UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

DIPARTIMENTO DI FISICA

CORSO DI LAUREA TRIENNALE IN FISICA

A deep learning approach to event generator tuning

Candidato:
Marco Lazzarin
Matricola 887355

Relatore:
Dott. Stefano Carrazza
Correlatori:
Prof. Simone Alioli
Prof. Stefano Forte

Anno accademico 2018/2019

Abstract

Monte Carlo event generators are tools that simulate the collision of particles at high energies. They introduce many parameters, mainly due to the usage of phenomenological models like the hadronization model or the underlying event model. These parameters are difficult to obtain on a theoretical basis, so they must be carefully tuned in order to make the generators reproduce the experimental measurements. The procedure of estimating the best value for each parameter is called **event generator tuning**. This tuning procedure is made more difficult by the high computational cost of running a generator, so it requires methods to study the dependence between a generator output and its parameters. The current state-of-the-art tuning procedure is based on a polynomial parametrisation of the generator response to parameter variations, followed by a numerical fit of the parametrised behaviour to experimental data. This procedure is implemented in the `Professor` tool. This thesis investigates new tuning procedures based on deep learning algorithms. In brief, deep learning algorithms employ parametric models called artificial neural networks to solve machine learning and artificial intelligence problems. Two different tuning procedures are presented, called Per Bin and Inverse from now on. The former follows the same approach of `Professor`, but with a different parametrisation model made of fully-connected neural networks and a different minimization algorithm, which is an evolutionary algorithm called Covariance Matrix Adaptation - Evolution Strategy. The latter takes a completely different approach: by using a fully-connected neural network, it learns to predict directly the parameters that the generator needs to output a given result. These two procedures were implemented in a `Python` package, called `MCNNTUNES` from now on, and then tested with the event generator `PYTHIA8`. Two different datasets of Monte Carlo runs were generated, with three and four tunable parameters respectively. The procedures were tested with pseudo-data (i.e. other Monte Carlo runs) and with real experimental data taken from the ATLAS experiment. The tests with pseudo-data (called closure tests from now on) consist on replacing the experimental data with other Monte Carlo runs, and comparing the obtained tunes with the actual parameters used to generate those runs. These are useful to verify if the models are really able to learn the generator behaviour. The tests with real experimental measurements, instead, are useful to see if the tuning procedures are practical and behave reasonably in an actual tuning context. The Per Bin model closure tests were very limited, due to computational time constraints, but showed solid results. The test with experimental data showed a behaviour similar to the one of `Professor`. On the other hand, the Inverse model closure tests presented on average slightly better performances than the ones of `Professor`, while the test with real experimental data showed some differences from the other procedures, and raised some practical problems that may be investigated in the future.

Contents

Chapter 1

Machine learning introduction

Machine learning is a field of computer science which deals with the design of algorithms that are able to accomplish a task without being explicitly programmed; they extract the information about the task from some sort of **experience**. The main references for this chapter are [1] and [2].

1.1 Main classification

Machine learning problems can be classified in three main classes, according to the type of experience they exploit:

- **Supervised learning:** in supervised learning the experience is a dataset of examples. Each example is made of some **features**, i.e. some properties, and a **label**. The goal is to learn a function that labels new examples. In most cases, the examples are vectors $\mathbf{x} \in \mathbb{R}^n$ and the labels are vectors $\mathbf{y} \in \mathbb{R}^m$. Then, the aim is to map a previously unobserved input $\mathbf{x}$ to an output $\mathbf{y} = \mathbf{f}(\mathbf{x})$, without explicitly defining $\mathbf{f}(\cdot)$.

- **Unsupervised learning:** in contrast with supervised learning, in unsupervised learning the examples have no labels, just features. So, the dataset in most cases is just a set of observations $\mathbf{x}_i$, and the goal is to extract useful information from it. For example, the task may be the subdivision of the examples into clusters (**clustering**); another one may be to learn the probability distribution of the examples, and then using it to generate new examples (**generative** models).

- **Reinforcement learning:** in reinforcement learning an AI agent interacts with an **environment**, generating experience at run-time. This environment should give a **feedback** to the AI agent with a reward-penalty system. The goal is to make the agent learn by **trial and error** how to maximize the rewards and minimize the penalties. An example of reinforcement learning task may be the development of an AI agent that learns how to play a video game. In fact, many video games are environments with a straightforward built-in reward-penalty system. As an example, in [3] a reinforcement learning algorithm was designed to play seven Atari 2600 games.

This thesis focuses on supervised learning, which is presented in section 1.2.

1.2 Supervised learning

In supervised learning, a dataset with labelled examples is given, and the aim is to build a model that labels new examples. Usually, the examples are vectors in $\mathbb{R}^n$, where each component is one of their features, and the labels are also vectors in $\mathbb{R}^m$, but sometimes the structure of the data is more complicated. The word **supervised** reminds of a teacher that labels each example for his students, so that they learn how to label new ones.
In order to solve a supervised learning problem, one should recognize the type of **task** required (see section 1.2.1) and choose a **model** which is able to solve such task (see section 1.2.2). If the model is parametric, its parameters need to be estimated using the labelled examples (see section 1.2.3). The parameters estimation must be done with care, because the algorithm should behave well on new unlabelled examples, not on the labelled ones; it should **generalize** to new examples. This topic is debated in section 1.2.4. At the end of this chapter, a procedure for the **selection** of different models, or **fine tuning** of the model architecture is presented (see section 1.2.5).

1.2.1 Tasks

There are three main types of supervised learning tasks:

- **Regression:** in regression labels are numerical real valued quantities, or real-valued vectors.
- **Classification:** in classification the labels are categories, and the algorithm must classify the examples within these categories. An example of a classification task is **object recognition**, in which the examples are images and the categories are object identifiers. Usually, the categories are codified so that the labels are numerical quantities.
- **Structured output problems:** structured output problems involve labels with a complicated data structure, usually with many components, which are not categories nor numerical values. An example is **image captioning**, in which the examples are images and the labels are descriptions of the images in natural language.

There is not a rigid classification of supervised learning tasks, just like there is not a clear-cut subdivision between supervised, unsupervised and reinforcement learning. The tuning procedures presented in section 2.3 and in chapter 3, however, involve only regression tasks, so this thesis will focus on them.

1.2.2 Models

The prediction is accomplished by using a model that can adapt to the examples in the dataset. Usually, a model is a parametric function $\mathbf{f}(\mathbf{x}; \boldsymbol{\theta})$ whose parameters are chosen to make the model perform well, and the estimation of these parameters is done by processing the dataset (see section 1.2.3). In this subsection two different types of models are presented: **linear models**, employed in section 2.3, and **feedforward neural networks**, employed in chapter 3.

Linear models

Linear models are simple but powerful tools that can be applied easily to regression tasks (**linear regression**). A linear model maps input vectors $\mathbf{x} \in \mathbb{R}^n$ to output vectors $\mathbf{y} \in \mathbb{R}^m$ through an affine transformation

$$\mathbf{y} = \mathbf{K} \cdot \mathbf{x} + \mathbf{b} \tag{1.1}$$

where $\mathbf{K}$ is a $m \times n$ matrix and $\mathbf{b} \in \mathbb{R}^m$. It is convenient to add a unit feature to each of the examples in order to remove the bias $\mathbf{b}$ without loss of generality. Equation 1.1 represents a polynomial of order one, but linear regression could be used to model **non-linear** dependencies because it is possible to add more features to the input vectors by using combinations of features. For example, if $\mathbf{x} = (1, x_1, x_2)$, a quadratic function can be implemented by replacing $\mathbf{x}$ with a feature vector $\overline{\mathbf{x}} = (1, x_1, x_2, x_1 x_2, x_1^2, x_2^2)$. Polynomials of arbitrary order are implemented in a similar way, at the cost of increasing the number of parameters: if the original features are k, a generic polynomial of order n has

$$N_n^{(k)} = 1 + \sum_{i=1}^{n} \frac{1}{i!} \prod_{j=0}^{i-1} (k + j) \tag{1.2}$$

parameters [4]. Transformations of output vectors are also useful: for example, replacing $\mathbf{y}$ with $\overline{\mathbf{y}} = \ln \mathbf{y}$ (computed element-wise) introduces an exponential dependence between inputs and outputs. In order to use the linear regression methods described in subsection 1.2.3, the model should be **linear in the parameters**, not in the original features.

Feedforward neural networks

Feedforward neural networks are the simplest type of artificial neural networks. As the name suggests, they propagate the information within the network in just one direction, from the input to the output. This section presents the simplest ones, called **multilayer perceptrons (MLPs)** or fully-connected neural networks. They are parametric models made of a composition of many affine maps alternated with non-parametric non-linear functions called **activation functions**. Let $\mathbf{x} \in \mathbb{R}^n$ be the input vector; a generic affine map $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^m$ consists on a matrix multiplication followed by a translation:

$$\mathbf{f}(\mathbf{x}) = \mathbf{K} \cdot \mathbf{x} + \mathbf{b} \tag{1.3}$$

where $\mathbf{b} \in \mathbb{R}^m$ is called **bias**, and $\mathbf{K}$ is a $m \times n$ matrix called **kernel**. A neural network is a composition of many such affine transformations interleaved with some non-parametric non-linear activation functions $\mathbf{a}(\cdot)$. Some examples of activation functions are

$$\begin{aligned} \text{Tanh} &: \frac{e^{x_i} - e^{-x_i}}{e^{x_i} + e^{-x_i}} \\ \text{Sigmoid} &: \frac{1}{1 + e^{-x_i}} \\ \text{ReLU} &: \frac{|x_i| + x_i}{2} \\ \text{Softmax} &: \frac{e^{x_i}}{\sum_j e^{x_j}} \end{aligned} \tag{1.4}$$

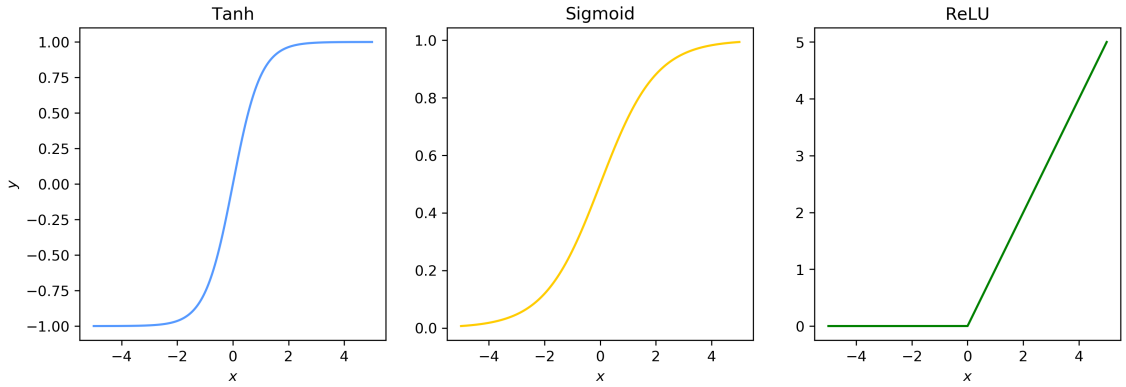


Figure 1.1: Some activation functions.

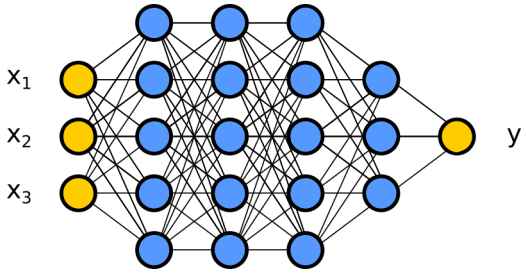


Figure 1.2: A graph of a neural network with input dimension 3, output dimension 1, and hidden layer architecture [5, 5, 5, 3]. The edge from the i-th node of layer m to the j-th one of layer $m+1$ represents the value at column i and at row j of the kernel matrix of layer $m+1$.

and are plotted, whether possible, in figure 1.1. The combination of an affine transformation with an activation function is called **layer**:

$$\mathbf{h} = \mathbf{a}(\mathbf{K} \cdot \mathbf{x} + \mathbf{b}) \tag{1.5}$$

Each layer is stacked upon the previous one in such a way that dimensions match: the activations of each layer should be in the domain of the affine map of the next layer. The first layer is applied to the input $\mathbf{x}$ (**input layer**), the last one represents the **output layer** and should have an activation function compatible with the task (usually linear for a regression task, softmax for a classification task). Layers that are not input nor output layers are called **hidden layers**; a neural network with N hidden layers is defined recursively by

$$\mathbf{h}_{i+1} = \mathbf{a}_{i+1}(\mathbf{K}_{i+1} \cdot \mathbf{h}_i + \mathbf{b}_{i+1}) \qquad i = 0, ..., N \tag{1.6}$$

where $\mathbf{h}_0 = \mathbf{x}$ and $\mathbf{h}_{N+1} = \mathbf{y}$. Neural networks are usually represented with a graph (see figure 1.2), where layers are vertical sequences of nodes. Each node is a component of that layer, and each edge is a component of the kernel matrices. Usually, bias vectors are not drawn.

Multilayer perceptrons are **universal approximators** [5]. In fact, consider a neural network with a single hidden layer of size j, with input dimension n and output dimension 1:

$$y = \sum_{i=1}^{j} \lambda_i a_i(\mathbf{K} \cdot \mathbf{x} + \mathbf{b}) \tag{1.7}$$

Let denote with Υ_j the set of all neural networks like 1.7, for all $\mathbf{K} \in \mathbb{R}^{n\times j}$ and $\mathbf{b}, \boldsymbol{\lambda} \in \mathbb{R}^j$. Let $\Upsilon = \bigcup_{j\in\mathbb{N}} \Upsilon_j$. Let μ be an arbitrary **finite** measure on $\mathbb{R}^k$, and define $L^p_\mu(\mathbb{R}^k)$ the space of function defined on $\mathbb{R}^k$ such that

$$\|f\|_{p,\mu,\mathbb{R}^k} = \left(\int_{\mathbb{R}^k} |f(x)|^p d\mu(x)\right)^{\frac{1}{p}} < \infty \tag{1.8}$$

A subset $S \subset L^p_\mu(\mathbb{R}^k)$ is dense in $L^p_\mu(\mathbb{R}^k)$ if, for an arbitrary $f \in L^p_\mu(\mathbb{R}^k)$ and $\epsilon > 0$ there is a function $g \in S$ such that $\|f - g\|_{p,\mu,\mathbb{R}^k} < \epsilon$. Then, the following theorem holds:

Theorem 1.2.1 (Universal Approximation [5]) *If $\mathbf{a}(\cdot)$ is bounded and nonconstant, then Υ is dense in $L^p_\mu(\mathbb{R}^k)$ for all finite measure μ on $\mathbb{R}^k$.*

This theorem is useful in a context where the input is assumed to be a random variable, so the quality of the approximation can be averaged over the input space, weighted by the input finite measure μ. If instead, the approximation should be simultaneously good on all inputs $\mathbf{x}$, stronger hypotheses are required. Let $C(X)$ be the space of continuous functions defined on $X \subset \mathbb{R}^k$. Then, the following theorem holds:

Theorem 1.2.2 (Universal Approximation [5]) *If $\mathbf{a}(\cdot)$ is continuous, bounded and nonconstant, then for each compact subset $X \subset \mathbb{R}^k$, given any function $f \in C(X)$ and $\epsilon > 0$, there exists a neural network $F \in \Upsilon$ such that $|f(\mathbf{x}) - F(\mathbf{x})| < \epsilon$, for all $\mathbf{x} \in X$.*

These theorems involve shallow networks, where the expressive power grows with the width; a similar result holds also for width-bounded deep neural networks with ReLU activation functions (which are unbounded, and so excluded from the previous theorems[1]):

Theorem 1.2.3 (Universal Approximation [7]) *For any Lebesgue-integrable function $f : \mathbb{R}^n \to \mathbb{R}$ and any $\epsilon > 0$, there exists a fully connected ReLU network with width lower or equal than $n + 4$ for each hidden layer, such that the function $F : \mathbb{R}^n \to \mathbb{R}$ represented by this network satisfies*

$$\int_{\mathbb{R}^n} |f(\mathbf{x}) - F(\mathbf{x})| d\mathbf{x} < \epsilon \tag{1.9}$$

i.e. the set of all fully connected ReLU networks with width lower or equal than $n+4$ for each hidden layer is dense in $L^1(\mathbb{R}^n)$ with standard Lebesgue measure.

In practice, one cannot use a neural network with infinite parameters, but these theorems give a theoretical foundation that, at least in principle, neural networks can learn highly non-linear functions. The hierarchical structure of a neural network, based on a composition of many simple functions, enables it to learn by extracting multiple hierarchical-structured features from the data: the higher-level features are transformations of the lower ones. This type of feature learning is called **deep learning**. Multilayer perceptrons are the simplest type of neural networks, but more sophisticated variations have been designed in order to exploit different data structures (e.g. convolutional neural networks for translational invariant data or recurrent neural networks for temporal sequences).

[1] However, the universal approximation property was proved also for a wider class of activation functions, which includes the ReLU [6].

1.2.3 Training

The parameters of parametric models such as neural networks need to be estimated from the dataset. This process is called **training**. The standard procedure consists in defining a measurement of the performance of the model (with the "lower is better" format), called **loss** or **cost** function, and minimizing it.

Loss function

There are many choices for the loss function, and choosing the best one is problem-dependent. Let $\mathbf{x}_i$, $i = 1, ..., N$ be the dataset examples, and $\mathbf{y}_i$ the corresponding labels. In a regression problem, a standard choice could be the mean squared error, or the mean absolute error:

$$\begin{aligned} \text{Mean squared error} &: \frac{1}{N}\sum_{i=1}^{N} \|\mathbf{y}_i - \mathbf{f}(\mathbf{x}_i)\|_2^2 \\ \text{Mean absolute error} &: \frac{1}{N}\sum_{i=1}^{N} \|\mathbf{y}_i - \mathbf{f}(\mathbf{x}_i)\|_1 \end{aligned} \tag{1.10}$$

where $\|\cdot\|_1$ and $\|\cdot\|_2$ are the L^1 and L^2 norm, respectively. While the idea of minimizing these losses is intuitively, some theoretical insights are interesting. Let $\mathbf{x}_0$ be a generic example. If the prediction $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}_0)$ of the model is considered as a conditional probability distribution $P(\mathbf{y}|\mathbf{x};\boldsymbol{\theta})$, then a maximum likelihood estimator could be used to estimate the parameters:

$$\boldsymbol{\theta}^\star = \text{argmax}_{\boldsymbol{\theta}} P(\mathbf{Y}|\mathbf{X};\boldsymbol{\theta}) \tag{1.11}$$

where $\mathbf{X}$ and $\mathbf{Y}$ are the whole dataset of examples. If the examples are independent and identically distributed, then the probability factorizes over each example. Taking the logarithm transforms products into sums:

$$\boldsymbol{\theta}^\star = \text{argmax}_{\boldsymbol{\theta}} \ln\left(\prod_{i=1}^{N} P(\mathbf{y}_i|\mathbf{x}_i;\boldsymbol{\theta})\right) = \text{argmax}_{\boldsymbol{\theta}} \sum_{i=1}^{N} \ln P(\mathbf{y}_i|\mathbf{x}_i;\boldsymbol{\theta}) \tag{1.12}$$

If $P(\mathbf{y}|\mathbf{x};\boldsymbol{\theta})$ is chosen to be a multivariate Gaussian distribution, then the **maximum likelihood estimator** for the model parameters turns out to be the minimum of the mean squared error, while if is considered as a Laplace distribution, it turns out to be the minimum of the mean absolute error. Another interesting insight [2] is that the best prediction $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}_0)$ is the conditional mean $\text{E}(\mathbf{y}|\mathbf{x} = \mathbf{x}_0)$ when using the mean squared error, while it is the conditional median when using the mean absolute error.

Minimization

The minimization step consists in minimizing the loss function calculated on the examples in the dataset. The minimization strategies differ from model to model: sometimes a closed-form solution to jump to the global minimum is available, but in general iterative algorithms are required.

