296875, 'hp_metric': 24038.296875}

train_model result: {'val_loss': 23852.763671875, 'hp_metric': 23852.763671875}

train_model result: {'val_loss': 22770.611328125, 'hp_metric': 22770.611328125}
```

```
train_model result: {'val_loss': 23918.626953125, 'hp_metric': 23918.626953125}

train_model result: {'val_loss': 23849.4296875, 'hp_metric': 23849.4296875}
train_model result: {'val_loss': 23438.02734375, 'hp_metric': 23438.02734375}
spotpython tuning: 22770.611328125 [----------] 2.15%

train_model result: {'val_loss': 23946.251953125, 'hp_metric': 23946.251953125}
spotpython tuning: 22770.611328125 [----------] 4.48%

train_model result: {'val_loss': 23783.455078125, 'hp_metric': 23783.455078125}
spotpython tuning: 22770.611328125 [#---------] 7.81%

train_model result: {'val_loss': 4903.05126953125, 'hp_metric': 4903.05126953125}
spotpython tuning: 4903.05126953125 [#---------] 9.75%

train_model result: {'val_loss': 5344.9443359375, 'hp_metric': 5344.9443359375}
spotpython tuning: 4903.05126953125 [#---------] 13.54%

train_model result: {'val_loss': 24064.033203125, 'hp_metric': 24064.033203125}
spotpython tuning: 4903.05126953125 [##--------] 18.01%

train_model result: {'val_loss': 23473.666015625, 'hp_metric': 23473.666015625}
spotpython tuning: 4903.05126953125 [##--------] 22.09%

train_model result: {'val_loss': 23331.263671875, 'hp_metric': 23331.263671875}
spotpython tuning: 4903.05126953125 [###-------] 29.44%

train_model result: {'val_loss': 23682.443359375, 'hp_metric': 23682.443359375}
spotpython tuning: 4903.05126953125 [####------] 43.07%

train_model result: {'val_loss': 3258.4697265625, 'hp_metric': 3258.4697265625}
spotpython tuning: 3258.4697265625 [######----] 55.55%

train_model result: {'val_loss': 23730.748046875, 'hp_metric': 23730.748046875}
spotpython tuning: 3258.4697265625 [######----] 64.05%

train_model result: {'val_loss': 22964.625, 'hp_metric': 22964.625}
spotpython tuning: 3258.4697265625 [#######---] 73.29%
```

```
train_model result: {'val_loss': 21555.431640625, 'hp_metric': 21555.431640625}
spotpython tuning: 3258.4697265625 [#########-] 86.62%

train_model result: {'val_loss': 23992.6328125, 'hp_metric': 23992.6328125}
spotpython tuning: 3258.4697265625 [#########-] 92.97%

train_model result: {'val_loss': 24022.75390625, 'hp_metric': 24022.75390625}
spotpython tuning: 3258.4697265625 [##########] 97.82%

train_model result: {'val_loss': 24006.078125, 'hp_metric': 24006.078125}
spotpython tuning: 3258.4697265625 [##########] 100.00% Done...

Experiment saved to spot_604_experiment.pickle
```

The tuned hyperparameters can be obtained as a dictionary with the following code.

```
from spotpython.hyperparameters.values import get_tuned_hyperparameters
get_tuned_hyperparameters(spot_tuner)
```

```
{'l1': 4.0,
 'epochs': 5.0,
 'batch_size': 6.0,
 'act_fn': 3.0,
 'optimizer': 0.0,
 'dropout_prob': 0.024999427371776895,
 'lr_mult': 3.1577462928629245,
 'patience': 2.0,
 'batch_norm': 0.0,
 'initialization': 3.0}
```

Here , 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, fun_control)
```

```
{'l1': 4.0,
 'epochs': 5.0,
 'batch_size': 6.0,
 'act_fn': 'LeakyReLU',
 'optimizer': 'Adadelta',
 'dropout_prob': 0.024999427371776895,
 'lr_mult': 3.1577462928629245,
 'patience': 2.0,
 'batch_norm': 0,
 'initialization': 'xavier_uniform'}
```

```
spot_tuner.save_experiment(path="userExperiment", overwrite=False)
```

```
Error: File userExperiment/spot_604_experiment.pickle already exists. Use overwrite=True to o
```

The results from the experiment are stored in the pickle file spot_branin_experiment.pickle. The experiment can be reloaded with the following code.

```
print(f"filename: {filename}")
```

```
filename: spot_branin_experiment.pickle
```

```
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)
```

```
Loaded experiment from spot_604_experiment.pickle
```

Plot the progress of the original experiment are identical to the reloaded experiment.

```
spot_tuner.plot_progress(log_y=True)
spot_tuner_1.plot_progress(log_y=True)
```

Finally, the tuned hyperparameters can be obtained as a dictionary from the reloaded experiment with the following code.

```
get_tuned_hyperparameters(spot_tuner_1, fun_control_1)
```

```
{'l1': 4.0,
 'epochs': 5.0,
 'batch_size': 6.0,
 'act_fn': 'LeakyReLU',
 'optimizer': 'Adadelta',
 'dropout_prob': 0.024999427371776895,
 'lr_mult': 3.1577462928629245,
 'patience': 2.0,
 'batch_norm': 0,
 'initialization': 'xavier_uniform'}
```

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.

