# Chapter 1

# Machine Learning

## 1.1 Supervised learning and optimization

Machine Learning is about having an algorithm improve through the use of *training data*, so that is it able to make predictions or decisions automatically, without any specific outcome being explicitly programmed in.

There are many kinds of problems this general approach can be used to solve, one of which is known as *supervised learning*. The idea is to start from a set of training data consisting of pairs $(x, y)$, where $x$ and $y$ are typically high-dimensional objects, and to train a system so that it is able to reconstruct $y_{\text{new}}$ from a given $x_{\text{new}}$ to within some tolerable margin of error.

Supervised learning differs from *unsupervised* learning, in which data is provided without labels of any sort, and the algorithm must seek some sort of structure in it.

The software **mb!** (**mb!**) uses both kinds: the **NN!** (**NN!**) is trained with supervised learning, while the **PCA!** (**PCA!**) dimensionality reduction technique and the downsampling are unsupervised.

### 1.1.1 Dataset management

The way a supervised learning system typically works is to build a model $y_{\text{pred}} = f(x; \alpha, \beta)$ which depends on variable parameters $\alpha$ and fixed hyperparameters $\beta$. For concreteness, we can think of polynomial regression: $\beta$ may then be the degree of the polynomial with which we fit the data, while $\alpha$ is the set of coefficients of this polynomial.

We can evaluate the performance of this model by comparing $y_{\text{pred}}$ with the known label $y$ for all the data in the training dataset. This allows us to compute some cost function $C(\alpha; \beta)$ — the choice of $C$ is not simple in general, let us keep it abstract for now.

Then, we can use some optimization procedure to find

$$\overline{\alpha} = \underset{\alpha}{\operatorname{argmax}}\, C(\alpha; \beta)\,. \tag{1.1.1}$$

This sounds good, but there is a problem: a procedure as described can get arbitrarily low costs by learning *specific features* of the training dataset which will not generalize. This is known as *overfitting*, and it is characterized by a low error in the training dataset but a high error on new data.

Its counterpart, *underfitting*, may happen when our model is not "detailed" enough to capture some feature of the data. Here, the model will not only perform badly on new data, but also on its training data.

The parameters $\beta$ can typically be adjusted to find an optimum between these two extremes. A concrete example is shown in figure 1.1: some noisy data is fitted with polynomials of differing degrees — this corresponds to minimizing $C(\alpha;\beta)$ for different values of $\beta$. Only the data in the central region is used for the fit, while the rest is interpreted as validation data.

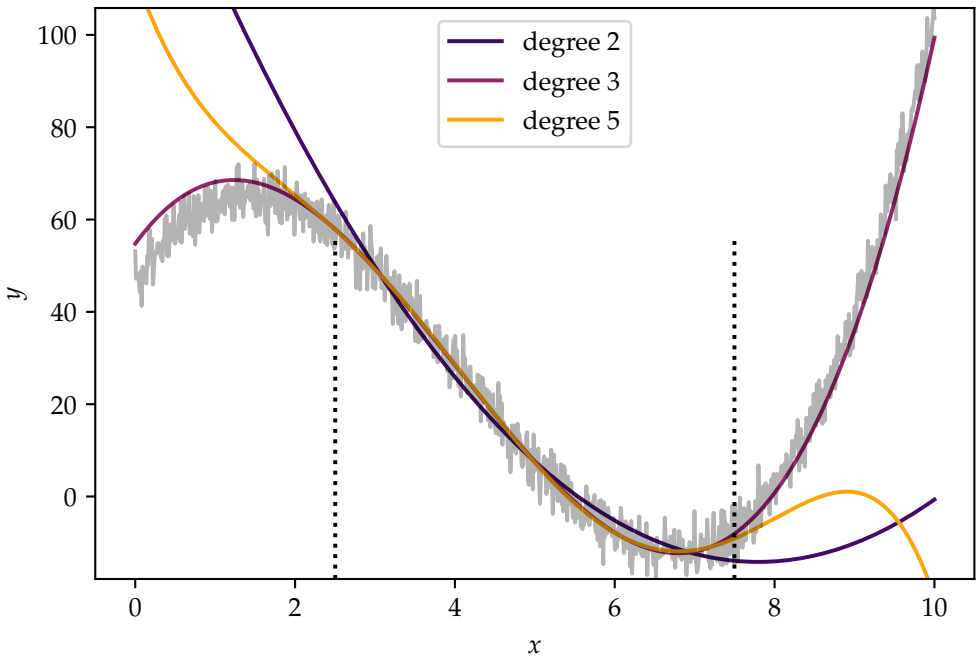


Figure 1.1: Demonstration of under- and over-fitting: the training data here are the central region between the two vertical lines; the models are then validated in the remaining part of the region. See figure 1.2 for the exact value of the training and validation errors at various polynomial degrees.