Linear regression In linear regression minimizing the mean squared error brings to a closed-form solution. Consider a linear regression with a one-dimensional output, let $\mathbf{X}$ be a matrix where each row is an example from the dataset and let $\mathbf{y}$ be the corresponding labels. If the intercept is incorporated in the parameter vector $\boldsymbol{\theta}$, the mean squared error is proportional to

$$\overline{L} = (\mathbf{y} - \mathbf{X} \cdot \boldsymbol{\theta})^T (\mathbf{y} - \mathbf{X} \cdot \boldsymbol{\theta}) \tag{1.13}$$

which is a quadratic function of the parameters. Differentiating with respect to $\boldsymbol{\theta}$ and setting the first derivative to zero leads to a closed-form solution called *least squares*:

$$\boldsymbol{\theta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \tag{1.14}$$

at least if $\mathbf{X}^T \mathbf{X}$ is invertible. For a multi-output regression, a similar result holds:

$$\mathbf{K} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{Y} \tag{1.15}$$

where $\mathbf{K}$ is the matrix of parameters and $\mathbf{Y}$ a matrix where each row is the label of an example from the dataset. If $\mathbf{X}^T \mathbf{X}$ is singular, the **Moore-Penrose pseudoinverse** $\mathbf{X}^+$ can still solve the problem. The Moore-Penrose pseudoinverse of a matrix $\mathbf{A}$ is defined as

$$\mathbf{A}^+ = \lim_{\alpha \to 0^+} (\mathbf{A}^T \mathbf{A} + \alpha \mathbf{I})^{-1} \mathbf{A}^T \tag{1.16}$$

Usually, practical implementations do not use its definition, but exploit the singular value decomposition of $\mathbf{A}$. The Moore-Penrose pseudoinverse is useful for solving linear equations. In a single-output linear regression, the predictions of all examples of a dataset $\mathbf{X}$ can be written as

$$\mathbf{y} = \mathbf{X} \cdot \boldsymbol{\theta} \tag{1.17}$$

which is a system of linear equations. Depending on the particular problem, there could be many solutions, a unique solution, or none. The pseudoinverse will always propose a solution:

$$\boldsymbol{\theta} = \mathbf{X}^+ \mathbf{y} \tag{1.18}$$

If many solutions exist, the pseudoinverse will return the one with minimum Euclidean norm $\|\boldsymbol{\theta}\|_2^2$; if a unique solution exists, the pseudoinverse $\mathbf{X}^+$ is equal to the inverse $\mathbf{X}^{-1}$ and will return the exact solution. If no solution exists, the pseudoinverse will return the solution that minimizes the mean squared error $\|\mathbf{y} - \mathbf{X} \cdot \boldsymbol{\theta}\|_2^2$, and the minimization problem is solved. By the way, if there is at least one solution, this means that a function that maps perfectly each input example to its label is found. This corresponds to a mean squared error of zero, which is of course a global minimum. So, the pseudoinverse will always return a minimum of the loss function. This solution translates easily to the multi-output regression:

$$\mathbf{K} = \mathbf{X}^+ \mathbf{Y} \tag{1.19}$$

Of course, if $(\mathbf{X}^T \mathbf{X})^{-1}$ exists, then $\mathbf{X}^+ = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T$.
If another loss function is used, other methods are required. In particular, calculating the gradient of the loss function with respect to the parameters is straightforward, so the minimization can be performed by using the same methods of feedforward neural networks, which are described in the next paragraph.

Neural networks A closed-form solution for the training of neural networks does not exist. However, neural networks offer the possibility to calculate the gradient of the loss function with respect to each of the model parameters, by using an efficient algorithm called **back-propagation**. So, many algorithms for training neural networks are gradient-based, and are more sophisticated versions of the **stochastic gradient descent** algorithm. These algorithms require some settings like the number of iterations over the whole dataset (**epochs**), the number of samples processed at each parameters update (**batch size**), or the size of each parameters update (**learning rate**). These settings can influence the performance of the model, and so must be tuned with care. An in-depth description of these algorithms is presented in appendix A, while a procedure for tuning these parameters is presented in subsection 1.2.5.

Not all models require a training procedure: for example the k-**nearest neighbours** algorithm simply learns by heart the dataset. Let $\mathbf{x}$ be a new example, and suppose a distance function is defined on the space of all examples. Then, the k-nearest neighbours searches for the k examples in the dataset which are nearest $\mathbf{x}$, and labels $\mathbf{x}$ with the average of their labels. Of course k is still a parameter that can affect the performance of the model, but it does not fit into a training procedure. In fact, the value of k that minimizes the loss will always be 1, which is not the best one, at least in general. The determination of parameters like k is debated in subsection 1.2.5.

1.2.4 Regularization

The main point in machine learning is not to learn the training set by heart (this is achieved successfully by a 1-nearest neighbours algorithm) but to generalize well on previously unobserved examples. This means that evaluating the performance of a model on the training set is useful only in the training phase, but not for the quality estimation of the model. The standard procedure in machine learning consists in dividing the dataset in a **training set** and in a **validation set**: the training set is used for the optimization of the model parameters, the validation set is used to evaluate the performance of the model. The validation loss can be analysed with the **bias-variance** decomposition. Consider a regression task in which the examples are generated from an underlying function plus an additive Gaussian noise $y = f(\mathbf{x}) + \epsilon$. Let $\hat{f}(\mathbf{x})$ be a model trained for this task, and use the mean squared error as loss function. Then, the expected prediction error for a new example $\mathbf{x}$ can be written as:

$$\begin{aligned}
&\mathrm{E}\left[\left(Y - \hat{f}(\mathbf{x})\right)^2 |\mathbf{x} = \mathbf{x}_0\right] = \mathrm{E}\left[\left(f(\mathbf{x}) + \epsilon - \hat{f}(\mathbf{x})\right)^2 |\mathbf{x} = \mathbf{x}_o\right] \\
&= \mathrm{E}\left[f(\mathbf{x})^2 + \epsilon^2 + \hat{f}(\mathbf{x})^2 + 2\epsilon f(\mathbf{x}) - 2f(\mathbf{x})\hat{f}(\mathbf{x}) - 2\epsilon\hat{f}(\mathbf{x})|\mathbf{x} = \mathbf{x}_o\right] \\
&= f(\mathbf{x})^2 + \sigma_\epsilon^2 + \mathrm{E}[\hat{f}(x)^2] - 2f(x)\mathrm{E}[\hat{f}(x)] \\
&= \sigma_\epsilon^2 + f(\mathbf{x})^2 + \mathrm{E}[\hat{f}(x)]^2 - 2f(x)\mathrm{E}[\hat{f}(x)] + \mathrm{E}[\hat{f}(x)^2] - \mathrm{E}[\hat{f}(x)]^2 \\
&= \sigma_\epsilon^2 + \left(f(\mathbf{x}) - \mathrm{E}[\hat{f}(x)]\right)^2 + \mathrm{E}[\hat{f}(x)^2] - \mathrm{E}[\hat{f}(x)]^2 \\
&= \mathrm{Noise} + (\mathrm{Bias})^2 + \mathrm{Variance}
\end{aligned} \tag{1.20}$$

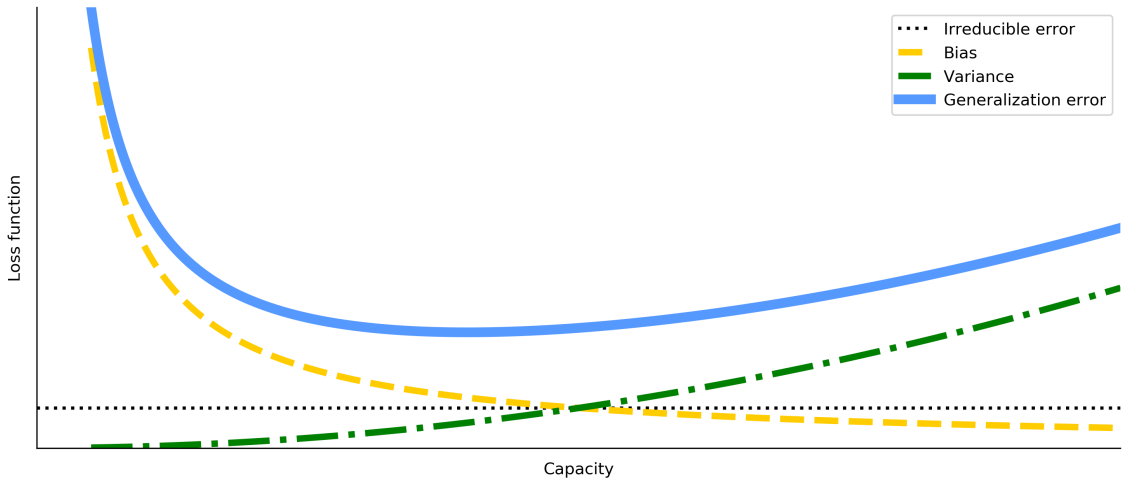


Figure 1.3: Qualitative dependence of the validation loss on capacity.

The first term is the noise originated from the data-generating process, the second is the bias of the prediction and the last one is its variance. The first one is not removable if the new example is independent from the dataset, while the other two depend on the model. In particular, they are influenced by the **capacity** of the model, which is a central concept in machine learning. Informally, the capacity of a model is a measurement of the ability to represent many different functions, i.e. the expressive power of a model. For example, the capacity of a polynomial is an increasing function of its order. If the capacity of a model is changed, training error and validation error respond differently. Usually, the training error is a decreasing function of the model capacity, because the model can represent more functions. The validation error behaves differently:

- If the model capacity is too low, it cannot represent well the underlying data-generating function (**high bias**). This leads to high training error and high validation error. This situation is called **underfitting**.
- If the model capacity is too high, it can represent well the underlying data-generating function, but it can also represent the statistical noise in the training set, or it can make a nearly perfect interpolation through the training set examples but behaves bad between them (**high variance**). This situation is called **overfitting**, and leads to low training error and high validation error.
- If the model capacity is comparable to the underlying data-generating process, the model can represent well the training set without overfitting, and the validation error is minimized.

The qualitative dependence of the validation loss on capacity is plotted in figure 1.3, while some examples of underfitting, overfitting and optimal capacity are illustrated in figure 1.4. Low capacity models usually have low variance but high bias, and viceversa. The best model is then obtained with a trade-off between bias and variance. [1] defines **regularization** as "any modification made to a learning algorithm that is intended to reduce its generalization error but not its training error", where the generalization error is the prediction error on new examples.

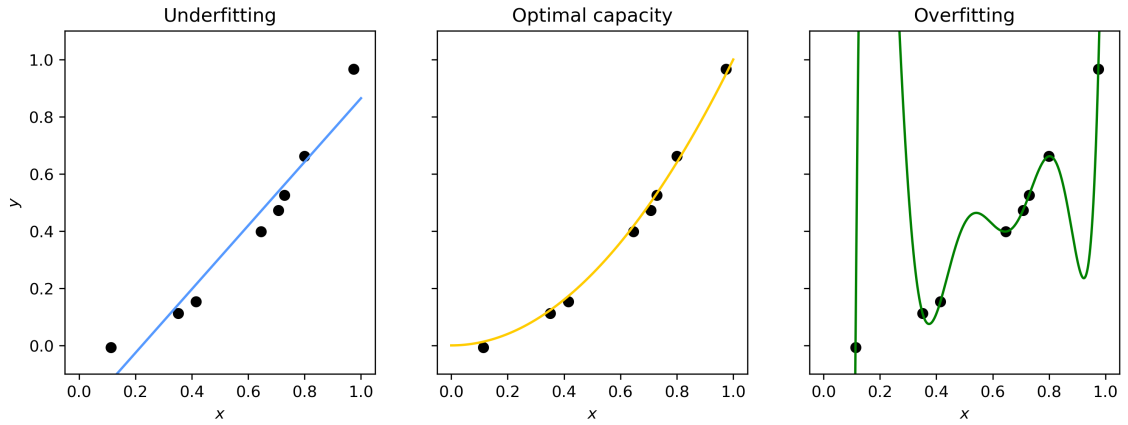


Figure 1.4: In this figure the points are generated from a quadratic function with some Gaussian noise. A linear function (sx) has lower capacity than required, so it underfits. A high-order polynomial (dx) has too much capacity and overfits badly. In the central plot there is a second-order polynomial, which has the optimal capacity by construction. Fits have been performed with the `SciPy` [8] package.

Regularizers

Equation 1.20 shows that an unbiased model is not necessarily the best model. Consider a linear regression task where the underlying data-generating function is linear. The least squares model is then unbiased, but the contribution of variance to the prediction error is still present. By reducing the capacity of the model, a **bias-variance trade-off** mechanism could eventually decrease the variance more then the corresponding increase in bias, leading to a better performing model. The capacity of a model can be limited by adding a parameter norm penalty to the loss function, for example a L^2 norm penalty:

$$\overline{L}(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) + \lambda \left\|\boldsymbol{\theta}\right\|_2^2 \tag{1.21}$$

This method is called **weight decay**, **ridge regression** or **Tikhonov regularization**. The weight decay can be applied to both linear models and neural networks. In linear models, there still exists a closed-form solution similar to least squares:

$$\boldsymbol{\theta} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y} \tag{1.22}$$

Notice that in the limit $\lambda \to 0$ the pseudoinverse method is recovered. Another example of parameter norm penalty is the L^1 norm:

$$\overline{L}(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) + \lambda \left\|\boldsymbol{\theta}\right\|_1 \tag{1.23}$$

also known as **Lasso** (Least Absolute Shrinkage and Selection Operator) when applied to linear models with mean squared error loss. In this case, a closed-form solution does not exist anymore. Usually the intercept (or the biases for a neural network) is excluded from the norm penalty. In subsection 1.2.3 it was shown that minimizing the mean squared error or the mean absolute error could be interpreted as maximizing the conditional log-likelihood assuming the prediction was centered on a Gaussian or a Laplace distribution, respectively. Introducing a norm penalty, instead, is equivalent to impose a **prior** distribution on the weights and, using the

same likelihood as before, estimate the parameters with a **Maximum a Posteriori** Bayesian inference. The prior is a Gaussian for the L^2 norm, and a Laplace distribution for the L^1 norm.

Other methods

Other regularization methods exist in addition to parameter norm penalties, i.e. noise injection, model averaging, dropout [9], adversarial training [10, 11], etc. If iterative gradient-based training algorithms are used, another simple regularization technique to prevent overfitting is **early stopping**: during each iteration over the whole dataset, the algorithm evaluates the validation loss, and stops when the validation loss stops decreasing. When using early stopping, the model generalization error should be evaluated on an independent **test set**, because the validation set is used by the training algorithm, and so the validation loss could underestimate the true generalization error.

1.2.5 Hyperparameter tuning

The architecture of a machine learning model, the choice of the model itself and the details of the training procedure (e.g. optimization algorithms, loss functions, regularizers, epochs or batch sizes) can be seen as parameters as well as the biases and the kernels of the network layers, but they do not fit into the training procedure. They are called **hyperparameters**. Hyperparameter tuning consists in finding the hyperparameter configuration that minimizes the validation error. As noted in [12], hyperparameter tuning is a big concern in machine learning research, a "direct impediment to scientific progress". That is because it can influence heavily the performance of an algorithm so, when testing different strategies, it is difficult to understand if a new algorithm performs better than the previous one because it is intrinsically better or because a lucky hyperparameter configuration is used. Vice versa, if a new algorithm performs badly, it is hard to find out if it is because the new ideas are not good or because the hyperparameters were tuned badly.

The main problem with hyperparameter tuning is the **high computational cost** of evaluating the validation error of a single point in the hyperparameter space, which consists in training a new machine learning model. Moreover, the configuration space often is **tree-structured**, in the sense that some variables are defined only if some other variables take on a particular value (for example, the dimension of the third layer of a neural network is defined only if the number of layers is ≥ 3). Some common strategies are manual tuning, grid search over the configuration space, random search or more sophisticated algorithms like Sequential Model-Based Global Optimization algorithms (see appendix B for details and section 3.4.1 for a practical implementation).

After hyperparameter tuning, the performance of the best configuration should be evaluated on an independent dataset, called **test set**, because the tuning procedure may overlearn the validation set.

Chapter 2

Tuning event generators

Event generators are programs that simulate the collision of particles at high energies. They are made of different components, from theoretical predictions (e.g. cross sections of hard processes etc.) to phenomenological models (e.g. underlying event model, hadronization model). Event generators, especially their phenomenological components, introduce some parameters that are difficult to obtain on a theoretical basis, so they must be estimated by comparison with the experimental data. This procedure is called **tuning**. This chapter contains a brief introduction to event generators[1], an overview of the tuning problem and some simple methods to solve it, and finally a presentation of the current state-of-the-art tuning procedure, called `Professor` [4].

2.1 Event generators

Event generators are tools for the simulation of collisions of particles at high energies, which give a detailed description of the final state, in contrast with QCD computations of inclusive quantities (e.g. results summed over all final states). In a generator, the collision steps are simulated as follows:

1. The user defines the particle type of the incoming beams and the collision center of mass energy.

2. One parton for each beam is selected as the initiator of the **initial state shower**: it begins a series of splittings (e.g. $g \to q\overline{q}$) until the shower algorithm reaches a region where it is not applicable anymore and needs to be stopped. More details about showers will be presented in the next paragraph.

3. One parton for each initial state shower participates at the **hard process**, which is the core of the collision; a certain number of particles are produced.

4. Each of these particles starts a **final state shower**, i.e. it starts a series of splittings until the shower stops.

5. Secondary interactions between the partons not involved in the hard process may occur.

[1] The main references for that section are [13] and [14]

6. Quarks and gluons fragment to hadrons, to ensure confinement of coloured particles.

7. Some hadrons generated by the hadronization step may be unstable, so their decay must be taken into account.

More precisely, the program execution does not follow this "chronological" order: for example, it starts from the hard process, so the initial state shower is generated backwards. Ignoring these details, from the list above it is apparent that the main components of an event generator are

- A library of parton distribution functions and processes with their cross sections, computed at fixed order. Processes can be activated or deactivated by the user, depending on the analysis he wants to perform.

- A shower algorithm: it adds the contribution of soft and collinear QCD emissions of coloured particles, which can be sizable. In QCD calculations of inclusive quantities, the effects of soft and collinear singularities from real and virtual emissions cancel out; however, event generators aim at describing also more exclusive final states. Two different showers are implemented: one for initial state radiation and one for final state radiation.

- A hadronization model that transforms quarks and gluons in hadrons, in order to ensure confinement of coloured particles.

- An underlying event model for the treatment of the hadron remnants that were not involved in the hard process, including the possibility of secondary interactions.

- A library for the decay of unstable particles.

This brief presentation is focused on QCD aspects, but QED and EW interactions may also be included in the simulation framework. Event generators rely on parameters that must be carefully tuned in order to give realistic results. These parameters are mostly introduced by phenomenological models e.g. the hadronization and the underlying event model. In the remaining of this section, some examples of tunable parameters are given. Some of them will be tuned in chapter 4 with the generator `PYTHIA8` [15].

Parton shower The shower algorithm adds a number of quark or gluon emissions associated to the soft and collinear singularities to the hard process computed at fixed order. The inclusion of these emissions can give rise to contributions of order $O(1)$ to the kinematical distributions. This shower algorithm is implemented as a sequence of partonic branchings, where each branching is a decay of a parton ($q \to qg$, $g \to gg$ or $g \to q\overline{q}$). For final-state emissions, starting from the partons outgoing from the hard process, the parton shower is then a tree-structured probabilistic model where each edge is a parton and each vertex is the decay of a parton. Each splitting is characterized by an ordering variable t which is maximum for the parton that initiates the shower, is zero in the collinear limit, and is strictly decreasing along the tree. The shower algorithm is sketched in this way, at least for final state showers:

1. After the hard process, set the ordering variable t of the initiator parton to the typical scale of the process.

2. Sample the variable $0 < t' < t$ according to a theoretically based probability distribution. If $t < t_0$, stop the shower.

3. Split the parton into two other partons with ordering variable t', each with a portion of the momentum, and reconstruct their directions. The type of decay and the momentum subdivison are sampled from theoretically based probability distributions.