33.5 Saving and Loading PyTorch Lightning Models

Section 33.2 and Section 33.4 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.

33.5.1 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. The `config` dictionary can be passed to the model without any modifications.

```
from spotpython.hyperparameters.values import get_tuned_architecture
config = get_tuned_architecture(spot_tuner, fun_control)
pprint.pprint(config)
```

```
{'act_fn': LeakyReLU(),
 'batch_norm': False,
 'batch_size': 64,
 'dropout_prob': 0.024999427371776895,
 'epochs': 32,
 'initialization': 'xavier_uniform',
 'l1': 16,
 'lr_mult': 3.1577462928629245,
 'optimizer': 'Adadelta',
 'patience': 4}
```

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)
```

```
        Test metric                DataLoader 0

         hp_metric               4248.83251953125
         val_loss                4248.83251953125


test_model result: {'val_loss': 4248.83251953125, 'hp_metric': 4248.83251953125}

(4248.83251953125, 4248.83251953125)
```

33.5.2 Load a Model from Checkpoint

```
from spotpython.light.loadmodel import load_light_from_checkpoint
model_loaded = load_light_from_checkpoint(config, fun_control)
```

```
config: {'l1': 16, 'epochs': 32, 'batch_size': 64, 'act_fn': LeakyReLU(), 'optimizer': 'Adade
Loading model with 16_32_64_LeakyReLU_Adadelta_0.025_3.1577_4_False_xavier_uniform_TEST from
Model: NNLinearRegressor(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=16, bias=True)
    (1): LeakyReLU()
    (2): Dropout(p=0.024999427371776895, inplace=False)
    (3): Linear(in_features=16, out_features=8, bias=True)
    (4): LeakyReLU()
    (5): Dropout(p=0.024999427371776895, inplace=False)
    (6): Linear(in_features=8, out_features=8, bias=True)
    (7): LeakyReLU()
    (8): Dropout(p=0.024999427371776895, inplace=False)
    (9): Linear(in_features=8, out_features=4, bias=True)
    (10): LeakyReLU()
    (11): Dropout(p=0.024999427371776895, inplace=False)
    (12): Linear(in_features=4, out_features=4, bias=True)
    (13): LeakyReLU()
    (14): Dropout(p=0.024999427371776895, inplace=False)
    (15): Linear(in_features=4, out_features=4, bias=True)
```

```
    (16): LeakyReLU()
    (17): Dropout(p=0.024999427371776895, inplace=False)
    (18): Linear(in_features=4, out_features=2, bias=True)
    (19): LeakyReLU()
    (20): Dropout(p=0.024999427371776895, inplace=False)
    (21): Linear(in_features=2, out_features=2, bias=True)
    (22): LeakyReLU()
    (23): Dropout(p=0.024999427371776895, inplace=False)
    (24): Linear(in_features=2, out_features=2, bias=True)
    (25): LeakyReLU()
    (26): Dropout(p=0.024999427371776895, inplace=False)
    (27): Linear(in_features=2, out_features=2, bias=True)
    (28): LeakyReLU()
    (29): Dropout(p=0.024999427371776895, inplace=False)
    (30): Linear(in_features=2, out_features=1, bias=True)
  )
)
```

```
vars(model_loaded)
```

```
{'training': False,
 '_parameters': OrderedDict(),
 '_buffers': OrderedDict(),
 '_non_persistent_buffers_set': set(),
 '_backward_pre_hooks': OrderedDict(),
 '_backward_hooks': OrderedDict(),
 '_is_full_backward_hook': None,
 '_forward_hooks': OrderedDict(),
 '_forward_hooks_with_kwargs': OrderedDict(),
 '_forward_hooks_always_called': OrderedDict(),
 '_forward_pre_hooks': OrderedDict(),
 '_forward_pre_hooks_with_kwargs': OrderedDict(),
 '_state_dict_hooks': OrderedDict(),
 '_state_dict_pre_hooks': OrderedDict(),
 '_load_state_dict_pre_hooks': OrderedDict(),
 '_load_state_dict_post_hooks': OrderedDict(),
 '_modules': OrderedDict([('layers',
               Sequential(
                 (0): Linear(in_features=10, out_features=16, bias=True)
                 (1): LeakyReLU()
                 (2): Dropout(p=0.024999427371776895, inplace=False)
                 (3): Linear(in_features=16, out_features=8, bias=True)
```