4. Restart the shower for each of the two new partons, with $t = t'$.

An illustration of the development of a final state shower is presented in figure 2.1. At some point during the development of the shower, the algorithm reaches a region where it is not applicable anymore and needs to be stopped. So, a cut-off parameter t_0 is introduced in step 2 as a break condition. `PYTHIA8` implements the shower with p_T as ordering variable, and the lower cut-off parameter is labelled[2] `TimeShower:pTmin`. Moreover, it regularizes the $p_T \to 0$ emission divergence with a damping factor $p_T^2/(p_{T0}^2+p_T^2)$ and the use of an $\alpha_S(p_{T0}^2+p_T^2)$, where p_{T0} at the reference center-of-mass energy is fixed by the tunable parameter `TimeShower:pT0Ref`. Another parameter of the shower is the α_S value, whose value at scale M_Z^2 is tunable with the parameter `TimeShower:alphaSvalue`. Analogously, the same parameters are present in the `SpaceShower`: the `TimeShower` takes care of the final state radiation, while the `SpaceShower` takes care of the initial state radiation.

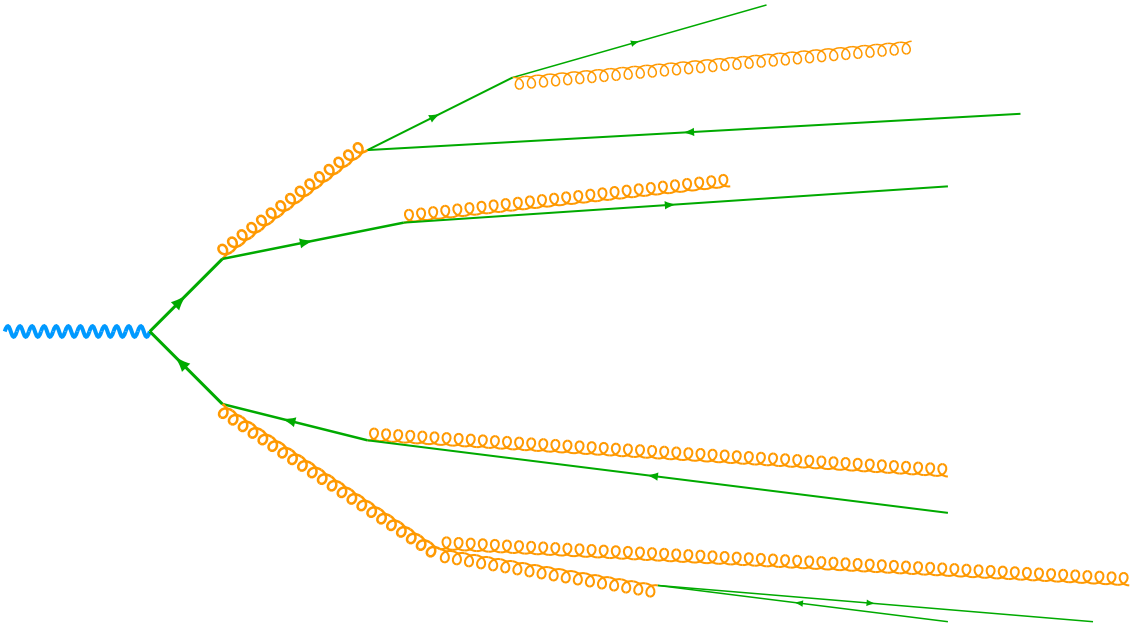


Figure 2.1: A qualitative illustration of a final state parton shower. Made with [16].

[2]This label and the following ones are used to set the value of the corresponding parameters, and are usually written in a configuration file that the generator reads at run-time. For an example of configuration file for `PYTHIA8`, see subsection 2.3.1

Underlying event model The **beam remnants** are what remains of the two colliding particles after the shower initiators are taken out of them. The correct treatment of the beam remnants requires taking into account multiparticle interactions, i.e. the possibility that particles left out from the main process enter a relatively-hard process. As previously said, this is implemented through phenomenological models with introduction of parameters: for example, it is assumed that the partons within the hadrons present an **intrinsic transverse momentum**, sampled from a given probability distribution. Then, the secondary interaction is implemented with a **cut-off** in the transverse momentum, because the parton cross section diverges for $p_T \to 0$. The parameters of the intrinsic p_T probability distribution and the p_T cut-off for secondary interactions are tunable parameters.

In `PYTHIA8` the intrinsic k_T is sampled from a Gaussian distribution in p_x and p_y, with a σ value that depends on many parameters. At the hard-interaction limit, it is determined by the `BeamRemnants:primordialKThard` parameter. The multiparton interaction divergence at $p_T \to 0$ is managed similarly to the showers, with a sharp cut-off `MultipartonInteraction:pTmin` and with a smooth damping factor, related to the parameter `MultipartonInteraction:pT0Ref`. This last parameter, however, is related to many other choices, so it has not an independent meaning.

2.1.1 Analysis of events

Running a generator produces an event record, which contains the full history of all events of the simulation. These event records are huge files with highly redundant information, and they must be analysed in order to obtain some theoretical predictions about some observables. The `Rivet` [17] toolkit is both an analysis system for Monte Carlo events and a library of built-in analyses, with the actual experimental measurements conveniently attached. It is generator-independent: all event records in the event record format `HepMC` [18] can be analysed.

2.2 Overview of the tuning problem

At first, a set of experimental measurements is selected. In practice, these measurements are a set of histograms. The generator is configured with the same configuration of the experiment (center-of-mass energy, incoming beams, etc.), and with the corresponding processes activated. Then, the generator is configured with a set $\mathbf{p}$ of parameters. Running the generator produces an event record, which will be analysed by an analysis system. An analysis that corresponds to the experimental measurements must be used. The analysis system will conveniently use the same bins of the experimental measurements, so the histograms are directly comparable. In the end, a set of bins $\mathbf{h}(\mathbf{p})$ directly comparable with the experimental values $\mathbf{h}_{exp}$ is obtained. The goal is to find the set of parameters $\mathbf{p}_{tune}$ that makes the generator reproduce the experimental data. An example with two Monte Carlo runs is presented in figure 2.2. This problem can be classified as an **optimization** problem: after the definition of a goodness of fit function with the "lower is better" format (a loss function $L(\mathbf{p})$), the tuning problem is reduced to a minimization problem

$$\mathbf{p}_{tune} = \text{argmin}_{\mathbf{p}} L(\mathbf{p}) \tag{2.1}$$

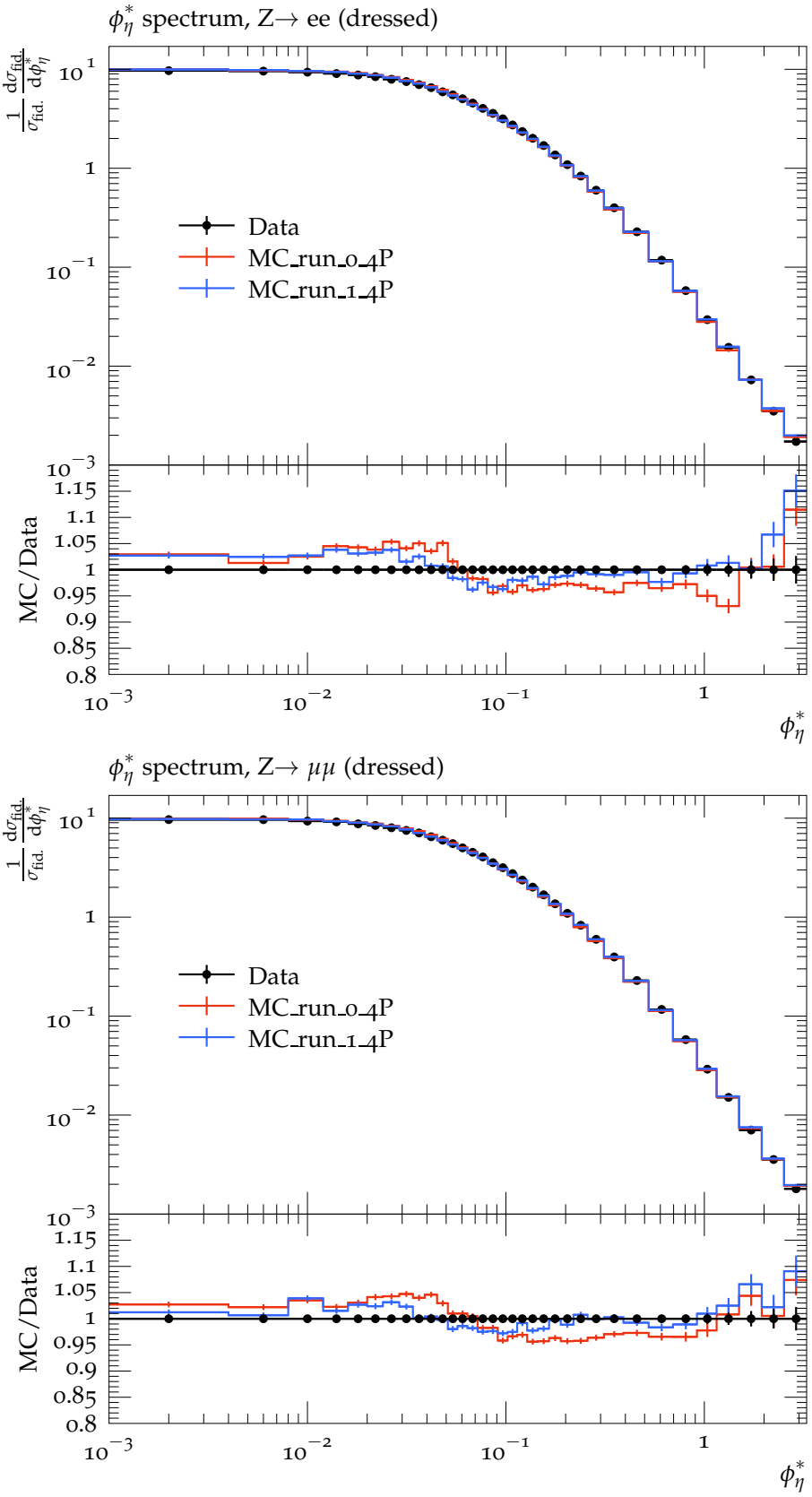


Figure 2.2: Examples of two Monte Carlo runs with different parameters against the experimental data, for two different histograms. The goal is to adjust the parameters in order to make the theoretical prediction overlap the experimental data. A description of these measurements is given in subsection 4.2.

though a hard one, due to the **high computational cost** of each evaluation.

2.2.1 Tuning methods

This minimization problem has many issues:

- Each evaluation has a high computational cost.
- The theoretical predictions are **uncertain**, and reducing the uncertainty requires more computational resources. So, a compromise between the number of evaluations and the signal-to-noise ratio of the results must be found.
- There is no access to information about derivatives, so gradient-based algorithms are not applicable.
- Some components of the generator are phenomenological models which are not theoretically exact, so a perfect tune for all processes and observables may not exist.

Some straightforward tuning methods [4] are **manual tuning** and **brute-force tuning**, but these methods have many drawbacks. Manual tuning requires an operator with a deep understanding of the generator details, and it does not scale well with the number of parameters. On the other hand, brute-force scansion of the parameter space, for example with a grid search or a random search, is also problematic with a high number of parameters, because the computational cost goes up exponentially. At present, the best methods are **parametrisation-based**. They consists in parametrising the generator behaviour with a parametric model, in order to replace the real loss function $L(\mathbf{p})$ with a surrogate loss function $\overline{L}(\mathbf{p})$ which is calculated with the model, not with the actual generator. If the model is chosen to be easy to minimize, but also capable of representing the generator behaviour at least approximately, then the final tune could be approximated by

$$\mathbf{p}_{tune} \approx \overline{\mathbf{p}}_{tune} = \text{argmin}_{\mathbf{p}} \overline{L}(\mathbf{p}) \tag{2.2}$$

This can be seen as a supervised learning problem: the dataset is built by running the generator many times with different parameters in some variation ranges; the model is then trained on this dataset, and then it can be interfaced with an optimization algorithm. This is the strategy adopted by `Professor` (see section 2.3), the current state-of-the-art tuning procedure, and by one of the strategies presented in chapter 3. This supervised learning problem has an interesting property, which is the fact that the examples can be generated at will by a computer, so the algorithm itself could generate new examples with some parameters and see the response.

2.3 `Professor`

The current state-of-the-art tuning procedure is the one implemented in the `Professor` [4] package, which is based on the ideas described in [19]. `Professor` parametrises the generator response to parameter variations using polynomials, then numerically optimizes a goodness of fit measurement between the parametrised behaviour of the generator and the experimental data. This section presents the basic work cycle for tuning a generator with `Professor`, with reference version `2.2.2`.

2.3.1 Sampling the parameter space

The first step is to evaluate the generator behaviour on many points in the parameter space, in order to build a dataset of Monte Carlo runs that will be used for the polynomial interpolation. At first, a template of the configuration file for the generator must be prepared, with all settings but the steerable parameters fixed. The steerable parameters must be set with the placeholder {parameter_name}, while the variation ranges of these parameters must be defined in a separate file. For example, the template for `PYTHIA8` could be something like this:

```
Main:numberOfEvents = 1000000
...
WeakSingleBoson:ffbar2gmZ = on
...
Tune:pp = 5
...
BeamRemnants:primordialKThard = {intrinsicKT}
SpaceShower:alphaSvalue = {asMZisr}
SpaceShower:pT0Ref = {pT0isr}
MultipartonInteractions:pT0Ref = {mpipt0}
```

while the parameter ranges file should be in this format:

```
#PARAMETER MIN MAX
intrinsicKT 1 2.5
asMZisr 0.12 0.14
pT0isr 0.5 2.5
mpipt0 1.9 2.2
```

Then, the script `prof2-sample` will read these files, sample N parameter configurations in the parameter space and make N numbered directories. Each of these directories will contain a file with some parameter values and a configuration file for the generator with those parameter values. Then, the generator can be launched in each of these folders, usually interfaced with an analysis system. After that, the generator output will be available for all N parameter configurations. This is the most computationally expensive step in the tuning work cycle.

The script `prof2-envelopes` can be used to plot the envelopes of the Monte Carlo runs and compare them to the experimental data. If the sampled configurations are not able to cover the experimental data, the tuning procedure must extrapolate the behaviour outside the variation ranges of the parameters, and the final tune could be unreliable.

2.3.2 Parametrisation

The parametrisation of the generator response is done **independently** for each bin by means of a polynomial fit. A polynomial is linear in its coefficients, so the parametrisation consists in a linear regression for each bin. In particular, `Professor` computes the least squares solution using the Moore-Penrose pseudoinverse (see section 1.2.3). The pseudoinverse is implemented with a **singular value decomposition** of the dataset matrix, implemented with the `Eigen` [20] library. This step

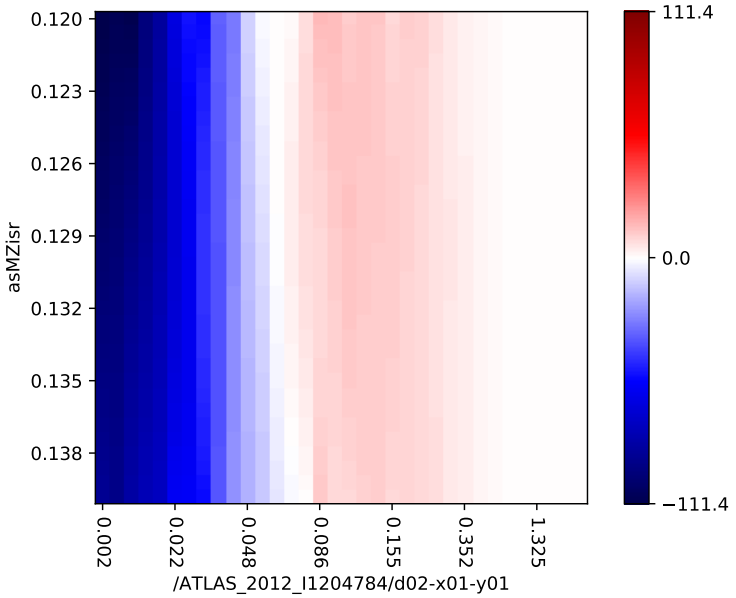


Figure 2.3: Example of sensitivity analysis for the parameter $\alpha_S(M_Z^2)$ of the initial state shower (see chapter 4 for details about this analysis/observable). The x-axis represents the bin center, the y-axis the parameter value and the color map represents the derivative of the bin value with respect to the parameter. The other parameters are evaluated at the center of the sampled parameter space.

is implemented in the `prof2-ipol` script.

There is a limit to the order of the polynomial: the number of coefficients (see formula 1.2) must be less than the number of runs in the dataset. Apart from this limit, the order of the polynomial must be chosen to prevent both underfitting and overfitting. The validity of the polynomial interpolation can be checked with the `prof2-residuals` script. Given an independent dataset of Monte Carlo runs (a validation set), the script computes the relative deviations of the predicted bin values from the corresponding actual values. This can be seen as a validation loss, and one can choose the best order for the polynomial fit by minimizing it.

In order for the tune to be precise, it must be performed on observables that are sensitive to those parameters. The script `prof2-sens` computes the derivative of the bin values with respect to the parameters, in order to check whether a parameter influences a specific bin or not. An example of sensitivity analysis is presented in figure 2.3.

2.3.3 Tuning

The goodness of fit function (or loss function) used by `Professor` is a χ^2 estimator. Let $\overline{h}^{(i)}(\mathbf{p})$ be the value of the i-th bin as predicted by the polynomial fit, and let

$h_{exp}^{(i)}$ be the experimental value. Then

$$\chi^2(\mathbf{p}) = \sum_{i=1}^{N} \frac{\left(\overline{h}^{(i)}(\mathbf{p}) - h_{exp}^{(i)}\right)^2}{\sigma_{(i)}^2} \tag{2.3}$$

This loss function is convenient because it takes advantage of the bins uncertainty. If the runs uncertainties have been interpolated, σ is the experimental uncertainty added in quadrature to the interpolated uncertainty, otherwise it is just the experimental uncertainty. Moreover, `Professor` implements the possibility to weight each bin differently in the χ^2 computation, in order to impose a preference on some bins, or excluding some other bins. The uncertainty of each best parameter is computed as the interval in which the χ^2/N_{dof} increments by one.

This last step is implemented in the `prof2-tune` script, where the minimization of the predicted χ^2 is performed using the `MINUIT` algorithm [21] via the `iminuit` [22] `Python` interface.

Chapter 3

MCNNTUNES procedure

The MCNNTUNES package implements two different strategies for generator tuning, both based on feedforward neural networks. In this chapter these two strategies are presented in detail, along with a description of the implementation details.

3.1 Per Bin Model

The first strategy is a parametrisation-based method similar to Professor [4]. In fact, the work cycle is divided in the same three steps as in Professor:

1. At first, the dataset is generated by sampling parameter configurations from the parameter space, and then running the generator with each configuration.
2. Then, the generator response is parametrised one bin at a time with the input parameters, using the dataset.
3. Finally, a χ^2 estimator calculated with the difference between the parametrised generator response and the experimental data is minimized. The parameters that minimize the χ^2 are the tunes.

The differences between this procedure and Professor show up in the details of the parametrisation and the minimization steps.

3.1.1 Parametrisation

The parametrisation of the generator response is modelled by feedforward neural networks, which take the parameters as input and return the value of a single bin. An independent neural network is used for each bin[1] (see figure 3.1 for an illustration). The models are trained with a gradient-based algorithm, as usual for feedforward neural networks, with mean squared error as loss. The details of the architecture, the choice of the optimization algorithm and its settings are all configurable by the user.

[1] The hyperparameters are the same for each bin, only the parameters of the biases and the kernels are different.

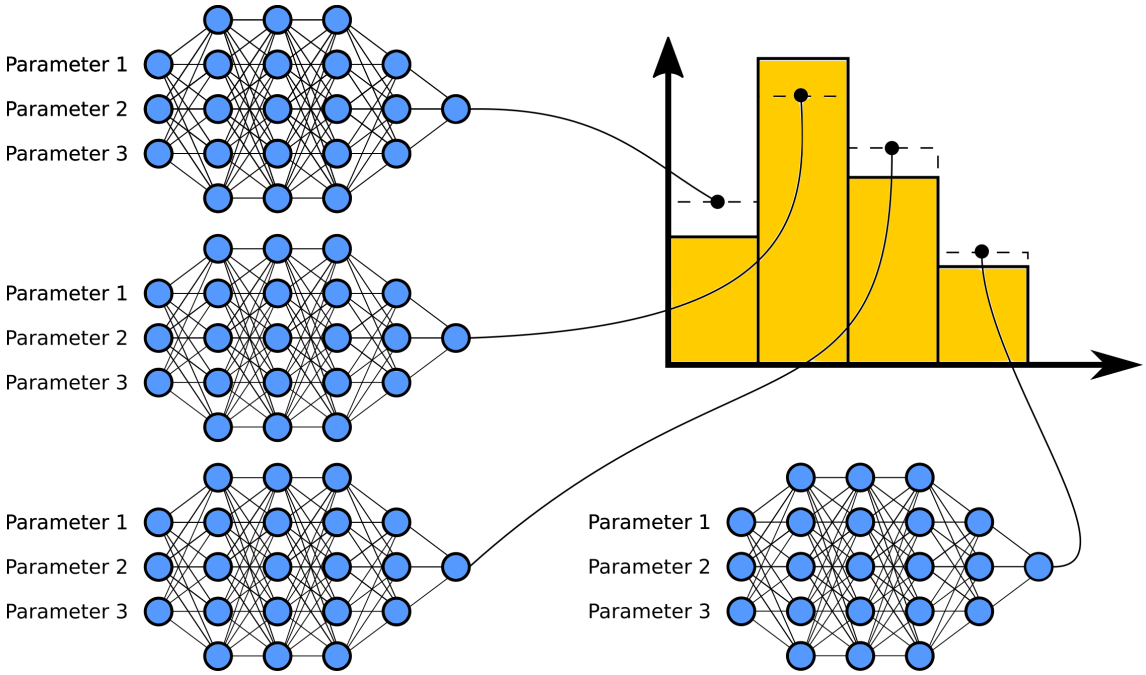


Figure 3.1: An illustration of the parametrisation of the generator response as implemented in the Per Bin Model.

3.1.2 Tuning

As in `Professor`, the parametrisation of the generator response enables the prediction of a Monte Carlo run result with every set of parameters for which the parametrisation is approximately valid. So, the χ^2 estimator calculated with the differences between the experimental data and the Monte Carlo run result can be approximately replaced by a surrogate one with the model instead of the Monte Carlo generator. The possibility of weighting each bin differently in the χ^2 is also implemented. Then, the minimization of this χ^2 is performed with the **Covariance Matrix Adaptation - Evolution Strategy** algorithm, which is a stochastic optimization method for non-linear non-convex functions. An introduction to this algorithm is presented in appendix C.

3.2 Inverse Model

Let $\mathbf{h}(\mathbf{p})$ be the function that represents the Monte Carlo event generator combined with an analysis: given a set of parameters, it returns the histograms related to some observables. The Inverse Model tries to learn the inverse of $\mathbf{h}(\mathbf{p})$, using a feedforward neural network with the bin values as input layer and with the generator parameters as output layer (see 3.2 for an illustration). In case of success, the model is able to predict the parameters used for the generator given the generator results. Then, tuning the generator consists in feeding the experimental data into the model and inferring the parameters that the generator needs to reproduce them. The uncertainties of the predictions are computed in three steps:

1. At first, the experimental data are resampled many times by using a multivariate Gaussian centered around the actual measurement, with a diagonal

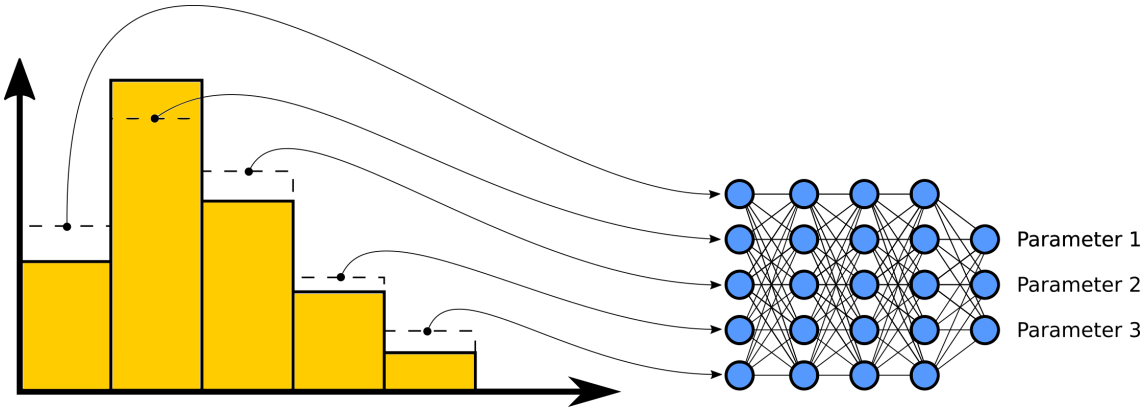


Figure 3.2: An illustration of the Inverse Model strategy

covariance matrix that includes the data uncertainties:

$$\text{norm} \cdot \exp\left(-\frac{1}{2}\sum_{j=1}^{N_{bins}}\left((x_j - h_{j,exp})^2/\sigma^2_{j,exp}\right)\right)$$

2. This set of histograms is fed into the neural network.

3. The output of the network is a distribution of predictions for each parameter (an example is shown in figure 4.3), and the uncertainties are computed as the standard deviations of these distributions.

3.2.1 Data augmentation

In this configuration the output variables (the parameters) are exact, but the input variables (the histogram bins) have a known uncertainty. In order to exploit this information, the **training with jitter** [23] method was implemented as an optional feature: at each training epoch, the entire dataset is resampled following the data uncertainty. More precisely, let $\mathbf{X}$ be the dataset matrix where each row is a Monte Carlo run, and each column is the value of a bin, and let σ_{ij} be the corresponding error for each element X_{ij} of $\mathbf{X}$. Then, the training is done by replacing $\mathbf{X}$ with $\overline{\mathbf{X}}$ such that each element $\overline{X}_{ij}$ is a random variable distributed according to a Gaussian with mean X_{ij} and variance σ^2_{ij}. $\overline{\mathbf{X}}$ is resampled at each epoch. This resembles a Gaussian noise layer applied to the input layer, but here the σ of the Gaussian noise is different for each node of the input layer, and for each element of the training set. This method can be seen as a form of regularization, as proven in [23].

3.3 Performance assessment

The performance of the procedure could be measured by means of a **closure test**. A closure test consists in using one Monte Carlo run as the experimental data, and then performing the tune; in this way, the obtained tunes can be directly compared with the real parameters used to generate that run. Notice that that run must be excluded from the training set, otherwise the result does not measure the ability of the procedure to generalize to new examples.

The program implements a performance assessment procedure based on closure tests. The user can provide two different datasets of Monte Carlo runs: a **training set**, used to train the model, and a **validation set**, used to perform closure tests. Once the model has been trained, a closure test is performed for each Monte Carlo run in the validation set, and a loss function defined as

$$L = \sum_i \frac{|p_i^{true} - p_i^{pred}|}{p_i^{true}} \tag{3.1}$$

is computed. The average of the losses of all closure tests is used as validation loss. This loss could be interpreted as an estimator of the accuracy of the tuning procedure, even though it is unsatisfactory: the experimental data and the Monte Carlo runs are not identically distributed nor generated by the same data-generating underlying process. In fact, the generator may be unable to represent the experimental data at all. However, this loss could be used to tune the hyperparameters of the model, or to compare different models.

3.4 Implementation

The procedures are implemented in the `mcnntune` script. The script accepts a configuration runcard in `YAML` format, which contains all program settings. This is the basic work cycle:

1. `mcnntune preprocess` loads the Monte Carlo runs and the experimental data, transforms the training set so that each input or output has mean 0 and variance 1, computes some useful statistics and saves all the data for future use.

2. `mcnntune model` trains the model specified in the runcard, and saves it for future use.

3. `mcnntune tune` performs the tune with the experimental data, and generates a `HTML` report with some information about the whole tuning process.

Some additional features are useful for performance assessment and hyperparameter tuning:

1. `mcnntune benchmark` performs the procedure presented in section 3.3 and returns a validation loss.

2. `mcnntune optimize` performs a hyperparameter search to obtain the best hyperparameters, using the loss returned by the benchmark mode as validation loss (see details in section 3.4.1).

3.4.1 Hyperparameter tuning

The hyperparameter search is implemented with the `Hyperopt` [24] package. `Hyperopt` is a `Python` library dedicated to the optimization of scalar-valued functions whose arguments are defined over a **search space** with a potentially complicated structure: some arguments could be real-valued, others could be discrete, and the

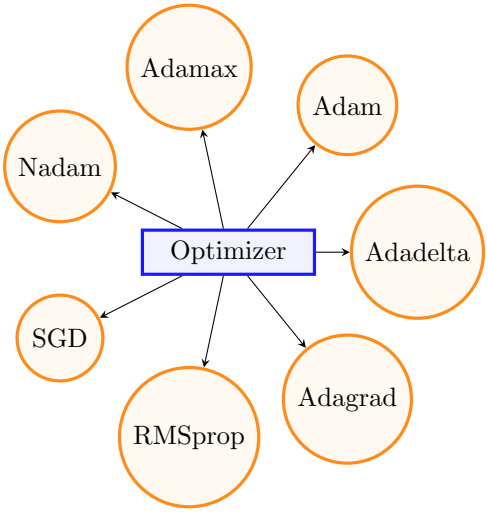
search space could be tree-structured, i.e. some variables are defined only when other parent variables take on a specific value. This is exactly the structure of the hyperparameter space[2]: some parameters are real-valued (e.g. learning rates), some are discrete (e.g. the choice of the optimization algorithm) and others are defined only when one or more parent parameters take on a certain value (e.g. the number of hidden layers and the number of units of each hidden layer). An illustration of a hyperparameter search space is given in figure 3.3. The implementation of `Hyperopt` requires the definition of a search space and the definition of the function to minimize. The search space must be defined by means of nested function expressions, that could be both deterministic or stochastic. Stochastic ones are the hyperparameters that will be tuned. These expressions are given by the user in the configuration runcard as strings in a `YAML` file. For example, the search space in figure 3.3 could look like this:

```
architecture: "hp.choice('layers', [
    [hp.quniform(f'size_{_}_2',5,10,1) for _ in range(2)],
    [hp.quniform(f'size_{_}_3',5,10,1) for _ in range(3)],
    [hp.quniform(f'size_{_}_4',5,10,1) for _ in range(4)]])"
optimizer: "hp.choice('optimizer', ['sgd','rmsprop',
    'adagrad','adadelta','adam','adamax','nadam'])"
optimizer_lr: "hp.loguniform('learning_rate', -10, -1)"
```

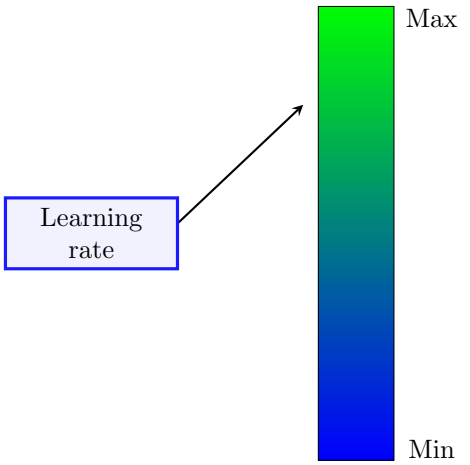
The function to minimize in hyperparameter tuning must receive the sampled hyperparameters as input, create a model with these hyperparameters, train it, and evaluate the validation loss of that model, or at least some sort of performance measurement with the "lower is better" format. The `mcnntune optimize` script computes the performance measurement presented in section 3.3 as validation loss, provided that a valid validation set of Monte Carlo runs is available. Specifically, it trains the model on the training set, then performs a closure test for every run in the validation set, computes the loss 3.1 for each of them, and finally returns the average of those losses as validation loss.
In addition, `Hyperopt` provides a `Trials` object that can store the details of each evaluation of the search space. Eventually, the search can be executed in parallel: the `Trials` object is replaced by a `MongoTrials` one, that interfaces with a `MongoDB` database. Then, the main script will send the evaluations as work items to the database. The script `hyperopt-mongo-worker` implements a worker that automatically collects a work item from the database, evaluates it and sends the results to the database. This makes `Hyperopt` a **distributed asynchronous** optimization algorithm; moreover, the database is **persistent**, that is if the main script crashes, it will recover the previous results.

`Hyperopt` implements two different algorithms: a random search, and a Sequential Model-Based Optimization algorithm called Tree-structured Parzen Estimator (see appendix B). `MCNNTUNES` uses the latter.

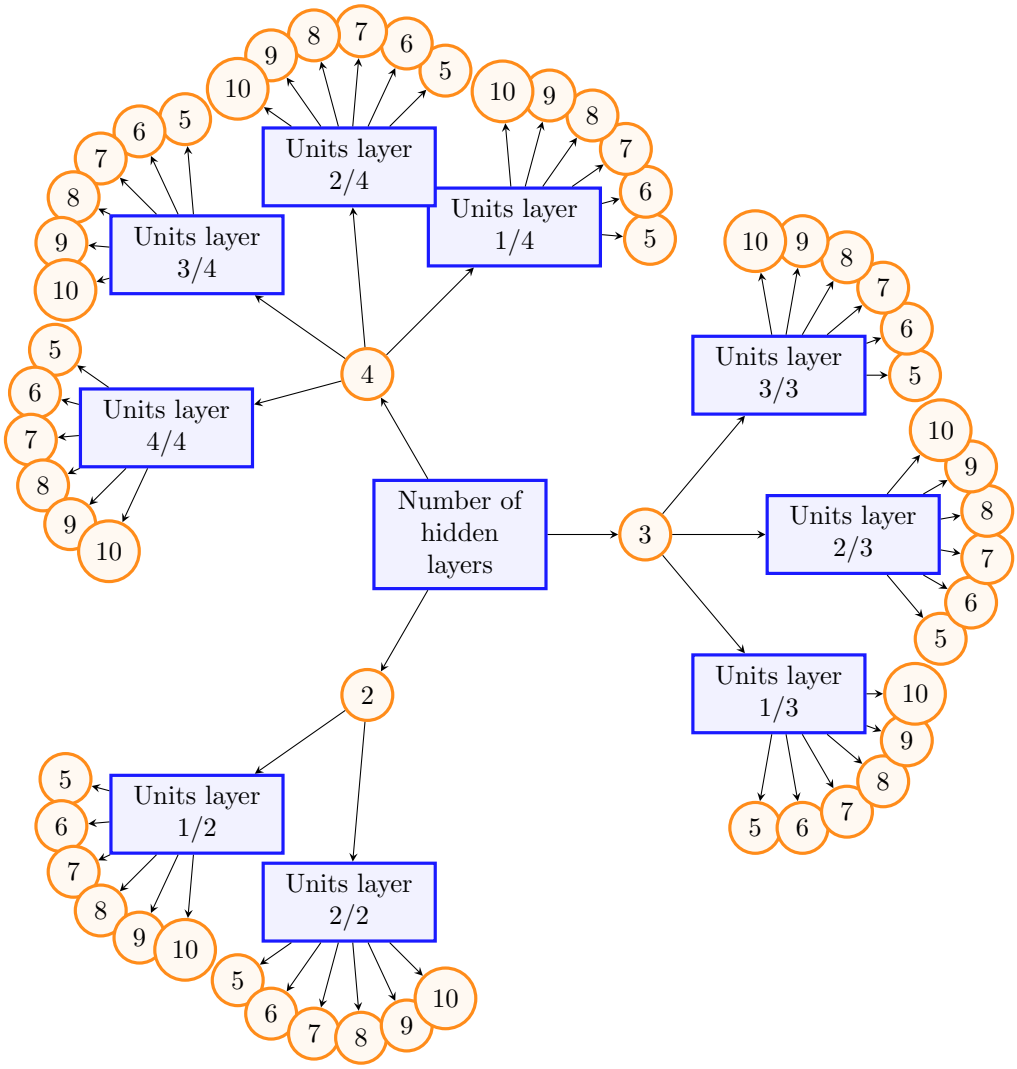
[2]In fact, `Hyperopt` was designed specifically for hyperparameter optimization of machine learning algorithms.



(a) The optimizer parameter is a discrete variable.



(b) The learning rate is a continuous variable.



(c) The architecture hyperparameters are tree-structured: some hyperparameters are defined only when their parent variables take on a particular value.

Figure 3.3: Example of a hyperparameter search space with a discrete variable (3.3a), a continuous variable (3.3b) and a tree-structured set of variables (3.3c).

3.4.2 Dependencies

The procedure is implemented in `Python`. The Monte Carlo runs (histograms) are loaded with the `YODA` library, which is the default histogram format of `Rivet` [17]. The basic operations are implemented with `NumPy` [25], while the machine learning aspects use a high-level neural networks API called `Keras` [26], with the `TensorFlow` framework [27] as backend. It uses the `pycma` package [28] for the Covariance Matrix Adaptation - Evolution Strategy algorithm. As stated in subsection 3.4.1, the hyperparameter tuning is implemented using the `Hyperopt` [24] package. The plots are made with `Matplotlib` [29] and `Seaborn` [30]. The results are written on `HTML` pages build with the `Jinja2` template engine. Moreover, some data management is made with `Pandas` [31].

Chapter 4

Testing MCNNTUNES

This chapter presents the testing phase of the MCNNTUNES procedures. The choice of the generator, the parameters with their variation ranges, the process and the observables on which performing the tunes were chosen following the AZ tune [32] as reference. The generation of some datasets of Monte Carlo runs is presented in section 4.1; an AZ-like tune that tries to reproduce some results of [32] is presented in section 4.2; a systematic hyperparameter search to obtain the best performing models is presented in section 4.3; finally, the results of section 4.2 are revisited using the best performing models found in section 4.3.

4.1 Datasets

The generation of the datasets followed the procedure presented in [32]. The Monte Carlo runs were generated with PYTHIA version 8.240 [15], interfaced with the Rivet [17] package, version 2.7.0. Two different analyses were performed: one involved the measurement of the Z/γ^* boson transverse momentum distribution[1] p_T^Z in pp collisions at $\sqrt{s} = 7$ TeV (analysis ATLAS 2014 I1300647), the other involved the measurement of angular correlation[2] ϕ_η^* [34] (analysis ATLAS 2012 I1204784), which probes the same physics of p_T^Z but with higher experimental resolution. Thus, the activated process was $f\overline{f} \to Z/\gamma^*$. The investigated parameters are the primordial k_T, the parton shower $\alpha_S(m_Z^2)$ and the parton shower damping factor for the lower p_T cut-off (both for the initial state radiation, ISR from now on), and the damping factor for the lower p_T cut-off for the multiparton interaction (see section 2.1 for details about these parameters). Two different datasets were generated: one, the most similar to [32], fixes the multiparton interaction parameter, while the other does not. The former will be the dataset 3P from now on, while the latter will be called

[1] Let the third axis be parallel to the incoming beam, and let p^μ be the particle four-momentum. Then

$$p_T = \sqrt{(p^1)^2 + (p^2)^2}$$

[2] Consider the decay channels $Z \to ee$ and $Z \to \mu\mu$, let $\phi_{acop} = \pi - \Delta\phi$, with $\Delta\phi$ the azimuthal angle between the outgoing leptons, and let θ_η^* be the scattering angle of the leptons with respect to the incoming proton beam, measured in the rest frame of the dilepton system, then

$$\phi_\eta^* = \tan(\phi_{acop}/2) \sin\theta_\eta^*$$

as proposed in [33].

4P. The variation range of each parameter and the setup of PYTHIA8 are presented in table 4.1. Measurements of ϕ_η^* from two runs of the dataset 4P were presented

Parameter	Dataset 3P	Dataset 4P
Primordial k_T [GeV]	1.0 - 2.5	1.0 - 2.5
ISR $\alpha_S(m_Z^2)$	0.120 - 0.140	0.120 - 0.140
ISR $p_{T,0}$ Ref [GeV]	0.5 - 2.5	0.5 - 2.5
MPI $p_{T,0}$ Ref [GeV]	2.18 (fixed)	1.9 - 2.2
PYTHIA8 base tune	tune 4C [35]	tune 4C [35]
Number of events	$4 \cdot 10^6$	$4 \cdot 10^6$
Number of runs	243	1024

Table 4.1: PYTHIA8 setup and variation ranges

in figure 2.2 as an example, while the analogous for p_T^Z are presented in figure 4.1. The number of events, fixed to $4 \cdot 10^6$, was chosen to obtain a statistical error which is comparable to the experimental error. The analysis of the uncertainties of a run picked from the 3P dataset is shown in figure 4.2.