```
                (4): LeakyReLU()
                (5): Dropout(p=0.024999427371776895, inplace=False)
                (6): Linear(in_features=8, out_features=8, bias=True)
                (7): LeakyReLU()
                (8): Dropout(p=0.024999427371776895, inplace=False)
                (9): Linear(in_features=8, out_features=4, bias=True)
                (10): LeakyReLU()
                (11): Dropout(p=0.024999427371776895, inplace=False)
                (12): Linear(in_features=4, out_features=4, bias=True)
                (13): LeakyReLU()
                (14): Dropout(p=0.024999427371776895, inplace=False)
                (15): Linear(in_features=4, out_features=4, bias=True)
                (16): LeakyReLU()
                (17): Dropout(p=0.024999427371776895, inplace=False)
                (18): Linear(in_features=4, out_features=2, bias=True)
                (19): LeakyReLU()
                (20): Dropout(p=0.024999427371776895, inplace=False)
                (21): Linear(in_features=2, out_features=2, bias=True)
                (22): LeakyReLU()
                (23): Dropout(p=0.024999427371776895, inplace=False)
                (24): Linear(in_features=2, out_features=2, bias=True)
                (25): LeakyReLU()
                (26): Dropout(p=0.024999427371776895, inplace=False)
                (27): Linear(in_features=2, out_features=2, bias=True)
                (28): LeakyReLU()
                (29): Dropout(p=0.024999427371776895, inplace=False)
                (30): Linear(in_features=2, out_features=1, bias=True)
              ))]),
'prepare_data_per_node': True,
'allow_zero_length_dataloader_with_multiple_devices': False,
'_log_hyperparams': True,
'_dtype': torch.float32,
'_device': device(type='mps', index=0),
'_trainer': None,
'_example_input_array': tensor([[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., 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., 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., 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., 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.],
[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., 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., 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., 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., 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.],
[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., 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., 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., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]),
'_automatic_optimization': True,
'_strict_loading': None,
'_current_fx_name': None,
'_param_requires_grad_state': {},
'_metric_attributes': None,
'_compiler_ctx': None,
'_fabric': None,
'_fabric_optimizers': [],
'_device_mesh': None,
'_L_in': 10,
'_L_out': 1,
'_torchmetric': 'mean_squared_error',
'metric': <function torchmetrics.functional.regression.mse.mean_squared_error(preds: torch.T
'_hparams_name': 'kwargs',
'_hparams': "act_fn":         LeakyReLU()
"batch_norm":     False
"batch_size":     64
"dropout_prob":   0.024999427371776895
"epochs":         32
"initialization": xavier_uniform
"l1":             16
"lr_mult":        3.1577462928629245
"optimizer":      Adadelta
"patience":       4,
'_hparams_initial': "act_fn":         LeakyReLU()
"batch_norm":     False
"batch_size":     64
"dropout_prob":   0.024999427371776895
"epochs":         32
"initialization": xavier_uniform
"l1":             16
```

```
 "lr_mult":        3.1577462928629245
 "optimizer":      Adadelta
 "patience":       4}
```

```python
import torch
torch.save(model_loaded, "model.pt")
```

```python
mymodel = torch.load("model.pt")
```

```python
# show all attributes of the model
vars(mymodel)
```

```
{'training': False,
 '_parameters': OrderedDict(),
 '_buffers': OrderedDict(),
 '_non_persistent_buffers_set': set(),
 '_backward_pre_hooks': OrderedDict(),
 '_backward_hooks': OrderedDict(),
 '_is_full_backward_hook': None,
 '_forward_hooks': OrderedDict(),
 '_forward_hooks_with_kwargs': OrderedDict(),
 '_forward_hooks_always_called': OrderedDict(),
 '_forward_pre_hooks': OrderedDict(),
 '_forward_pre_hooks_with_kwargs': OrderedDict(),
 '_state_dict_hooks': OrderedDict(),
 '_state_dict_pre_hooks': OrderedDict(),
 '_load_state_dict_pre_hooks': OrderedDict(),
 '_load_state_dict_post_hooks': OrderedDict(),
 '_modules': OrderedDict([('layers',
               Sequential(
                 (0): Linear(in_features=10, out_features=16, bias=True)
                 (1): LeakyReLU()
                 (2): Dropout(p=0.024999427371776895, inplace=False)
                 (3): Linear(in_features=16, out_features=8, bias=True)
                 (4): LeakyReLU()
                 (5): Dropout(p=0.024999427371776895, inplace=False)
                 (6): Linear(in_features=8, out_features=8, bias=True)
                 (7): LeakyReLU()
                 (8): Dropout(p=0.024999427371776895, inplace=False)
                 (9): Linear(in_features=8, out_features=4, bias=True)
                 (10): LeakyReLU()
```