4.2 AZ-like tune of PYTHIA8

In order to reproduce the results of [32], the tunes presented in this section involve only the dataset 3P. Following [32], the tunes used the dressed-level measurements, but only the ones inclusive in rapidity, and are performed only for $p_T^Z < 26$ GeV and $\phi_\eta^\star < 0.29$. The final tune has been performed using only the muon channel p_T^Z measurement and the electron channel $\phi_\eta^\star$ measurement. So, three different set of measurements are selected:

1. One with only p_T^Z measurements (called p_T^Z from now on), which corresponds to the two histograms in figure 4.1, respectively with code `ATLAS 2014 I1300647 d01-x01-y01` and `ATLAS 2014 I1300647 d01-x01-y02`.

2. Another one with only $\phi_\eta^\star$ measurements (called $\phi_\eta^\star$ from now on), which corresponds to the two histograms in figure 2.2, respectively with code `ATLAS 2012 I1204784 d02-x01-y01` and `ATLAS 2012 I1204784 d02-x01-y02`.

3. Finally, one with both observables ($p_T^Z \phi_\eta^*$ from now on), but only the muon channel for p_T^Z (`ATLAS 2014 I1300647 d01-x01-y02`) and the electron channel for ϕ_η^* (`ATLAS 2012 I1204784 d02-x01-y01`).

Then, a tune was performed for each of these sets, using both Professor and MCNNTUNES (with both models). The results are presented in table 4.2, along with the ones in [32]. The order of the polynomial was chosen following [32], while the hyperparameter configurations of MCNNTUNES were set manually. In section 4.3 the hyperparameters of both Professor and MCNNTUNES will be tuned, and in section 4.4 the tunes will be performed again with the tuned models. The results obtained with Professor are compatible with the ones in [32]. The results with the Per Bin model seem solid, but one parameter is predicted at the left bound of its variation

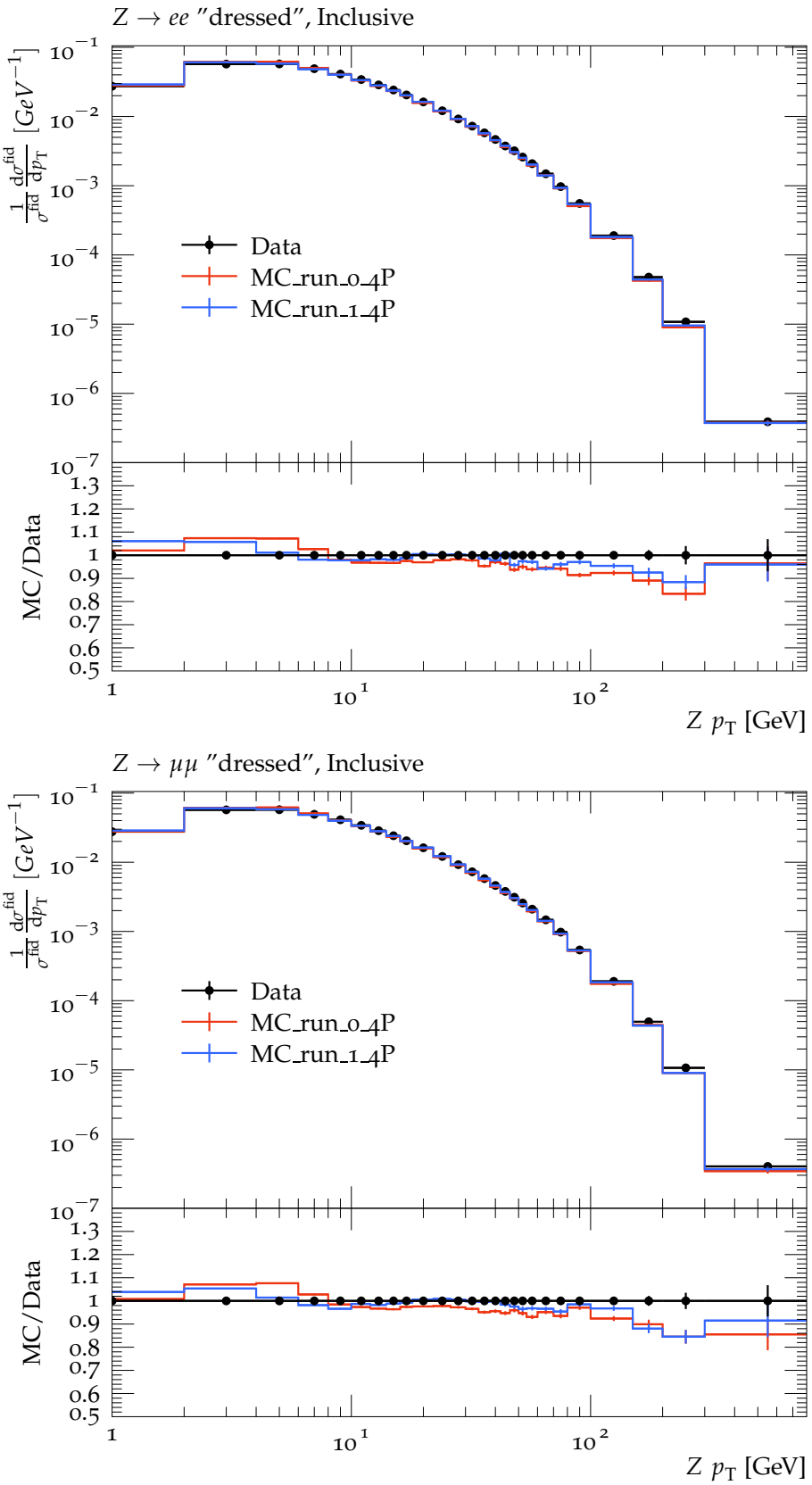


Figure 4.1: Measurements of the Z/γ^* boson transverse momentum distribution in pp collisions at $\sqrt{s} = 7$ TeV, for two runs picked from the dataset 4P.

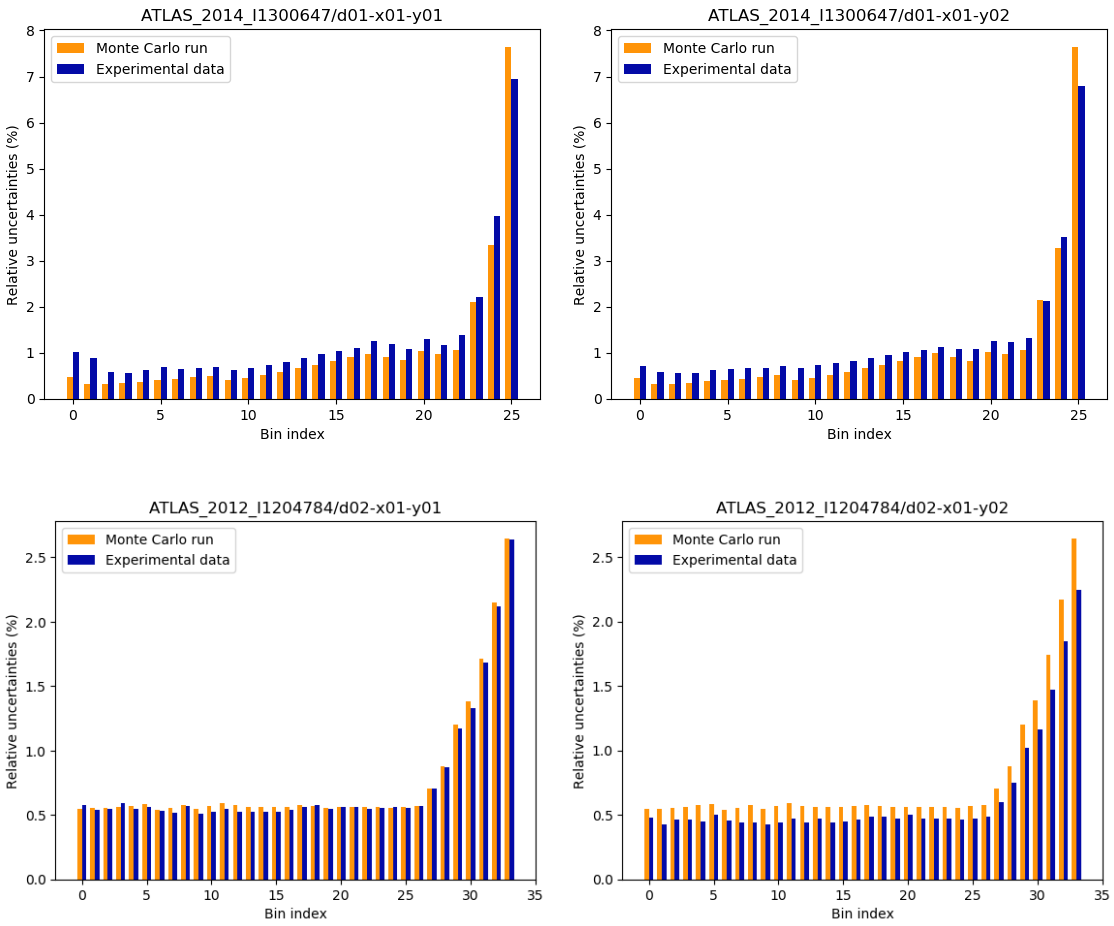


Figure 4.2: A comparison of the relative uncertainties of a Monte Carlo run picked from the dataset 3P. The comparison is less satisfactory for the ϕ^*_η measurements, because the experimental uncertainties are lower.

RESULTS OF [32]			
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.74 ± 0.03	1.73 ± 0.03	1.71 ± 0.03
ISR $\alpha_S(m_Z^2)$	0.1233 ± 0.0003	0.1238 ± 0.0002	0.1237 ± 0.0002
ISR $p_{T,0}$ Ref [GeV]	0.66 ± 0.14	0.58 ± 0.07	0.59 ± 0.08
χ^2_{min}/dof	1.26	1.33	1.42
`Professor` (fourth-order polynomial)			
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.77 ± 0.04	1.79 ± 0.04	1.75 ± 0.04
ISR $\alpha_S(m_Z^2)$	0.1233 ± 0.0003	0.1237 ± 0.0002	0.1237 ± 0.0003
ISR $p_{T,0}$ Ref [GeV]	0.5 ± 0.2	0.59 ± 0.09	0.54 ± 0.10
χ^2_{min}/dof	0.91	1.06	1.16
`MCNNTUNES` (Per Bin model, [3, 5], tanh, adam)			
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.75	1.79	1.76
ISR $\alpha_S(m_Z^2)$	0.1232	0.1235	0.1235
ISR $p_{T,0}$ Ref [GeV]	left bound	left bound	left bound
χ^2_{min}/dof	0.85	0.89	1.05
`MCNNTUNES` (Inverse model, [30, 20, 10], tanh, adam)			
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.68 ± 0.09	1.98 ± 0.08	1.64 ± 0.12
ISR $\alpha_S(m_Z^2)$	0.1234 ± 0.0006	0.1214 ± 0.0007	0.1254 ± 0.0008
ISR $p_{T,0}$ Ref [GeV]	0.7 ± 0.2	0.25 ± 0.08	0.9 ± 0.3
`MCNNTUNES` (Inverse model, [30, 20, 10], tanh, adam, DA)			
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.71 ± 0.05	1.80 ± 0.03	1.75 ± 0.05
ISR $\alpha_S(m_Z^2)$	0.1237 ± 0.0003	0.1234 ± 0.0002	0.1235 ± 0.0003
ISR $p_{T,0}$ Ref [GeV]	0.61 ± 0.07	0.60 ± 0.05	0.58 ± 0.03

Table 4.2: AZ-like tunes obtained with `Professor` and `MCNNTUNES`, compared with the ones obtained in [32]. DA is an abbreviation for data augmentation.

range, so the tunes may not be reliable. Finally, the results with the Inverse model are very different whether data augmentation was used or not: in the former case the results seem solid, in the latter the model seems unreliable, at least without hyperparameter tuning.

4.2.1 Unrestricted tunes

Removing the restriction $p_T^Z < 26$ GeV and $\phi_\eta^\star < 0.29$ modifies the obtained best parameters into the ones of table 4.3. The sets of measurements without restrictions will be marked by "all bins" from now on. The results seem more chaotic than the

PROFESSOR (fourth-order polynomial, all bins)			
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.83 ± 0.05	1.80 ± 0.04	1.72 ± 0.04
ISR $\alpha_S(m_Z^2)$	0.1253 ± 0.0003	0.12377 ± 0.00019	0.1241 ± 0.0002
ISR $p_{T,0}$ Ref [GeV]	1.32 ± 0.14	0.65 ± 0.10	0.55 ± 0.18
χ^2_{min}/dof	2.62	1.31	2.08
MCNNTUNES (Per Bin model, [3, 5], tanh, adam, all bins)			
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.74	1.78	1.75
ISR $\alpha_S(m_Z^2)$	0.1246	0.1236	0.1243
ISR $p_{T,0}$ Ref [GeV]	0.90	0.53	0.81
χ^2_{min}/dof	2.6	1.2	1.9
MCNNTUNES (Inverse model, [30, 20, 10], tanh, adam, DA, all bins)			
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.88 ± 0.08	1.73 ± 0.02	1.67 ± 0.08
ISR $\alpha_S(m_Z^2)$	0.1258 ± 0.0005	0.12351 ± 0.00012	0.1237 ± 0.0003
ISR $p_{T,0}$ Ref [GeV]	1.4 ± 0.3	0.57 ± 0.02	0.6 ± 0.2

Table 4.3: Tunes obtained with Professor, compared with the ones obtained with MCNNTUNES, minimizing over all bins.

restricted ones. However, an interesting fact may be worth mentioning: the $p_{T,0}$ value for the Inverse model with data augmentation relative to the p_T^Z measurements is way higher than the ones relative to the other two measurements (see last line of table 4.3). This is very similar to what happens with Professor (see third line). The same happens for the other two parameters, though less evident. A possible explanation for the Inverse model may be found by analysing the distribution of predictions within the experimental errors (see figure 4.3). The distributions seem multimodal: the minor peaks are more similar to the tunes relative to the other two measurements. This shows that the deep model of that tune is not robust against noise.

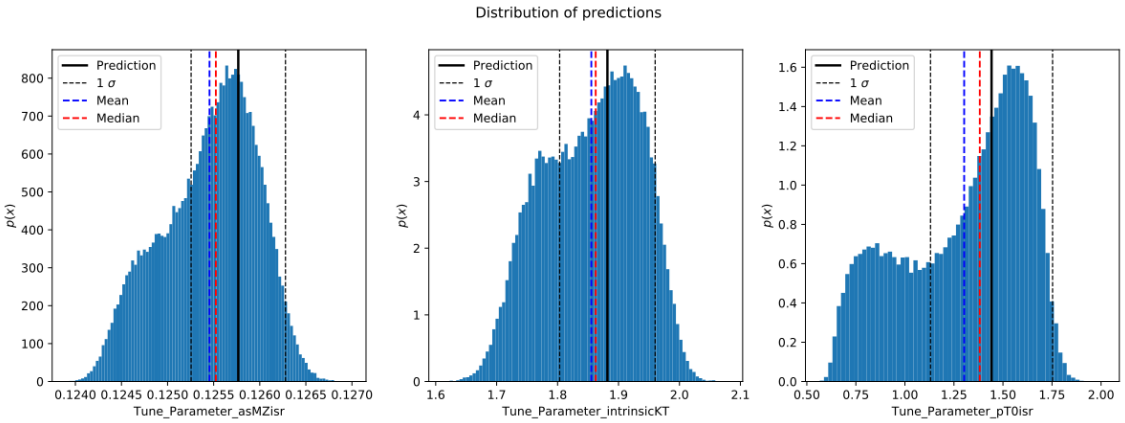


Figure 4.3: The spread of predictions of the Inverse model with data augmentation, for the p_T^Z measurements. This distributions are the output of the second point of the error estimation procedure described in section 3.2.

4.3 Performance measurements

This section presents some performance measurements on `Professor` and on the two different models implemented in `MCNNTUNES` (see chapter 3 for details). The performance measurement is the one presented in section 3.3, which consists on dividing the dataset in training set and validation set, training the model on the training set, performing many closure tests on the validation set, computing the loss 3.1 for each closure test, and taking the average of these losses. The uncertainty is computed as the standard deviation of the mean. The losses are always expressed as a percentage.

4.3.1 Closure tests with `Professor`

In order to compare, at least approximately, the performances of `Professor` and `MCNNTUNES`, the performance measurement described above was executed on `Professor`[3] for the combinations of measurements presented in section 4.2. Each dataset was divided in a training set (90%) and in a validation set (10%). The training set was used for the polynomial interpolation, and the validation set was used for the closure tests. A **grid search** was used to tune the order of the polynomial, along with some other options (`iminuit` strategy, scan for the best starting point, whether to extrapolate outside the sampled hypercube in parameter space)[4]. Results are presented in table 4.4. The grid searches for the polynomial order are shown in figures 4.4 and 4.5.

4.3.2 Level-zero closure tests

Coming back to `MCNNTUNES`, the ability of neural network models to interpolate the training data almost perfectly was checked by using validation sets that are subset of the training set. On an outer loop, an hyperparameter scan (with the validation

[3]The default `Professor` procedure was used, changing only the polynomial order and adding, whether specified, some options to `prof2-tune`.

[4]Respectively with options `-s`, `--scan-n` and `-x`.

Observables	default	`-s 2 -scan-n=100`	`-s 2 --scan-n=100 -x`
$\phi^\star_\eta$, 3P	3.6 ± 0.6 (3)	3.6 ± 0.6 (3)	3.6 ± 0.6 (3)
p_T^Z, 3P	2.3 ± 0.4 (3)	2.3 ± 0.4 (3)	2.4 ± 0.4 (3)
$\phi^\star_\eta p_T^Z$, 3P	3.0 ± 0.5 (4)	2.6 ± 0.4 (3)	2.6 ± 0.4 (3)
$\phi^\star_\eta$, 3P, all bins	2.7 ± 0.4 (3)	2.7 ± 0.4 (3)	2.7 ± 0.4 (3)
p_T^Z, 3P, all bins	2.0 ± 0.4 (3)	2.0 ± 0.4 (3)	2.0 ± 0.4 (3)
$\phi^\star_\eta p_T^Z$, 3P, all bins	1.8 ± 0.2 (3)	1.8 ± 0.2 (3)	1.8 ± 0.2 (3)
$\phi^\star_\eta$, 4P	4.4 ± 0.3 (3)	4.0 ± 0.3 (3)	4.4 ± 0.3 (3)
p_T^Z, 4P	3.1 ± 0.2 (4)	3.1 ± 0.2 (4)	3.3 ± 0.2 (5)
$\phi^\star_\eta p_T^Z$, 4P	3.7 ± 0.2 (4)	3.5 ± 0.2 (3)	3.9 ± 0.3 (4)
$\phi^\star_\eta$, 4P, all bins	3.9 ± 0.3 (3)	3.9 ± 0.3 (3)	3.9 ± 0.3 (3)
p_T^Z, 4P, all bins	3.0 ± 0.2 (4)	2.9 ± 0.2 (3)	3.1 ± 0.2 (3)
$\phi^\star_\eta p_T^Z$, 4P, all bins	3.3 ± 0.2 (4)	3.3 ± 0.3 (3)	3.4 ± 0.2 (3)

Table 4.4: Closure test results with `Professor` (order within parenthesis, `prof2-tune` options in the column header). The results are expressed as a percentage.