```
                (11): Dropout(p=0.024999427371776895, inplace=False)
                (12): Linear(in_features=4, out_features=4, bias=True)
                (13): LeakyReLU()
                (14): Dropout(p=0.024999427371776895, inplace=False)
                (15): Linear(in_features=4, out_features=4, bias=True)
                (16): LeakyReLU()
                (17): Dropout(p=0.024999427371776895, inplace=False)
                (18): Linear(in_features=4, out_features=2, bias=True)
                (19): LeakyReLU()
                (20): Dropout(p=0.024999427371776895, inplace=False)
                (21): Linear(in_features=2, out_features=2, bias=True)
                (22): LeakyReLU()
                (23): Dropout(p=0.024999427371776895, inplace=False)
                (24): Linear(in_features=2, out_features=2, bias=True)
                (25): LeakyReLU()
                (26): Dropout(p=0.024999427371776895, inplace=False)
                (27): Linear(in_features=2, out_features=2, bias=True)
                (28): LeakyReLU()
                (29): Dropout(p=0.024999427371776895, inplace=False)
                (30): Linear(in_features=2, out_features=1, bias=True)
              ))]),
'prepare_data_per_node': True,
'allow_zero_length_dataloader_with_multiple_devices': False,
'_log_hyperparams': True,
'_dtype': torch.float32,
'_device': device(type='mps', index=0),
'_trainer': None,
'_example_input_array': tensor([[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., 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., 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., 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., 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.],
[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., 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., 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., 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., 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.],
[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., 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., 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.]]),
'_automatic_optimization': True,
'_strict_loading': None,
'_current_fx_name': None,
'_param_requires_grad_state': {},
'_metric_attributes': None,
'_compiler_ctx': None,
'_fabric': None,
'_fabric_optimizers': [],
'_device_mesh': None,
'_L_in': 10,
'_L_out': 1,
'_torchmetric': 'mean_squared_error',
'metric': <function torchmetrics.functional.regression.mse.mean_squared_error(preds: torch.T
'_hparams_name': 'kwargs',
'_hparams': "act_fn":         LeakyReLU()
"batch_norm":     False
"batch_size":     64
"dropout_prob":   0.024999427371776895
"epochs":         32
"initialization": xavier_uniform
"l1":             16
"lr_mult":        3.1577462928629245
"optimizer":      Adadelta
"patience":       4,
'_hparams_initial': "act_fn":         LeakyReLU()
"batch_norm":     False
"batch_size":     64
"dropout_prob":   0.024999427371776895
"epochs":         32
"initialization": xavier_uniform
"l1":             16
"lr_mult":        3.1577462928629245
"optimizer":      Adadelta
"patience":       4}
```

33.6 Converting a Lightning Model to a Plain Torch Model

33.6.1 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)
```

```
{'_automatic_optimization': True, '_current_fx_name': None, '_device': device(type='mps', inc
"batch_norm":     False
"batch_size":     64
"dropout_prob":   0.024999427371776895
"epochs":         32
"initialization": xavier_uniform
"l1":             16
"lr_mult":        3.1577462928629245
"optimizer":      Adadelta
"patience":       4, '_fabric': None, '_example_input_array': tensor([[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., 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., 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.],
        [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.]]), '_strict_loading': None, '_fabric_optimiz
"batch_norm":     False
"batch_size":     64
"dropout_prob":   0.024999427371776895
"epochs":         32
"initialization": xavier_uniform
"l1":             16
"lr_mult":        3.1577462928629245
"optimizer":      Adadelta
"patience":       4, 'metric': <function mean_squared_error at 0x31c8387c0>, '_L_in': 10, '_
```

```
print(torch_net)
```

```
NNLinearRegressor(
  (layers): Sequential(
    (0): Linear(in_features=10, out_features=16, bias=True)
    (1): LeakyReLU()
    (2): Dropout(p=0.024999427371776895, inplace=False)
    (3): Linear(in_features=16, out_features=8, bias=True)
    (4): LeakyReLU()
    (5): Dropout(p=0.024999427371776895, inplace=False)
    (6): Linear(in_features=8, out_features=8, bias=True)
    (7): LeakyReLU()
    (8): Dropout(p=0.024999427371776895, inplace=False)
    (9): Linear(in_features=8, out_features=4, bias=True)
    (10): LeakyReLU()
    (11): Dropout(p=0.024999427371776895, inplace=False)
    (12): Linear(in_features=4, out_features=4, bias=True)
    (13): LeakyReLU()
    (14): Dropout(p=0.024999427371776895, inplace=False)
    (15): Linear(in_features=4, out_features=4, bias=True)
    (16): LeakyReLU()
    (17): Dropout(p=0.024999427371776895, inplace=False)
```

```
    (18): Linear(in_features=4, out_features=2, bias=True)
    (19): LeakyReLU()
    (20): Dropout(p=0.024999427371776895, inplace=False)
    (21): Linear(in_features=2, out_features=2, bias=True)
    (22): LeakyReLU()
    (23): Dropout(p=0.024999427371776895, inplace=False)
    (24): Linear(in_features=2, out_features=2, bias=True)
    (25): LeakyReLU()
    (26): Dropout(p=0.024999427371776895, inplace=False)
    (27): Linear(in_features=2, out_features=2, bias=True)
    (28): LeakyReLU()
    (29): Dropout(p=0.024999427371776895, inplace=False)
    (30): Linear(in_features=2, out_features=1, bias=True)
  )
)
```

33.6.2 An Example how to use the Plain Torch Net

```python
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=42)

# 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()
```

120000
100000
80000
60000
40000
20000
0
Loss
0
200
400
600
800
1000
1200
Epoch

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
−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/numerisc` 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-sommersemester2023.gi