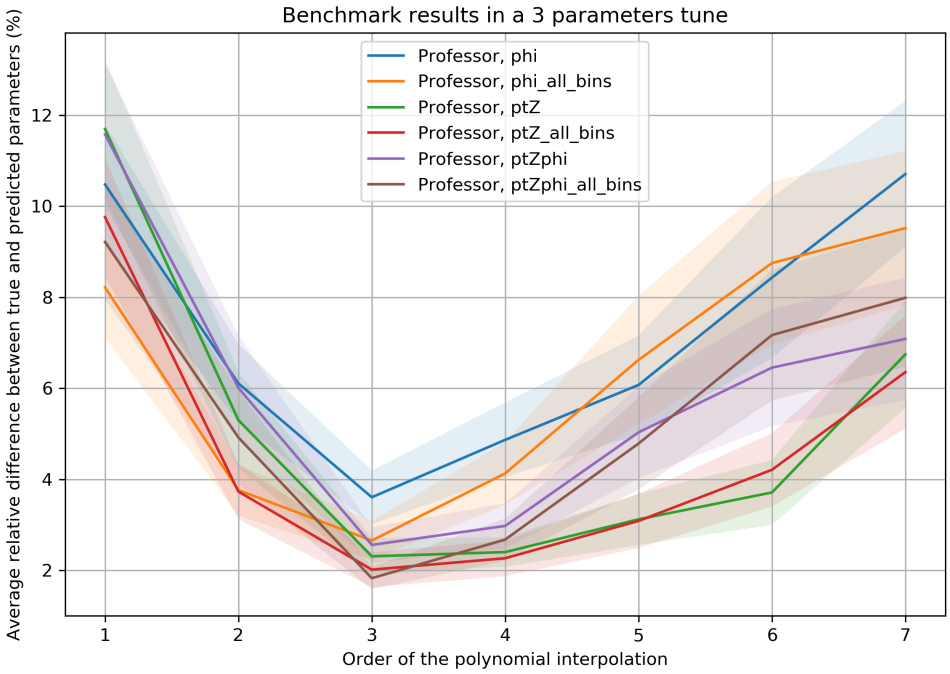


Figure 4.4: Closure test results for dataset 3P, options `-s 2 --scan-n=100`

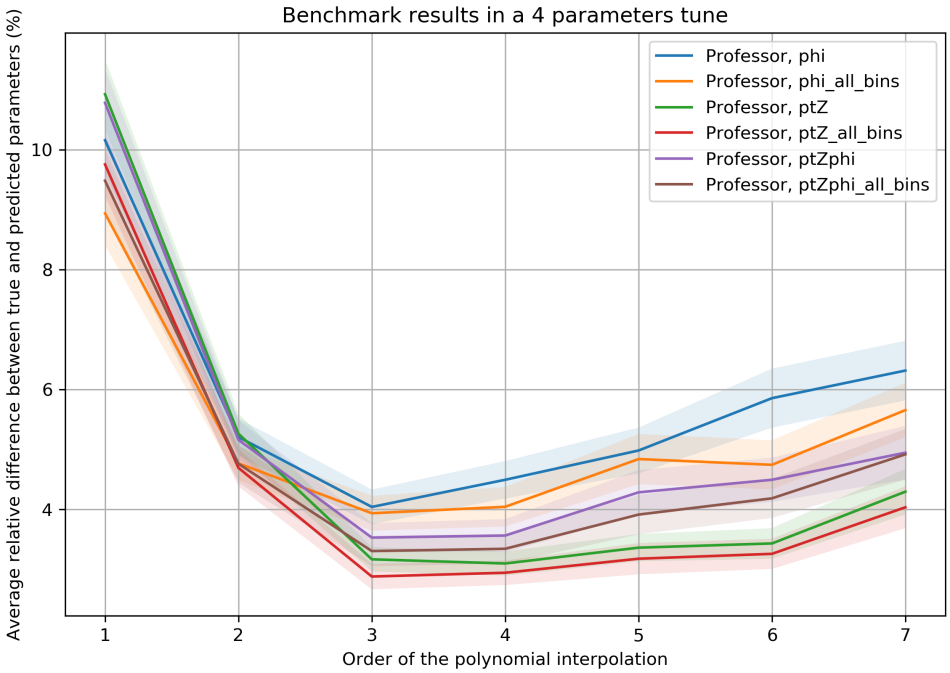


Figure 4.5: Closure test results for dataset 4P, options `-s 2 --scan-n=100`

loss presented at the beginning of section 4.3 as loss) searched for the best model. The expectations are that the validation loss of the best models should be very low. Of course these losses are not a good estimate of the performance of the model: they are just useful to check if the models can express their full expressive power, or to spot bugs in the code. Actually, during these tests one of the models performed poorly, and the investigation of the results brought to a bug that remained hidden from other tests.

Per Bin Model At first, a validation set with just one run copied from the training set was used: using the $\phi^\star_\eta$ measurements, after 100 evaluations the best configuration reached error 0.07 % with the 3P dataset, and error 0.06 % with the 4P dataset. Then, a 10% of the training set was used for the validation: with the same measurements, the best configuration reached error 0.2 % with the 3P dataset (99 evaluations), and error 0.7 % with the 4P dataset (58 evaluations). The `Hyperopt` configuration is presented in table 4.5.

Inverse Model Analogously, with the single run validation set and with the $\phi^\star_\eta$ measurements, after 100 evaluations the best configuration reached error $9 \cdot 10^{-5}$ % with the 3P dataset, and error 0.1 % with the 4P dataset. With 10% of the training set as validation, and the same measurements, after 100 evaluations the best configuration reached error $7 \cdot 10^{-4}$ % with the 3P dataset, and error 0.3 % with the 4P dataset. With the full training set as validation, and the same measurements, after 100 evaluations the best configuration reached error 10^{-4} % with the 3P dataset, and error 0.18 % with the 4P dataset. The `Hyperopt` configuration is presented in table 4.6.

Hyperparameter	Variation Range
Architecture	[8, 16, 16, 8]
Activation function	relu
Optimizer	rmsprop, adam, adadelta
Epochs	250 - 25000
Initializer	glorot uniform, glorot normal
Batch size	10 - 500
Learning rate	$e^{-11.5}$ - 1 (log uniform)

Table 4.5: `Hyperopt` configuration for the level zero closure test - Per Bin model

Hyperparameter	Variation Range
Architecture	[60, 40, 20]
Activation function	relu
Optimizer	rmsprop, adam, adadelta
Epochs	250 - 25000
Initializer	glorot uniform, glorot normal
Batch size	10 - 500
Learning rate	$e^{-11.5}$ - 1 (log uniform)

Table 4.6: `Hyperopt` configuration for the level zero closure test - Inverse model

4.3.3 Fine tuning

In this subsection a systematic hyperparameter search for the procedure fine tuning is presented for many combination of observables. Each dataset was divided in a training set (90%) and in a validation set (10%) (the same of subsection 4.3.1). The training is performed on the training set, and the closure tests for the validation loss on the validation set.

Per Bin Model The `Hyperopt` configuration is presented in table 4.7, while the results are presented in table 4.8.

Inverse Model The `Hyperopt` configuration is presented in table 4.9, while the results are presented in table 4.10. The configuration space was too broad for an efficient search, so a second trial was performed, using fixed architecture and activation function, and tuning the initializer, the optimizer and its learning rate. The `Hyperopt` configuration is presented in table 4.11. Then, another search was performed by fixing the parameters previously tuned, and tuning the architecture only. The `Hyperopt` configuration is presented in table 4.12. Unfortunately, the results (omitted here) were not better then the best model obtained in the first trial, except in a few situations. However, the analysis of the distribution of losses in the hyperparameter space may be useful. Considering all observables, `Hyperopt` selected all optimizers and initializers, so it does not seem that choosing different optimizers and initializers (within the hyperparameter space defined in table 4.11) makes a difference. The dependence on the learning rate, instead, was quite the same for all observables, and it is showed in figure 4.6 for a scansion in which the

Hyperparameter	Variation Range
# hidden layers	2-4
Units per layer	2-20
Activation function	tanh, relu, sigmoid
Optimizer	sgd, rmsprop, adagrad, adadelta, adam, adamax, nadam
Epochs	250, 500, 1000, 2500, 5000, 7500, 10000
Batch size	100, 200, 300, 400, 500
Number of trials	1000

Table 4.7: `Hyperopt` configuration for the Per Bin Model

Observables	Direct (%)	`Professor` (%)
$\phi_\eta^\star$, 3P	3.1 ± 0.5	3.6 ± 0.6
$\phi_\eta^\star$, 4P	4.0 ± 0.3	4.0 ± 0.3

Table 4.8: `Hyperopt` search results for the Per Bin Model

Hyperparameter	Variation Range
# hidden layers	2-5
Units per layer	2-20
Activation function	tanh, relu, sigmoid
Optimizer	sgd, rmsprop, adagrad, adadelta, adam, adamax, nadam
Epochs	250, 500, 1000, 2500, 5000, 7500, 10000
Batch size	100, 200, 300, 400, 500
Number of trials	1000

Table 4.9: `Hyperopt` configuration for the Inverse Model - first trial

Observables	Inverse (%)	Inverse with DA (%)	`Professor` (%)
$\phi_\eta^\star$, 3P	2.9 ± 0.4	3.0 ± 0.3	3.6 ± 0.6
p_T^Z, 3P	1.62 ± 0.18 (x5)	1.6 ± 0.2 (x5)	2.3 ± 0.4
$\phi_\eta^\star p_T^Z$, 3P	2.0 ± 0.2	2.2 ± 0.3	2.6 ± 0.4
$\phi_\eta^\star$, 3P, all bins	3.0 ± 0.4	2.6 ± 0.3	2.7 ± 0.4
p_T^Z, 3P, all bins	1.59 ± 0.19 (x5)	1.5 ± 0.3	2.0 ± 0.4
$\phi_\eta^\star p_T^Z$, 3P, all bins	2.1 ± 0.3 (x5)	1.7 ± 0.2 (x5)	1.8 ± 0.2
$\phi_\eta^\star$, 4P	3.5 ± 0.3 (x5)	3.7 ± 0.3	4.0 ± 0.3
p_T^Z, 4P	2.57 ± 0.16	2.64 ± 0.17	3.1 ± 0.2
$\phi_\eta^\star p_T^Z$, 4P	3.0 ± 0.2	2.98 ± 0.18	3.5 ± 0.2
$\phi_\eta^\star$, 4P, all bins	3.7 ± 0.3 (x5)	3.6 ± 0.3	3.9 ± 0.3
p_T^Z, 4P, all bins	2.79 ± 0.19	2.69 ± 0.18	2.9 ± 0.2
$\phi_\eta^\star p_T^Z$, 4P, all bins	2.9 ± 0.2	3.0 ± 0.2	3.3 ± 0.2

Table 4.10: `Hyperopt` search results for the Inverse Model - first trial. Results marked with x5 were obtained with 5000 trials instead of 1000.

Hyperparameter	Variation Range
Number of hidden layers	3
Units per layer	25
Activation function	sigmoid
Optimizer	rmsprop, adadelta, adam
Optimizer learning rate	$e^{-11.5}$ - $e^{-1.5}$ (log uniform)
Initializer	Glorot uniform, glorot normal
Epochs	2500 - 25000 in steps of 500
Batch size	16
Number of trials	500 (3P) or 250 (4P)

Table 4.11: `Hyperopt` configuration for the Inverse Model - second trial

Hyperparameter	Variation Range
Number of hidden layers	3
Units per layer	10 - 40 in step of 2
Activation function	sigmoid
Optimizer	previously tuned
Optimizer learning rate	previously tuned
Initializer	previously tuned
Epochs	2500 - 30000 in steps of 500
Batch size	16
Number of trials	500 (3P) or 250 (4P)

Table 4.12: `Hyperopt` configuration for the Inverse Model - third trial

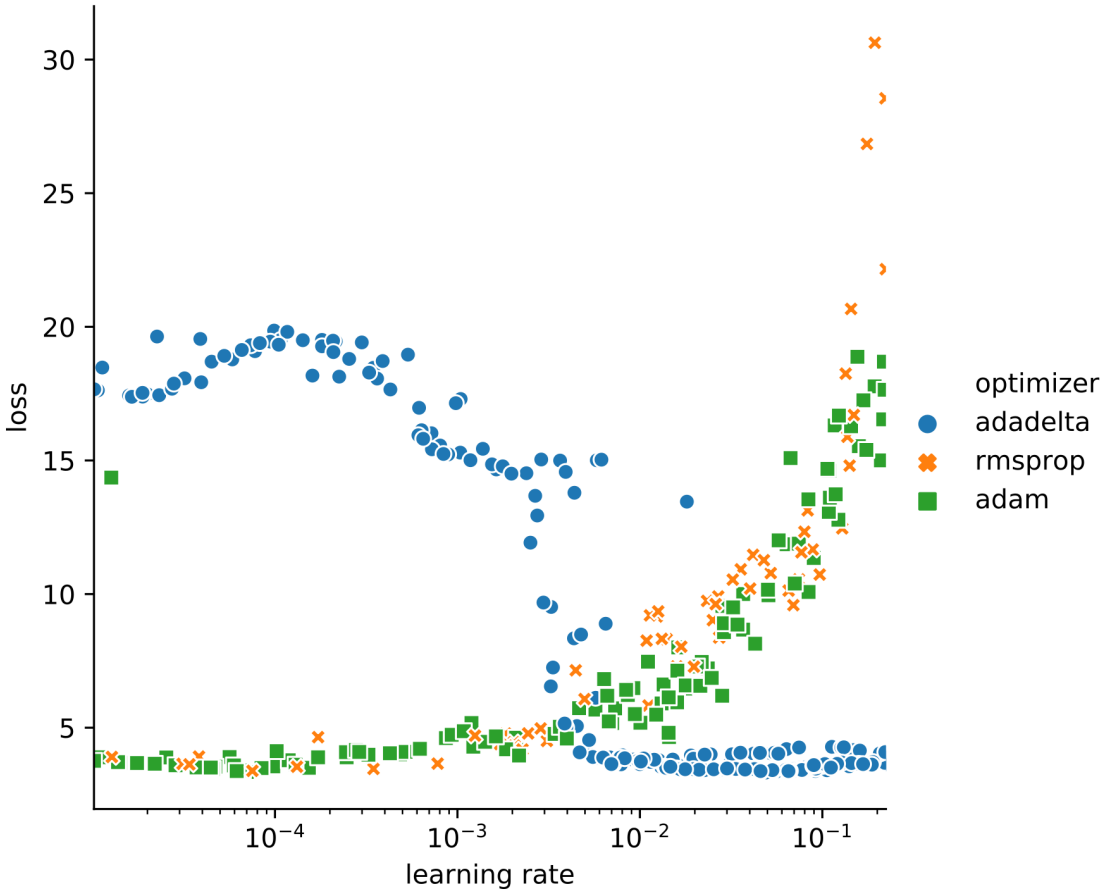


Figure 4.6: The losses distribution over the learning rate, marginalized over the other hyperparameters, for $\phi_\eta^\star$ with dataset 3P. RMSProp and Adam behave reasonably: if the learning rate is too high, the algorithm may not converge, if it is too low, it will perform good but the efficiency falls. AdaDelta behaves differently because the default learning rate, which is the recommended value, is 1, at the right bound of the search interval.

trends are evident for all optimizers. It seems that the default value for Adam and RMSProp, which is 10^{-3}, works fine. The same holds for AdaDelta[5], which is 1. Actually, `Hyperopt` selected lower learning rates: whether it is worth the higher computational cost (in terms of epochs) is questionable. This may explain why this two-step hyperparameter procedure does not improve the results. A focus on the architecture only may be more beneficial. Finally, an analysis of the closure test results of the best model for the $\phi_\eta^\star$ measurements, dataset 4P, third hyperparameter search (with data augmentation) is presented in figure 4.7. It shows an interesting fact: while the other parameters present no strong failure patterns, the multiparton interaction $p_{T,0}$ parameter does: the model predicts always a value biased toward the center of the variation range. This happens often, even though sometimes is less evident.

[5]Actually, in the original paper [36] AdaDelta is presented without a tunable learning rate, but `Keras` implements it.

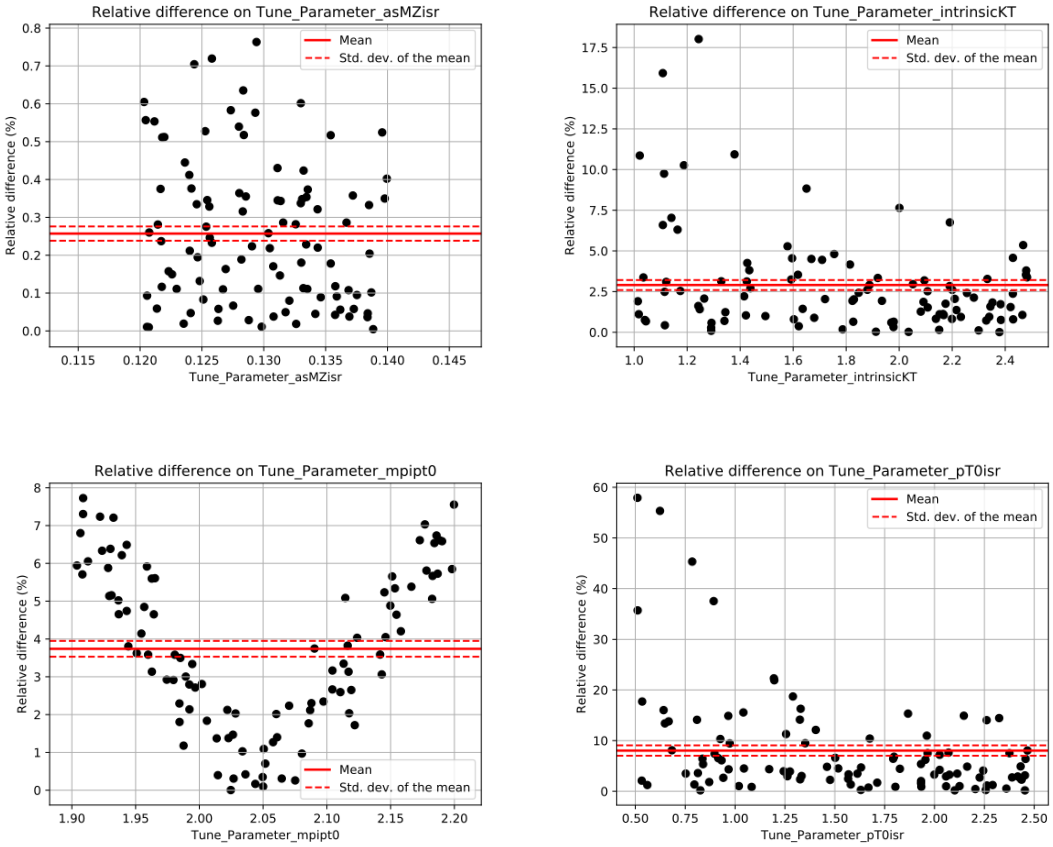


Figure 4.7: Analysis of closure test results of the best model for the $\phi^{\star}_{\eta}$ measurements, dataset 4P, third hyperparameter search, with data augmentation. The parameter value is on the x-axis, while the loss 3.1 is on the y-axis. While the other parameters present no strong failure patterns, the multiparton interaction $p_{T,0}$ parameter does: the model predicts always a value biased toward the center of the variation range.

Hyperparameter	Variation Range
Number of hidden layers	3-4
Units per layer	10 - 50 in step of 2
Activation function	sigmoid
Optimizer	adam
Optimizer learning rate	default value
Initializer	glorot uniform
Epochs	2500 - 15000 in steps of 500
Batch size	128
Number of trials	1000

Table 4.13: `Hyperopt` configuration for the Inverse Model - architecture only

4.3.4 Test errors

The hyperparameter scan could potentially introduce a bias in the estimation of the performance of the model. This happens because the model with the best configuration on the validation set could overfit the validation set. In order to have an unbiased estimator, the final loss should be computed on an independent additional dataset, called **test set**. Otherwise, comparisons with `Professor` would be misleading. The results presented in this subsection follow this procedure, for both `MCNNTUNES` and `Professor`:

1. Split the dataset into **training set** (80%), **validation set** (10%) and **test set** (10%).

2. Tune the hyperparameters of the model by training each configuration on the training set and selecting the configuration with the best loss computed on the validation set (validation loss). For `Professor`, this was obtained by performing a grid search for the polynomial order. The other options were kept at their default values, except the options `-s 2 --scan-n=100` for `prof2-tune`. For `MCNNTUNES`, this was obtained by running `Hyperopt`, feeding it with the validation loss. The `Hyperopt` configurations were the one in table 4.9 plus another one focused on the architecture only, presented in table 4.13. Whether using data augmentation or not, instead, was chosen by performing a grid search on top of the `Hyperopt` scansion.

3. Train the best model on the training and the validation set, and evaluate its performance on the **test set**.

The two datasets presented in section 4.1 were too small for a validation-test split, so they were extended to reach 512 runs for the 3P dataset (3PE from now on) and 1280 for the 4P (4PE from now on). The results are presented in table 4.14, for the Inverse model only. The performance of `MCNNTUNES` turns out to be slightly better than `Professor`, on average. Results are however limited to this particular benchmark, and may change with different random seeds, losses, datasets, parameters and observables. Moreover, this benchmark uses Monte Carlo runs, and not experimental data, so this performance measurement will not estimate the real tuning precision with real experimental data, because experimental data and Monte Carlo runs are not drawn from the same underlying data-generating distribution function.