# 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/studiengaenge/f07/ordnungen_plaene/f07_`

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_control_in
spot_1 = spot.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: 2.0030463147492306 [#########-] 85.71%
spotpython tuning: 0.010360622165042565 [##########] 100.00% Done...
```

```
<spotpython.spot.spot.Spot at 0x33fb39820>
```

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_control_in
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.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: 1.4886245843852055e-05 [##########] 100.00% Done...

<spotpython.spot.spot.Spot at 0x340555250>
```

D.5 Print the Results

```
spot_tuner.print_results()
```

```
min y: 1.4886245843852055e-05
x0: -0.003850983818095356
x1: -0.00023700100552480994

[['x0', -0.003850983818095356], ['x1', -0.00023700100552480994]]
```

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.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: 2.0030463147492306 [#########-] 85.71%
spotpython tuning: 0.010360622165042565 [##########] 100.00% Done...
```

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
```

```python
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.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)
```

```
[        nan         nan -0.02203599 -0.21843718  0.78240941         nan
 -0.3923345   0.67234256  0.31802454 -0.68898927 -0.75129705  0.97550354
  0.41757584         nan  0.82585329         nan -0.49274073         nan
 -0.17991251  0.1481835 ]

[-1.]
[nan]
[-0.57084122]
[0.166475]
[nan]
[-0.24270361]
[-0.47259301]
[0.95541987]
[0.17335968]
[-0.58552368]
[-0.20126111]
[-0.60100809]
[-0.97897336]
[-0.2748985]
[0.8359486]
[0.99035591]
[0.01641232]
[0.5629346]
```

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 gen_design_table
fun_control = fun_control_init(
    core_model_name="forest.AMFRegressor",
    hyperdict=RiverHyperDict,
)
print("Before modification:")
print(gen_design_table(fun_control))
set_int_hyperparameter_values(fun_control, "n_estimators", 2, 5)
print("After modification:")
print(gen_design_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 gen_design_table
```

```
fun_control = fun_control_init(
    core_model_name="forest.AMFRegressor",
    hyperdict=RiverHyperDict,
)
print("Before modification:")
print(gen_design_table(fun_control))
set_float_hyperparameter_values(fun_control, "step", 0.2, 5)
print("After modification:")
print(gen_design_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 gen_design_table
fun_control = fun_control_init(
    core_model_name="forest.AMFRegressor",
    hyperdict=RiverHyperDict,
)
```

```
print("Before modification:")
print(gen_design_table(fun_control))
set_boolean_hyperparameter_values(fun_control, "use_aggregation", 0, 0)
print("After modification:")
print(gen_design_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 gen_design_table
fun_control = fun_control_init(
    core_model_name="tree.HoeffdingTreeRegressor",
    hyperdict=RiverHyperDict,
)
print("Before modification:")
print(gen_design_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
| 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_1
| 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] + 5 * x
        )
    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] + 5 * x
        )
```

`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 Po
```

```
plot_data_with_drift_points(test, target_column, n_train, title="Testing Data with Drift Poi
```

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=True):
    data_generator = FriedmanDriftDataset(n_samples=n_samples, seed=seed, change_point1=chang
    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
plot_friedman_drift_data(n_samples=100, seed=42, change_point1=50, change_point2=75, constant
```

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/python

### Vergabe von Ressourcen
#SBATCH --job-name=CH10_Test
#SBATCH --ntasks-per-node=64
#SBATCH --gres=gpu:1
#SBATCH --time=24:00:00
#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

conda activate spot312

srun python startPython.py "$SPOT_PKL"
```

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

def main(pickle_file):
    spot_tuner = load_and_run_spot_python_experiment(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 be proc

    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:

```python
import os
from math import inf
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, surrogate_control_i
from spotpython.utils.eda import gen_design_table
from spotpython.hyperparameters.values import set_hyperparameter
from spotpython.spot import spot
from spotpython.utils.scaler import TorchStandardScaler
from spotpython.utils.device import getDevice
from spotpython.utils.file import get_experiment_filename

# load data (adapt this to your needs)
from pyhcf.utils.io import load_hcf_df, hcf_df2tensor
from pyhcf.utils.names import load_all_features_N_regression_list
df = load_hcf_df(A=True, H=True, param_list=load_all_features_N_regression_list(), target='N
data_set = hcf_df2tensor(df, target='N', return_X_y=False)

fun_control = fun_control_init(
    PREFIX="01",
    TENSORBOARD_CLEAN=False,
    tensorboard_log=False,
    fun_evals=inf,
    max_time=600,
    data_set=data_set,
    scaler=TorchStandardScaler(),
    device=getDevice(),
    max_surrogate_points=30,
    core_model_name="light.regression.NNLinearRegressor",
    hyperdict=LightHyperDict,
    _L_in=86,
    _L_out=1)

# The following "set_hyperparameter"-block is optional.
# You can adapt the hyperparameters to your needs as follows:
set_hyperparameter(fun_control, "optimizer", [
                "Adadelta",
                "Adagrad",
                "Adam",
                "AdamW",
                "Adamax",
            ])
```

```
set_hyperparameter(fun_control, "l1", [3,9])
set_hyperparameter(fun_control, "epochs", [10,12])
set_hyperparameter(fun_control, "batch_size", [4,8])
set_hyperparameter(fun_control, "dropout_prob", [0.0, 0.1])
set_hyperparameter(fun_control, "lr_mult", [0.1, 20.0])
set_hyperparameter(fun_control, "patience", [6,9])

# Modify the design control (optional)
design_control = design_control_init(init_size=20)
# Modify the surrogate control (optional)
surrogate_control = surrogate_control_init(
    noise=True,
    n_theta=2,
    min_Lambda=1e-3,
    max_Lambda=10,
)

# Check the design table
print(gen_design_table(fun_control))

# The remaining part is mandatory:
fun = HyperLight().fun
spot_tuner = spot.Spot(fun=fun,fun_control=fun_control, design_control=design_control, surrog
filename = get_experiment_filename(fun_control["PREFIX"])
# remove attribute spot_writer from spot_tuner object
if hasattr(spot_tuner, "spot_writer"):
    delattr(spot_tuner, "spot_writer")
spot_tuner.save_experiment(filename=filename)
```

The configuration is saved as a pickle-file that contains the full information. In our example, the filename is `spot_01_experiment.pickle`.

> **Data**
>
> The data is loaded from the `pyhcf` package. You can adapt the data loading to your needs. To generate dummy data, you can use the following code:
>
> ```
> num_samples = 1000
> input_dim = 86
> 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)
> ```

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 spot_01_experiment.pickle 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
cd /path/to/remote/directory
sh -x ./startSlurm.sh spot_01_experiment.pickle
```

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:spot_01_experiment.pickle .
```

F.7 Analyze the Results on the Local Machine

The file `spot_01_experiment.pickle` contains the results of the `spotpython` code. You can analyze the results on the local machine using the following code.

```
from spotpython.utils.file import load_experiment
spot_tuner, fun_control, design_control, surrogate_control, optimizer_control = load_experime
```

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 gen_design_table
print(gen_design_table(fun_control=fun_control, spot=spot_tuner))
```

F.7.3 Plotting Important Hyperparameters

```
spot_tuner.plot_important_hyperparameter_contour(max_imp=3)
```

> **spotgui**
>
> The `spotgui` can be used to analyze the results interactively.

G Solutions to Selected Exercises

> ⚠ Warning
>
> - Solutions are incomplete and need to be corrected!
> - They serve as a starting point for the final solution.

G.1 Data-Driven Modeling and Optimization

G.1.1 Histograms

Solution G.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

G.1.2 The Normal Distribution

Solution G.2 (TwoSDAnswer). 95%

Solution G.3 (OneSDAnswer). 68%

Solution G.4 (ThreeSDAnswer). 99,7%

Solution G.5 (DataRangeAnswer). 80 - 120

Solution G.6 (PeakHeightAnswer). low

G.1.3 The mean, the media, and the mode

G.1.4 The exponential distribution

G.1.5 Population and Estimated Parameters

Solution G.7 (ProbabilityAnswer). 50%

G.1.6 Calculating the Mean, Variance and Standard Deviation

Solution G.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 G.9 (EstimateMeanAnswer). Sum of the values divided by n.

Solution G.10 (SigmaSquaredAnswer). Variance

Solution G.11 (EstimatedSDAnswer). The same as the normal standard deviation, but using n-1.

Solution G.12 (VarianceDifferenceAnswer). n and $n-1$

Solution G.13 (ModelBenefitsAnswer).

- Approximation
- Prediction
- Understanding

Solution G.14 (SampleDefinitionAnswer). It's a subset of the data.

G.1.7 Hypothesis Testing and the Null-Hypothesis

Solution G.15 (RejectHypothesisAnswer). It means the evidence supports the alternative hypothesis, indicating that the null hypothesis is unlikely to be true.

Solution G.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 G.17 (BetterDrugAnswer). By conducting experiments and statistical tests to compare the new drug's effectiveness against the current standard and demonstrating a significant improvement.

G.1.8 Alternative Hypotheses, Main Ideas

G.1.9 p-values: What they are and how to interpret them

Solution G.18 (PValueIntroductionAnswer). We can reject the null hypothesis. We can make a decision.

Solution G.19 (PValueRangeAnswer). It can only be between 0 and 1.

Solution G.20 (PValueRangeAnswer). It can only be between 0 and 1.

Solution G.21 (TypicalPValueAnswer). The chance that we wrongly reject the null hypothesis.

Solution G.22 (FalsePositiveAnswer). If we have a false-positive, we succeed in rejecting the null hypothesis. But in fact/reality, this is false -> False positive.

G.1.10 How to calculate p-values

Solution G.23 (CalculatePValueAnswer). Probability of specific result, probability of outcome with the same probability, and probability of events with smaller probability.

Solution G.24 (SDCalculationAnswer). 7 is the SD.

Solution G.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 G.26 (CoinTestAnswer). TBD

Solution G.27 (BorderPValueAnswer). TBD

Solution G.28 (OneSidedPValueCautionAnswer). If you look in the wrong direction, there is no change.

Solution G.29 (BinomialDistributionAnswer). TBD

G.1.11 p-hacking: What it is and how to avoid it

Solution G.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 G.31 (AvoidPHackingAnswer). Specify the number of repeats and the sample sizes at the beginning.

Solution G.32 (MultipleTestingProblemAnswer). TBD

G.1.12 Covariance

Solution G.33 (CovarianceDefinitionAnswer). Formula

Solution G.34 (CovarianceMeaningAnswer). Large values in the first variable result in large values in the second variable.

Solution G.35 (CovarianceVarianceRelationshipAnswer). Formula

Solution G.36 (HighCovarianceAnswer). No, size doesn't matter.

Solution G.37 (ZeroCovarianceAnswer). No relationship

Solution G.38 (NegativeCovarianceAnswer). Yes