Observables	Inverse (%)	`Professor` (`-s 2 --scan-n=100`) (%)
$\phi^\star_\eta$, 3P	3.7 ± 0.5	4.8 ± 0.7 (3)
p^Z_T, 3P	2.6 ± 0.3	3.1 ± 0.5 (3)
$\phi^\star_\eta p^Z_T$, 3P	3.2 ± 0.4	3.6 ± 0.5 (3)
$\phi^\star_\eta$, 3P, all bins	4.4 ± 0.6	4.2 ± 0.7 (3)
p^Z_T, 3P, all bins	2.5 ± 0.3	2.6 ± 0.4 (3)
$\phi^\star_\eta p^Z_T$, 3P, all bins	3.1 ± 0.4	3.1 ± 0.5 (3)
$\phi^\star_\eta$, 4P	3.5 ± 0.2	4.6 ± 0.3 (5)
p^Z_T, 4P	2.71 ± 0.16	3.28 ± 0.18 (5)
$\phi^\star_\eta p^Z_T$, 4P	3.2 ± 0.2	3.8 ± 0.3 (4)
$\phi^\star_\eta$, 4P, all bins	3.7 ± 0.2	4.0 ± 0.3 (4)
p^Z_T, 4P, all bins	2.89 ± 0.15	3.2 ± 0.2 (4)
$\phi^\star_\eta p^Z_T$, 4P, all bins	3.0 ± 0.2	3.4 ± 0.2 (4)

Table 4.14: Test errors - Inverse Model against `Professor`

4.4 Some final tunes

Finally, the datasets 3PE and 4PE were used to perform some final tunes. The configurations used are the ones selected in the hyperparameter tuning step of subsection 4.3.4, except for the Per Bin model for which the hyperparameters were chosen manually. The whole datasets were used for training. The results for the 3PE dataset are presented in table 4.15 and 4.16 for the restricted and unrestricted tunes respectively. Analogously, the results for the 4PE dataset are presented in table 4.17 and 4.18. A few comments on these tunes may be made:

- `Professor` and the Per Bin model give similar results, usually compatible with each other.
- The Inverse model sometimes gives different results: the agreement with `Professor` is usually good for the primordial k_T parameter, with some exception. The same happens for the parameter $\alpha_S^{ISR}(m_Z^2)$. The results relative to the remaining parameters are harder to analyse and will be discussed in the next points.
- The estimation of the multiparton interaction parameter by the Inverse model does not work, unfortunately: the model predicts always a parameter near the midpoint of the variation range, as noted also during the fine tuning of the model and showed in figure 4.7. This did not prevent the Inverse model to perform better when the errors are averaged over all parameters, so the possibilities are that the model compensates with a better estimation of the other parameters, or that the estimation of `Professor` is worse than the one of a model with a trivial behaviour. The latter seems the right one: a simple numerical simulation shows that a model that predicts always the value 2.05 GeV for the parameter MPI $p_{T,0}$ Ref would make an average relative error of 3.7%. `Professor`, in the test errors of section 4.3.4, makes an average relative error of 4.8 % in the best case (considering the MPI $p_{T,0}$ Ref only), which means that looses against a trivial predictor. So, both algorithms fail, and the

fixed prediction is just the way the learning algorithm found to minimize the training loss.

- Unfortunately, many results suggest that the best value for ISR $p_{T,0}$ Ref is somewhere outside the left bound of its variation range. This is easy to observe for two steps methods like `Professor` and the Per Bin model: they model the generator behaviour in the parameter space, more precisely in the hyperrectangle populated by the dataset, while the tunes are found by a minimization algorithm that explores this hyperrectangle. When the tunes seem outside of the variation ranges the algorithm finds a minimum at the boundary of the parameters hyperrectangle. The minimizers can extrapolate the results outside of the variation ranges, but there the models may be unreliable. For the Inverse model it is more complicated, because the bounds are not hard-coded into the model. Moreover, it is difficult to understand if the experimental data are near some Monte Carlo runs, so that the prediction is reliable: the envelopes are not useful because they show only whether the experimental data are inside the bounding box of the Monte Carlo runs in histograms space, but the runs do not populate this bounding box uniformly. When `Professor` founds a value inside the variation range, the corresponding value for the Inverse model is compatible with it. When `Professor` suggests a value outside it, the behaviour of the Inverse model varies: sometimes it directly predicts a value outside the variation range, sometimes a value near the left bound, sometimes a value further away.

PROFESSOR			
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.77 ± 0.04	1.80 ± 0.04	1.77 ± 0.04
ISR $\alpha_S(m_Z^2)$	0.1232 ± 0.0002	0.1236 ± 0.0002	0.1236 ± 0.0002
ISR $p_{T,0}$ Ref [GeV]	left bound	left bound	left bound
χ^2_{min}/dof	1.04	0.95	1.24
MCNNTUNES (Per Bin model, [20, 15, 10], tanh, adamax)			
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.75	1.76	1.74
ISR $\alpha_S(m_Z^2)$	0.1233	0.1236	0.1236
ISR $p_{T,0}$ Ref [GeV]	left bound	left bound	left bound
χ^2_{min}/dof	0.82	0.90	1.08
MCNNTUNES (Inverse model, best hyperparameters)			
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.75 ± 0.05	1.81 ± 0.05	1.77 ± 0.04
ISR $\alpha_S(m_Z^2)$	0.1249 ± 0.0006	0.1233 ± 0.0004	0.1241 ± 0.0005
ISR $p_{T,0}$ Ref [GeV]	0.9 ± 0.2	0.24 ± 0.18	0.8 ± 0.2

Table 4.15: Dataset 3PE.

PROFESSOR			
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.82 ± 0.05	1.79 ± 0.04	1.74 ± 0.04
ISR $\alpha_S(m_Z^2)$	0.1252 ± 0.0003	0.12370 ± 0.00017	0.1244 ± 0.0002
ISR $p_{T,0}$ Ref [GeV]	1.27 ± 0.16	left bound	0.80 ± 0.14
χ^2_{min}/dof	2.71	1.24	2.04
MCNNTUNES (Per Bin model, [20, 15, 10], tanh, adamax)			
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.79	1.75	1.75
ISR $\alpha_S(m_Z^2)$	0.1251	0.1238	0.1246
ISR $p_{T,0}$ Ref [GeV]	1.18	0.54	0.89
χ^2_{min}/dof	2.46	1.17	1.87
MCNNTUNES (Inverse model, best hyperparameters)			
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.87 ± 0.04	1.79 ± 0.03	1.75 ± 0.05
ISR $\alpha_S(m_Z^2)$	0.1256 ± 0.0003	0.12363 ± 0.00016	0.1244 ± 0.0003
ISR $p_{T,0}$ Ref [GeV]	1.36 ± 0.14	0.61 ± 0.08	0.85 ± 0.14

Table 4.16: Dataset 3PE, all bins.

	PROFESSOR		
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.76 ± 0.05	1.80 ± 0.05	1.79 ± 0.04
ISR $\alpha_S(m_Z^2)$	0.1233 ± 0.0003	0.1237 ± 0.0002	0.1236 ± 0.0002
ISR $p_{T,0}$ Ref [GeV]	left bound	left bound	0.5 ± 1.9
MPI $p_{T,0}$ Ref [GeV]	2.11 ± 0.06	2.13 ± 0.07	right bound
χ^2_{min}/dof	0.98	0.93	0.97
MCNNTUNES (Per Bin model, [20, 15, 10], tanh, adamax)			
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.70	1.76	1.76
ISR $\alpha_S(m_Z^2)$	0.1233	0.1237	0.1236
ISR $p_{T,0}$ Ref [GeV]	left bound	left bound	left bound
MPI $p_{T,0}$ Ref [GeV]	1.95	right bound	right bound
χ^2_{min}/dof	0.86	0.82	0.94
MCNNTUNES (Inverse model, best hyperparameters)			
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.69 ± 0.07	1.69 ± 0.04	1.60 ± 0.06
ISR $\alpha_S(m_Z^2)$	0.1246 ± 0.0007	0.12345 ± 0.00018	0.1238 ± 0.0007
ISR $p_{T,0}$ Ref [GeV]	0.9 ± 0.2	0.29 ± 0.09	0.6 ± 0.2
MPI $p_{T,0}$ Ref [GeV]	2.0468 ± 0.0011	2.0431 ± 0.0009	2.0450 ± 0.0009

Table 4.17: Dataset 4PE.

	PROFESSOR		
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.80 ± 0.05	1.75 ± 0.04	1.69 ± 0.04
ISR $\alpha_S(m_Z^2)$	0.1253 ± 0.0003	0.12370 ± 0.00018	0.1241 ± 0.0003
ISR $p_{T,0}$ Ref [GeV]	1.33 ± 0.14	left bound	0.5 ± 0.4
MPI $p_{T,0}$ Ref [GeV]	2.00 ± 0.06	2.01 ± 0.04	2.05 ± 0.04
χ^2_{min}/dof	2.65	1.11	2

	MCNNTUNES (Per Bin model, [20, 15, 10], tanh, adamax)		
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.79	1.73	1.66
ISR $\alpha_S(m_Z^2)$	0.1252	0.1237	0.1241
ISR $p_{T,0}$ Ref [GeV]	1.32	left bound	left bound
MPI $p_{T,0}$ Ref [GeV]	1.90	1.99	2.01
χ^2_{min}/dof	2.39	1.07	1.82

	MCNNTUNES (Inverse model, best hyperparameters)		
Parameter	p_T^Z	$\phi_\eta^\star$	p_T^Z $\phi_\eta^\star$
Primordial k_T [GeV]	1.79 ± 0.05	1.90 ± 0.06	1.72 ± 0.04
ISR $\alpha_S(m_Z^2)$	0.1250 ± 0.0003	0.1231 ± 0.0002	0.1238 ± 0.0003
ISR $p_{T,0}$ Ref [GeV]	1.12 ± 0.15	0.526 ± 0.016	0.83 ± 0.13
MPI $p_{T,0}$ Ref [GeV]	2.0473 ± 0.0005	2.04411 ± 0.00016	2.0460 ± 0.0004

Table 4.18: Dataset 4PE, all bins.

Chapter 5

Conclusion

A deep learning approach to event generator tuning was presented by introducing two different procedures, called Per Bin strategy and Inverse strategy respectively. The former is a variation of the `Professor` tuning procedure, while the latter is a completely different approach. The procedures were tested with pseudo-data (closure tests) and real experimental data, though in low dimensional parameter spaces. The Per Bin model tests with pseudo-data were very limited, due to computational time constraints, but showed solid results. The test with real experimental data showed a behaviour similar to the one of `Professor`. The Inverse model tests with pseudo-data presented performances slightly better than the ones of `Professor`, while the test with real experimental data showed some differences from the other procedures. The advantages of this deep learning approach may be listed as follows:

- The parametrisation is based on neural networks, which have the properties guaranteed by the Universal Approximation Theorem (see subsection 1.2.2), so the learned functions are not biased by any particular functional form. Moreover, the models can learn highly non-linear functions, at least in principles.
- In the case of the Inverse model, the two-step method of parametrisation-minimization is replaced by a single-step one, which is conceptually simpler, even though the function to learn is more complicated.

On the other hand, the disadvantages may be summed up as follows:

- The procedure brings all the difficulties that are typical of deep learning algorithms: the complexity of the training step, the dependence of the performance on the choice of the hyperparameters, the difficulty in the interpretation of the behaviour of the trained model, the overfitting problem.
- The hyperparameter tuning is computationally expensive, especially for the Per Bin model, and this prevents the models to reach their full potential.
- The Inverse strategy introduces some practical problems, e.g. the error estimation and the reliability of the predictions when the experimental measurements have no Monte Carlo runs near them.

The behaviour of the procedures with real experimental data is still unclear, and requires more in-depth studies. In addition, whether the procedures scale well with the number of parameters is still to be determined. Future investigations may involve studying their performances in high dimensional parameter spaces. Finally, further developments may solve the practical problems highlighted above.

Appendix A

Training of neural networks

Training a neural network with gradient-based optimization algorithms requires the ability to compute the gradient of the loss function with respect to the parameters of the network. Let $\mathbf{f}(\mathbf{x};\boldsymbol{\theta})$ be a generic neural network, and let $L(\mathbf{f}(\mathbf{x};\boldsymbol{\theta}),\mathbf{y})$ be its loss function computed on the training example $(\mathbf{x},\mathbf{y})$. Then, the goal is to compute

$$\nabla_{\boldsymbol{\theta}} L(\mathbf{f}(\mathbf{x};\boldsymbol{\theta}),\mathbf{y}) \tag{A.1}$$

in order to combine it with a gradient-based minimizer. This appendix presents the back-propagation algorithm (section A.1), which is an efficient way to compute such gradient, and some gradient-based optimizers used for neural networks training (section A.2).

A.1 Backpropagation algorithm

The gradient $\nabla_{\boldsymbol{\theta}} L(\mathbf{f}(\mathbf{x};\boldsymbol{\theta}),\mathbf{y})$ with a fully connected feedforward neural network is conceptually simple to write down: in fact, a neural network is made of a composition of many simple functions, and so it is just an application of the chain rule of calculus. In practice, the number of parameters could be huge, and a naive implementation will eventually recompute the same quantities over and over, especially for deep networks. For example, computing the derivatives with respect to the parameters of a certain hidden layer requires computing them with respect to all successive layers. The back-propagation algorithm is designed to avoid these useless computations. During the forward propagation, the activations of each layer are computed and memorized, then the derivatives are computed backwards layer by layer, starting from the output down to the input. Once computed, derivatives are saved in memory so that they can be used when they appear in the chain rule formula for the derivative of another parameter. The backpropagation algorithm for a feedforward neural network is described in detail in algorithm 1. The feedforward neural network case is very simple, but useful to understand the working principles of the algorithm. The actual implementations are more sophisticated: they are designed to work with generic computational graphs with tensor-valued nodes, in order to work with many different types of neural networks. Some machine learning libraries (e.g. `Theano` [37] or `TensorFlow` [27]) compute the derivatives symbolically, by building a computational graph of the derivatives from the graph of the model. Then, the back-propagation algorithm could be applied recursively to obtain derivatives of higher order, even though this may be computationally unfeasible.

Data: Training example $(\mathbf{x}, \mathbf{y})$, loss function $L(\cdot, \mathbf{y})$
Result: Partial derivatives of the loss function with respect to the parameters of the network: $\nabla_{\mathbf{b}_i} L, \nabla_{\mathbf{K}_i} L \quad \forall i \in \{1, ..., N+1\}$, where N is the number of hidden layers.
Rename the input vector: $\mathbf{h}_0 \leftarrow \mathbf{x}$;
for $i \leftarrow 1$ **to** $N+1$ **do**
  Forward-propagate the input through the network;
  $\mathbf{h}_i \leftarrow \mathbf{K}_i \cdot \mathbf{h}_{i-1} + \mathbf{b}$;
  $\mathbf{a}_i \leftarrow \mathbf{f}_i(\mathbf{h}_i)$;
end
Rename the prediction: $\hat{\mathbf{y}} \leftarrow \mathbf{a}_{N+1}$;
Compute the gradient of the loss w.r.t the prediction: $\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y})$;
for $i \leftarrow N+1$ **down to** 1 **do**
  Back-propagate the derivatives through the network;
  $\mathbf{g} \leftarrow \left(\frac{\partial \mathbf{f}_i}{\partial \mathbf{h}_i}\right) \cdot \mathbf{g}$;
  $\nabla_{\mathbf{b}_i} L \leftarrow \mathbf{g}$;
  $\nabla_{\mathbf{K}_i} L \leftarrow \mathbf{g} \cdot \mathbf{h}_{i-1}^T$;
  $\mathbf{g} \leftarrow \mathbf{K}_i^T \cdot \mathbf{g}$;
end

Algorithm 1: Back-propagation algorithm for feedforward neural networks with loss functions without a parameter norm penalty. The activation function of the layer i is denoted by $\mathbf{f}_i(\cdot)$. If there is a penalty $P(\boldsymbol{\theta})$, the value $\nabla_{\boldsymbol{\theta}} P(\boldsymbol{\theta})$ must be added to the derivatives.

A.2 Optimization algorithms

The simplest gradient-based optimization algorithm is the **gradient descent (GD)**, or steepest descent. In practice, it is an iterative algorithm that, given a starting value for the parameters, updates the parameters to move against the gradient. The GD is described in detail in algorithm 2. The **learning rate** parameter calibrates

Data: Loss function $L(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y})$, training set $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1,...,N}$, initial parameters $\boldsymbol{\theta}_0$, number of iteration T, learning rate γ.
Result: Candidate for a local minimum $\boldsymbol{\theta}_{min}$ of $L(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y})$.
Set initial parameters: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_0$;
for $i \leftarrow 1$ **to** T **do**
  Estimate the gradient: $\mathbf{g} \leftarrow \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L(\mathbf{f}(\mathbf{x}_i; \boldsymbol{\theta}), \mathbf{y}_i)$;
  Update the parameters: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \mathbf{g}$;
end

Algorithm 2: Gradient descent

the step size of each update. It is critical for the performance of the algorithm: if it is too low, the algorithm may require too many iterations to converge; if it is too high, it may escape from minimums and fail to converge. The gradient descent is very simple, and works great for convex functions, but in general gets stuck easily in local minimums. In practice, the gradient descent is replaced by the **stochastic gradient descent**, in which the loss is computed only on a randomly picked batch

of training examples. This randomness can help to jump out of local minimums, because the parameters follow a noisy estimation of the gradient, instead of the gradient itself. Moreover, the computational cost of each update depends on the batch size, not on the training set size, and this could be useful for very large datasets. Other more elaborated variations of the stochastic gradient descent have been designed: for example, one could add a **momentum**, i.e. replacing the gradient with an exponential moving average of the gradients computed at all steps. The term "momentum" is used because this variation has an interpretation in physics: the algorithm consists on solving the Newton's equation in the parameter space for a particle with mass one and initial position given as input. There are two forces: one proportional to the gradient of the loss function, and one proportional to the velocity, i.e. a viscous drag. The equation is solved by means of the **Euler method**, which is the simplest numerical method for solving ordinary differential equations. The SGD with momentum is presented in algorithm 3. In addition to

Data: Loss function $L(\mathbf{f}(\mathbf{x};\boldsymbol{\theta}),\mathbf{y})$, training set $\{(\mathbf{x}_i,\mathbf{y}_i)\}_{i=1,\dots,N}$, batch size m, initial parameters $\boldsymbol{\theta}_0$, number of iteration T, learning rate γ, decay rate ρ.
Result: Candidate for a local minimum $\boldsymbol{\theta}_{min}$ of $L(\mathbf{f}(\mathbf{x};\boldsymbol{\theta}),\mathbf{y})$.
Set initial parameters: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_0$;
Set initial velocity: $\mathbf{v} \leftarrow \mathbf{0}$;
for $i \leftarrow 1$ **to** T **do**
  Sample a random minibatch of m examples from the training set $\{(\overline{\mathbf{x}}_i,\overline{\mathbf{y}}_i)\}_{i=1,\dots,m}$;
  Estimate the gradient: $\mathbf{g} \leftarrow \frac{1}{m}\sum_{i=1}^{m}\nabla_{\boldsymbol{\theta}} L(\mathbf{f}(\overline{\mathbf{x}}_i;\boldsymbol{\theta}),\overline{\mathbf{y}}_i)$;
  Update the velocity: $\mathbf{v} \leftarrow (1-\rho)\mathbf{v} - \gamma\mathbf{g}$;
  Update the parameters: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}$;
end

Algorithm 3: Stochastic Gradient Descent with momentum

momentum, many modern algorithms implement some routines for the **adaptation of the learning rate** at each iteration. For example, in SGD the learning rate may be a decreasing function of the iteration index. Other algorithms (e.g. AdaGrad [38], RMSProp [39], Adam [40]) introduce more sophisticated methods: for example, the **AdaGrad** algorithm at each iteration accumulates the squared values of the gradient $\mathbf{s} \leftarrow \mathbf{s} + \mathbf{g} \odot \mathbf{g}$ ($\odot$ is the element-wise multiplication) and then replaces the learning rate γ with $\gamma/(\sqrt{\mathbf{s}} + \delta)$, where the division and the square root are computed element-wise, and δ is just a regularization term to prevent divergence. Then, the algorithm will make larger steps for parameters whose derivative is often small, and smaller steps for parameters with large derivatives. The full history of the derivatives is used, and this could be a problem for non-convex losses where complicated structures may arise. The **RMSProp** is a variation of the AdaGrad where the damping factor $\mathbf{s}$ is not just the accumulation of the squared values of the gradient, but an exponential moving average $\mathbf{s} \leftarrow (1-\rho)\mathbf{s} + \rho\mathbf{g} \odot \mathbf{g}$. Finally, the **Adam** ("adaptive momentum") algorithm is a combination of RMSProp and momentum with some technical subtleties.