Solution G.39 (NegativeVarianceAnswer). No

G.1.13 Pearson's Correlation

Solution G.40 (CorrelationValueAnswer). Recalculate

Solution G.41 (CorrelationRangeAnswer). From -1 to 1

Solution G.42 (CorrelationFormulaAnswer). Formula

G.1.14 Boxplots

Solution G.43 (UnderstandingStatisticalPower). It is the probability of correctly rejecting the null hypothesis.

Solution G.44 (DistributionEffectOnPower). Power analysis is not applicable.

Solution G.45 (IncreasingPower). By taking more samples.

Solution G.46 (PreventingPHacking). TBD

Solution G.47 (SampleSizeAndPower). The power will be low.

G.1.15 Power Analysis

Solution G.48 (MainFactorsAffectingPower). The overlap (distance of the two means) and sample sizes.

Solution G.49 (PowerAnalysisOutcome). The sample size needed.

Solution G.50 (RisksInExperiments). Few experiments lead to very low power, and many experiments might result in p-hacking.

Solution G.51 (StepsToPerformPowerAnalysis).

1. Select power
2. Select threshold for significance (alpha)
3. Estimate the overlap (done by the effect size)

G.1.16 The Central Limit Theorem

Solution G.52 (CentralLimitTheoremAnswer). TBD

G.1.17 Boxplots

Solution G.53 (MedianAnswer). The median.

Solution G.54 (BoxContentAnswer). 50% of the data.

G.1.18 R-squared

Solution G.55 (RSquaredFormulaAnswer). TBD

Solution G.56 (NegativeRSquaredAnswer). If you fit a line, no, but there are cases where it could be negative. However, these are usually considered useless.

Solution G.57 (RSquaredCalculationAnswer). TBD

G.1.18.1 The main ideas of fitting a line to data (The main ideas of least squares and linear regression.)

Solution G.58 (LeastSquaresAnswer). It is the calculation of the smallest sum of residuals when you fit a model to data.

G.1.19 Linear Regression

G.1.20 Multiple Regression

G.1.21 A Gentle Introduction to Machine Learning

Solution G.59 (RegressionVsClassificationAnswer). Regression involves predicting continuous values (e.g., temperature, size), while classification involves predicting discrete values (e.g., categories like cat, dog).

G.1.22 Maximum Likelihood

Solution G.60 (LikelihoodConceptAnswer). The distribution that fits the data best.

G.1.23 Probability is not Likelihood

Solution G.61 (ProbabilityVsLikelihoodAnswer). Likelihood: Finding the curve that best fits the data. Probability: Calculating the probability of an event given a specific curve.

G.1.24 Cross Validation

Solution G.62 (TrainVsTestDataAnswer). Training data is used to fit the model, while testing data is used to evaluate how well the model fits.

Solution G.63 (SingleValidationIssueAnswer). The performance might not be representative because the data may not be equally distributed between training and testing sets.

Solution G.64 (FoldDefinitionAnswer). TBD

Solution G.65 (LeaveOneOutValidationAnswer). Only one data point is used as the test set, and the rest are used as the training set.

G.1.25 The Confusion Matrix

Solution G.66 (ConfusionMatrixAnswer). TBD

G.1.26 Sensitivity and Specificity

Solution G.67 (SensitivitySpecificityAnswer1). TBD

Solution G.68 (SensitivitySpecificityAnswer2). TBD

G.1.27 Bias and Variance

Solution G.69 (BiasAndVarianceAnswer). TBD

G.1.28 Mutual Information

Solution G.70 (MutualInformationExampleAnswer). TBD

G.1.29 Principal Component Analysis (PCA)

Solution G.71 (WhatIsPCAAnswer). A dimension reduction technique that helps discover important variables.

Solution G.72 (screePlotAnswer). It shows how much variation is defined by the data.

Solution G.73 (LeastSquaresInPCAAnswer). No, in the first step it tries to maximize distances.

Solution G.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 G.75 (EigenvaluePC1Answer). Formula (to be specified).

Solution G.76 (DifferencesBetweenPointsAnswer). No, because the first difference is measured on the PC1 scale and it is more important.

Solution G.77 (ScalingInPCAAnswer). Scaling by dividing by the standard deviation (SD).

Solution G.78 (DetermineNumberOfComponentsAnswer). TBD

Solution G.79 (LimitingNumberOfComponentsAnswer).

1. The dimension of the problem
2. Number of samples

G.1.30 t-SNE

Solution G.80 (WhyUseTSNEAnswer). For dimension reduction and picking out the relevant clusters.

Solution G.81 (MainIdeaOfTSNEAnswer). To reduce the dimensions of the data by reconstructing the relationships in a lower-dimensional space.

Solution G.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 G.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

G.1.31 K-means clustering

Solution G.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 G.85 (QualityOfClustersAnswer). Calculate the within-cluster variation.

Solution G.86 (IncreasingKAnswer). If k is too high, each point would be its own cluster. If k is too low, you cannot see the structures.

G.1.32 DBSCAN

Solution G.87 (CorePointInDBSCANAnswer). A point that is close to at least k other points.

Solution G.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 G.89 (OutliersInDBSCANAnswer). Points that are not core points and do not belong to existing clusters.

G.1.33 K-nearest neighbors

Solution G.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.

G.1.34 Naive Bayes

Solution G.91 (NaiveBayesFormulaAnswer). TBD

Solution G.92 (CalculateProbabilitiesAnswer). TBD

G.1.35 Gaussian Naive Bayes

Solution G.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.

G.1.36 Trees

Solution G.94 (Tree Usage). Classication, Regression, Clustering

Solution G.95 (Tree Usage). TBD

Solution G.96 (Tree Feature Importance). The most important feature.

Solution G.97 (Regression Tree Limitations). High dimensions

Solution G.98 (Regression Tree Score). SSR + alpha * T

Solution G.99 (Regression Tree Alpha Value Small). The tree is more complex.

Solution G.100 (Regression Tree Increase Alpha Value). We get smaller trees

Solution G.101 (Regression Tree Pruning). Decreases the complexity of the tree to enhance performance and reduce overfitting

G.2 Machine Learning and Artificial Intelligence

G.2.1 Backpropagation

Solution G.102 (ChainRuleAndGradientDescentAnswer). Combination of the chain rule and gradient descent.

Solution G.103 (BackpropagationNamingAnswer). Because you start at the end and go backwards.

G.2.2 Gradient Descent

Solution G.104 (GradDescStepSize). learning rate x slope

Solution G.105 (GradDescIntercept). Old intercept - step size

Solution G.106 (GradDescIntercept). When the step size is small or after a certain number of steps

G.2.3 ReLU

Solution G.107 (Graph ReLU). Graph of ReLU function: f(x) = max(0, x)

G.2.4 CNNs

Solution G.108 (CNNImageRecognitionAnswer).

- too many features for input layer -> high memory consumption
- always shift in data
- it learns local informations and local correlations

Solution G.109 (CNNFiltersInitializationAnswer). The filter values in CNNs are randomly initialized and then trained and optimized through the process of backpropagation.

Solution G.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 G.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.

G.2.5 RNN

Solution G.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 G.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.

G.2.6 LSTM

Solution G.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 G.115 (LSTMSigmoidTanhAnswer). State how much of the long term memory should be used.

Solution G.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 G.117 (LSTMLongTermInfoAnswer). Long-term information is used in the output gate of an LSTM network.

Solution G.118 (LSTMUpdateGatesAnswer). In the input and forget gates.

G.2.7 Pytorch/Lightning

Solution G.119 (PyTorchRequiresGradAnswer). In PyTorch, `requires_grad` indicates whether a tensor should be trained. If set to False, the tensor will not be trained.

G.2.8 Embeddings

Solution G.120 (NN STrings). No, they process numerical values.

Solution G.121 (Embedding Definition). Representation of a word as a vector.

Solution G.122 (Embedding Dimensions). We can model similarities.

G.2.9 Sequence to Sequence Models

Solution G.123 (LSTM). Because they are able to consider "far away" information.

Solution G.124 (Teacher Forcing). We need to force the correct words for the training.

Solution G.125 (Attention). Attention scores compute similarities for one input to the others.

G.2.10 Transformers

Solution G.126 (ChatGPT). Decoder only.

Solution G.127 (Translation). Encoder-Decoder structure.

Solution G.128 (Difference Encoder-Decoder and Decoder Only.).

- Encoder-Decoder: self-attention.
- Decoder only: masked self-attention.

Solution G.129 (Weights).

- a: Randomly
- b: Backpropagation

Solution G.130 (Order of Words). Positional Encoding

Solution G.131 (Relationship Between Words). Masked self-attention which looks at the previous tokens.

Solution G.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 G.133 (Softmax). Transformation to values between 0 and 1.

Solution G.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 G.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 G.136 (Residual Connections). They are bypasses, which combine the position encoded values with masked-self-attention values.

Solution G.137 (Generate Known Word in Sequence).

- Training
- Because it is a Decoder-Only transformer used for prediction and the calculations that you need.

Solution G.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.

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 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.

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.

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.

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."

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.

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.