Appendix B

Sequential Model-Based Global Optimization

Sequential Model-Based Global Optimization (SMBO, also known as Bayesian Optimization) algorithms are a class of algorithms for the minimization of functions $f : X \to \mathbb{R}$ where each evaluation is very expensive. They are iterative algorithms based on replacing the loss function f by a **surrogate** function $\overline{f} : X \to \mathbb{R}$ which is easier to manage: with this surrogate function $\overline{f}_i$ the algorithm proposes a new search point $\mathbf{x}_{i+1}$, $f(\mathbf{x}_{i+1})$ is computed, and then the surrogate function is updated or recomputed to approximate better the true loss function. An example of Sequential Model-Based Global Optimization algorithm is presented in algorithm 4. The criterion $L(\mathbf{x}, \overline{f})$ and the model for the surrogate function depend on the specific algorithm. An example of SMBO algorithm is the **Tree-structured Parzen Estimator**, presented in section B.1. The surrogate function is updated at each evaluation because the optimization cost is dominated by the evaluation of each point, so it is critical to process all available information about the true loss function before investing resources into a new evaluation. However, the algorithm can be parallelized: the efficiency will be lower, and the algorithm will be biased because there will be more statistics near points that are faster to evaluate, but it may be worth it due to the speed up factor.

Data: loss function f, initial surrogate $\overline{f}_0$, number of trials T
Result: Candidate $\mathbf{x}_{best}$ for the minimum of f
Set trials history $H = \emptyset$;
for $i = 1$ **to** T **do**
  $x^\star \leftarrow \text{argmin}_{\mathbf{x}} L(\mathbf{x}, \overline{f}_{i-1})$;
  Compute $f(x^\star)$;
  $H \leftarrow H \cup \{\mathbf{x}^\star, f(\mathbf{x}^\star)\}$;
  Model a new surrogate function f_i using H;
end

Algorithm 4: A generic SMBO algorithm.

B.1 Tree-structured Parzen Estimator

The Tree-structured Parzen Estimator (TPE) algorithm [12] is a SMBO algorithm designed for tree-structured search spaces, in particular hyperparameter spaces in machine learning algorithms. It implements the surrogate model as a probabilistic model $p(y|\mathbf{x})$, then chooses the next trial point by optimizing the **Expected Improvement** criterion [41]. The Expected Improvement criterion $\mathrm{EI}_{y^\star}(\mathbf{x})$ measures how much the loss function is expected to be lower than a threshold value $y^\star$:

$$\mathrm{EI}_{y^\star}(\mathbf{x}) = \int_{-\infty}^{+\infty} \max\left(y^\star - y, 0\right) p(y|\mathbf{x}) dy \tag{B.1}$$

The threshold value $y^\star$ is chosen so that $p(y < y^\star) = \gamma$, where γ is a parameter of the algorithm. By the way, the TPE does not model $p(y|\mathbf{x})$, but attempts to model $p(\mathbf{x}|y)$ and then calculates $p(y|\mathbf{x})$ via Bayes' theorem. The distribution $p(\mathbf{x}|y)$ is modelled using two different densities:

$$p(\mathbf{x}|y) = \begin{cases} l(\mathbf{x}) & \text{if } y < y^\star \\ g(\mathbf{x}) & \text{if } y \geq y^\star \end{cases} \tag{B.2}$$

where $l(\mathbf{x})$ and $g(\mathbf{x})$ are probability distributions estimated by using the trials $\mathbf{x}_i$ such that $f(\mathbf{x}_i)$ is respectively lower and higher or equal than $y^\star$. It remains to define how the algorithm estimates a probability distribution given some trials $\{\mathbf{x}_1, ..., \mathbf{x}_K\}$. The components of the search space could be continuous or discrete: the user specifies a prior over each component, e.g. uniform or log-uniform in a range for continuous components, categorical for discrete components, etc. The algorithm estimates a distribution function by replacing these priors with some adaptive densities. For example, if the component is continuous with uniform prior in a finite range, the algorithm replaces that prior with a truncated Gaussian Mixture Model where each Gaussian is centered in one observation, and where the standard deviations are set to the maximum distance from the nearest neighbours (the endpoints of the search range are considered neighbours). If the component is categorical, the algorithm will reweight each category depending on its frequency. The Expected Improvement for the TPE admits a closed form solution:

$$\mathrm{EI}_{y^\star}(\mathbf{x}) = \int_{-\infty}^{+\infty} \max\left(y^\star - y\right) \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})} dy = ... \propto \left(\gamma + \frac{g(\mathbf{x})}{l(\mathbf{x})}(1-\gamma)\right)^{-1} \tag{B.3}$$

Thus, maximizing the Expected Improvement is equivalent to minimizing $g(\mathbf{x})/l(\mathbf{x})$. So, the algorithm proposes a new point by minimizing $g(\mathbf{x})/l(\mathbf{x})$, evaluates that point, recomputes the distributions $l(\mathbf{x})$ and $g(\mathbf{x})$, and so on.

This algorithm was designed for tuning the hyperparameters of a machine learning model. For a brief introduction to the hyperparameter tuning problem, see subsection 3.1.2. For an implementation with the `Hyperopt` package, see subsection 3.4.1. This section presents the algorithm as described in [12], but some modifications are possible. For example, in [42] the parameter γ varies in time in order to fix the ratio between the number of trials used to sample $l(\mathbf{x})$ and $g(\mathbf{x})$, respectively. Moreover, they downweighted the trials as they progressively get old.

Appendix C

Covariance Matrix Adaptation - Evolution Strategy

The Covariance Matrix Adaptation - Evolution Strategy [43] is a stochastic algorithm for the minimization of non-linear, non-convex functions defined on continuous search spaces. It is an **evolutionary** algorithm, i.e. inspired by the principles of biological evolution. It is an iterative algorithm: at each iteration (called **generation**) a set of points x_i, $i = 1, ..., N$ is sampled from a certain distribution: these points are the individuals of the population of that generation. This distribution is a multivariate Gaussian: at each generation, the parameters of this Gaussian are updated, and the next generation is sampled with these new parameters. So, the sampling step reads

$$\mathbf{x}_i^{(g+1)} \sim N\left(\mathbf{m}^{(g)}, \sigma_{(g)}^2 \mathbf{C}^{(g)}\right) \tag{C.1}$$

Once defined the initial parameters $\mathbf{m}^{(0)}$, $\sigma^{(0)}$ and $\mathbf{C}^{(0)} = \mathbf{I}$, it remains to define the update of the parameters at each generation. The goal is to adapt the parameters so that the populations become better and better, where "better" means with lower values of f.

1. The update of $\mathbf{m}$ is the easiest one: the individuals $\mathbf{x}_i$ are sorted by **fitness**, i.e. by increasing values of $f(\mathbf{x}_i)$ (the notation for sorted individuals is $\mathbf{x}_{i:N}$). A sort of natural selection is implemented by using only the first $j < N$ individuals of the population. Then, the update reads

$$\mathbf{m}^{(g+1)} = (1 - l_m)\mathbf{m}^{(g)} + l_m \sum_{i=1}^{j} w_i \mathbf{x}_{i:N}^{(g+1)} \tag{C.2}$$

   where l_m is the **learning rate** that controls the exponential decay of the contribution of older generations (it is the inverse of the decay time), and w_i, $i = 1, ..., j$, are decreasing positive weights such that $\sum_{i=1}^{j} w_i = 1$. This implements a sort of weighted recombination of the selected individuals.

2. The covariance matrix $\mathbf{C}$ update involves two different contributions, one of rank j and one of rank one. The former is similar to the previous step, and consists of estimating the covariance of the selected individuals via

$$\sum_{i=1}^{j} w_i \mathbf{y}_{i:N}^{(g+1)} \left(\mathbf{y}_{i:N}^{(g+1)}\right)^T \tag{C.3}$$

where $\mathbf{y}^{(g+1)} = (\mathbf{x}^{(g+1)} - \mathbf{m}^{(g)})/\sigma^{(g)}$ is the distance from the mean in step size unit. Notice that the rank-j update involves only the current generation. In contrast, the rank-one update involves all generations. Let define an evolution path $\mathbf{p}_c$ that cumulates the progression of the mean $\mathbf{m}$ of each generation in step size unit:

$$\mathbf{p}_c^{(g+1)} = (1 - l_c)\mathbf{p}_c^{(g)} + \frac{1}{\text{norm.}} \sum_{i=1}^{j} w_i \mathbf{y}_{i:N}^{(g+1)} \tag{C.4}$$

where l_c is another learning rate, that makes the contribution of the previous generations decay exponentially with a decay time of $1/l_c$. Then, the rank-one update reads

$$\mathbf{p}_c^{(g+1)} \left(\mathbf{p}_c^{(g+1)}\right)^T \tag{C.5}$$

This contribution increases the probability to sample new individuals in the direction of the evolution path. By combining these two contributions, the update of the covariance matrix reads

$$\mathbf{C}^{(g+1)} = (1 - l_1 - l_j)\mathbf{C}^{(g)} + l_1 \mathbf{p}_c^{(g+1)} \left(\mathbf{p}_c^{(g+1)}\right)^T + l_j \sum_{i=1}^{j} w_i \mathbf{y}_{i:N}^{(g+1)} \left(\mathbf{y}_{i:N}^{(g+1)}\right)^T \tag{C.6}$$

where l_1, l_j are the learning rate of the two updates.

3. The step size σ update involves the use of an evolutionary path $\mathbf{p}_\sigma$ similar to $\mathbf{p}_c$, but with the contribution of each generation multiplied by $\mathbf{C}^{-\frac{1}{2}}$:

$$\mathbf{p}_\sigma^{(g+1)} = (1 - l_\sigma)\mathbf{p}_\sigma^{(g)} + \frac{1}{\text{norm.}} \mathbf{C}_{(g)}^{-\frac{1}{2}} \sum_{i=1}^{j} w_i \mathbf{y}_i^{(g+1)} \tag{C.7}$$

where $\mathbf{C}^{-\frac{1}{2}}$ is defined with the eigendecomposition[1] of $\mathbf{C}$. If $\mathbf{y}_i \sim N(\mathbf{0}, \mathbf{C})$, then $\mathbf{C}^{-\frac{1}{2}}\mathbf{y}_i \sim N(\mathbf{0}, \mathbf{I})$, so with this transformation $\mathrm{E}\|\mathbf{p}_\sigma\|$ does not depend on the direction of $\mathbf{p}_\sigma$ anymore. Then, under random selection and with the right normalization constant $\mathrm{E}\|\mathbf{p}_\sigma^{(g+1)}\| = \mathrm{E}\|N(\mathbf{0}, \mathbf{I})\|$. If $\mathbf{p}_\sigma$ is longer than expected, the steps have similar directions, and the step size should increase because many steps can be replaced by a single longer step. Instead, if it is shorter than expected, the steps tend to have opposite directions, so their effects cancel out, and the step size should decrease because the search requires more resolution. Finally, the step size update reads

$$\sigma^{(g+1)} = \sigma^{(g)} \cdot \exp\left(\text{const} \cdot \left(\frac{\|\mathbf{p}_\sigma^{(g+1)}\|}{\mathrm{E}\|N(\mathbf{0}, \mathbf{I})\|} - 1\right)\right) \tag{C.8}$$

This method is known as **cumulative step-size adaptation (CSA)**.

This is a simplified version of the algorithm: in the actual implementations there are some additional terms, and some steps are more elaborated. For a detailed description of the algorithm with an actual implementation, see [43]. An example of application of the CMA-ES is the minimization step of the `MCNNTUNES` procedure (see subsection 3.1.2). Another interesting application could be found in [44].

[1] If $\mathbf{C} = \mathbf{B}\mathbf{D}^2\mathbf{B}^T$, where $\mathbf{B}$ is orthogonal and $\mathbf{D}$ is diagonal (with the square roots of the eigenvalues of $\mathbf{C}$ as elements), then $\mathbf{C}^{-\frac{1}{2}} = \mathbf{B}\mathbf{D}^{-1}\mathbf{B}^T$.

Bibliography

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[2] Trevor Hastie et al. "The elements of statistical learning: data mining, inference and prediction". In: *The Mathematical Intelligencer* 27.2 (2005), pp. 83–85.

[3] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *arXiv e-prints* (Dec. 2013). arXiv: 1312.5602 [cs.LG].

[4] Andy Buckley et al. "Systematic event generator tuning for the LHC". In: *European Physical Journal C* 65.1-2 (Jan. 2010), pp. 331–357. DOI: 10.1140/epjc/s10052-009-1196-7. arXiv: 0907.2973 [hep-ph].

[5] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4.2 (1991), pp. 251–257. ISSN: 0893-6080. DOI: 10.1016/0893-6080(91)90009-T.

[6] Moshe Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". In: *Neural Networks* 6.6 (1993), pp. 861–867. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(05)80131-5.

[7] Zhou Lu et al. "The Expressive Power of Neural Networks: A View from the Width". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 6231–6239. URL: http://papers.nips.cc/paper/7203-the-expressive-power-of-neural-networks-a-view-from-the-width.pdf.

[8] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. 2001–. URL: http://www.scipy.org/.

[9] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[10] Christian Szegedy et al. "Intriguing properties of neural networks". In: *arXiv e-prints* (Dec. 2013). arXiv: 1312.6199 [cs.CV].

[11] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. "Explaining and Harnessing Adversarial Examples". In: *arXiv e-prints* (Dec. 2014). arXiv: 1412.6572 [stat.ML].

[12] James S. Bergstra et al. “Algorithms for Hyper-Parameter Optimization”. In: *Advances in Neural Information Processing Systems 24*. Ed. by J. Shawe-Taylor et al. Curran Associates, Inc., 2011, pp. 2546–2554. URL: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf.

[13] P. Nason, ed. *Frascati Physics Series - Proceedings of the Workshop on Monte Carlo's, Physics and Simulations at the LHC - Volume XLIX*. Istituto Nazionale di Fisica Nucleare - Laboratori Nazionali di Frascati, 2006. ISBN: 978-88-86409-58-2. URL: http://www.lnf.infn.it/sis/frascatiseries/Volume49/volume49.pdf.

[14] Torbjörn Sjöstrand, Stephen Mrenna, and Peter Skands. “PYTHIA 6.4 physics and manual”. In: *Journal of High Energy Physics* 2006.05 (May 2006), pp. 026–026. DOI: 10.1088/1126-6708/2006/05/026.

[15] Torbjörn Sjöstrand et al. “An introduction to PYTHIA 8.2”. In: *Computer Physics Communications* 191 (2015), pp. 159–177. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2015.01.024.

[16] Alec Aivazis. https://github.com/AlecAivazis/feynman.

[17] Andy Buckley et al. “Rivet user manual”. In: *Comput. Phys. Commun.* 184 (2013), pp. 2803–2819. DOI: 10.1016/j.cpc.2013.05.021. arXiv: 1003.0694 [hep-ph].

[18] Matt Dobbs and Jørgen Beck Hansen. “The HepMC C++ Monte Carlo event record for High Energy Physics”. In: *Computer Physics Communications* 134.1 (2001), pp. 41–46. ISSN: 0010-4655. DOI: 10.1016/S0010-4655(00)00189-2.

[19] DELPHI Collaboration. “Tuning and test of fragmentation models based on identified particles and precision event shape data”. In: *Zeitschrift für Physik C Particles and Fields* 73.1 (Mar. 1997), pp. 11–59. ISSN: 1431-5858. DOI: 10.1007/s002880050295.

[20] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org.

[21] F. James and M. Roos. “Minuit – a system for function minimization and analysis of the parameter errors and correlations”. In: *Computer Physics Communications* 10 (Dec. 1975), pp. 343–367. DOI: 10.1016/0010-4655(75)90039-9.

[22] iminuit team. *iminuit – A Python interface to Minuit*. https://github.com/scikit-hep/iminuit.

[23] Wojciech M. Czarnecki and Igor T. Podolak. “Machine Learning with Known Input Data Uncertainty Measure”. In: *Computer Information Systems and Industrial Management*. Ed. by Khalid Saeed et al. Berlin, Heidelberg: Springer, 2013, pp. 379–388. ISBN: 978-3-642-40925-7. DOI: 10.1007/978-3-642-40925-7_35.

[24] James Bergstra et al. “Hyperopt: a Python library for model selection and hyperparameter optimization”. In: *Computational Science & Discovery* 8.1 (July 2015), p. 014008. DOI: 10.1088/1749-4699/8/1/014008.

[25] S. van der Walt, S. C. Colbert, and G. Varoquaux. “The NumPy Array: A Structure for Efficient Numerical Computation”. In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 22–30. DOI: 10.1109/MCSE.2011.37.

[26] François Chollet et al. *Keras*. https://keras.io. 2015.

[27] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[28] Nikolaus Hansen, Youhei Akimoto, and Petr Baudis. *CMA-ES/pycma on Github*. Feb. 2019. DOI: 10.5281/zenodo.2559634.

[29] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[30] Michael Waskom et al. *mwaskom/seaborn: v0.9.0 (July 2018)*. July 2018. DOI: 10.5281/zenodo.1313201.

[31] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 51–56.

[32] ATLAS collaboration. "Measurement of the Z/γ^* boson transverse momentum distribution in pp collisions at $\sqrt{s} = 7$ TeV with the ATLAS detector". In: *Journal of High Energy Physics* 2014.9 (Sept. 2014), p. 145. ISSN: 1029-8479. DOI: 10.1007/JHEP09(2014)145.

[33] A. Banfi et al. "Optimisation of variables for studying dilepton transverse momentum distributions at hadron colliders". In: *The European Physical Journal C* 71.3 (Mar. 2011), p. 1600. ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-011-1600-y.

[34] ATLAS collaboration. "Measurement of angular correlations in Drell-Yan lepton pairs to probe Z/γ^* boson transverse momentum at $\sqrt{s} = 7$ TeV with the ATLAS detector". In: *Physics Letters B* 720.1-3 (Mar. 2013), pp. 32–51. DOI: 10.1016/j.physletb.2013.01.054. arXiv: 1211.6899 [hep-ex].

[35] Richard Corke and Torbjörn Sjöstrand. "Interleaved parton showers and tuning prospects". In: *Journal of High Energy Physics* 2011.3 (Mar. 2011), p. 32. ISSN: 1029-8479. DOI: 10.1007/JHEP03(2011)032.

[36] Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: *arXiv e-prints* (Dec. 2012). arXiv: 1212.5701 [cs.LG].

[37] Rami Al-Rfou et al. "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* (May 2016). arXiv: 1605.02688 [cs.SC].

[38] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159.

[39] Geoffrey Hinton. *Neural networks for machine learning*. Coursera, video lectures. 2012.

[40] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv e-prints* (Dec. 2014). arXiv: 1412.6980 [cs.LG].

[41] Donald R. Jones. "A Taxonomy of Global Optimization Methods Based on Response Surfaces". In: *Journal of Global Optimization* 21.4 (Dec. 2001), pp. 345–383. ISSN: 1573-2916. DOI: 10.1023/A:1012771025575.

[42] James Bergstra, Daniel Yamins, and David Cox. "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. 1. PMLR, 2013, pp. 115–123. URL: http://proceedings.mlr.press/v28/bergstra13.html.

[43] Nikolaus Hansen. "The CMA Evolution Strategy: A Tutorial". In: *arXiv e-prints* (Apr. 2016). arXiv: 1604.00772 [cs.LG].

[44] Thomas Geijtenbeek, Michiel van de Panne, and A. Frank van der Stappen. "Flexible Muscle-based Locomotion for Bipedal Creatures". In: *ACM Trans. Graph.* 32.6 (Nov. 2013), 206:1–206:11. ISSN: 0730-0301. DOI: 10.1145/2508363.2508399.