Note that this split of training and validation data, while visually clear, is undesirable in real circumstances: we can achieve much better accuracy if the training data spans every part of the region of interest. This is the case for acmb, since we can generate data at will.

If we denote the error on the validation data as $C_V(\alpha;\beta)$, the full procedure can then be schematically be written as

$$\overline{\beta} = \underbrace{\operatorname*{argmax}_{\beta}}_{\text{validation}} \; C_V \left( \underbrace{\operatorname*{argmax}_{\alpha} C(\alpha;\beta)}_{\text{training}};\; \beta \right) . \tag{1.1.2}$$

The best ways to calculate these two optimums differ, both among each other and on

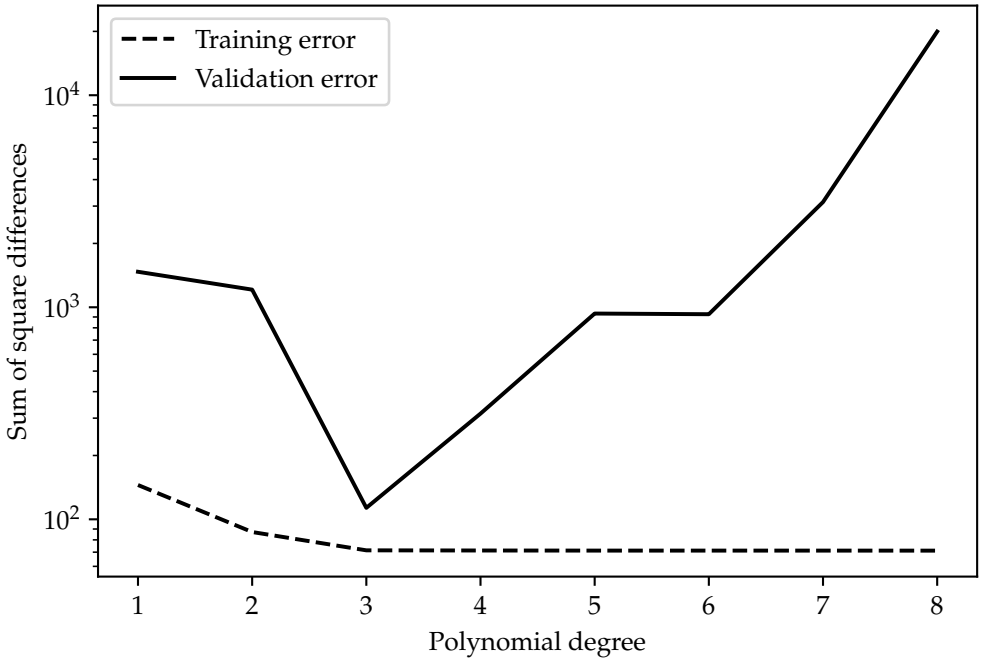


Figure 1.2: Training and validation errors for figure 1.1. The best choice for the degree here is clearly 3, since it minimizes the validation error as well as having a very low training error, but without the validation procedure we could not appreciate it being a better choice than higher values.

a case-by-case basis: the validation procedure must account for the fact that the training procedure for each value of $\beta$ analyzed can take some time.

In particular, the algorithm used in **mb!** for training is discussed in section 1.3.4, while the one used for validation is discussed in section 1.4.

The procedure described here makes use of a training and a validation dataset, but typically also we want to have some objective measure of the quality of the fit. This cannot be provided by only using training and validation data information, since the parameters of the model depend on them: we need a third separate *testing* dataset which does not inform the model in any way.[1]

## 1.2 Principal Component Analysis

After downsampling, a waveform used by **mb!** is described by several hundred points. It is convenient to reduce this number in order for the neural network to be faster. We are able to do so by making use of the fact that the components of the high-dimensional vector representing the waveform are correlated.

The technique of **PCA!** is quite general,[2] so let us describe it in general terms, and

[1] Confusingly, the **ML!** (**ML!**) literature sometimes inverts the labels "validation" and "testing" — we shall stick with the definition given above.

[2] This technique is quite old: it was developed in 1901 by Karl Pearson [Pea01; MN17], but it started to

then apply it to our specific problem.

Dimensionality reduction algorithms such as **PCA!** may be considered as a kind of unsupervised learning, since they can allow for the detection of relevant features in high-dimensional datasets. Here we will not use it for this purpose though, it will be more of a pre-processing step.

### 1.2.1 General method

We start with a dataset of $N$ points in $\mathbb{R}^D$, which we denote by $\{x^i\}_{i=0}^{N-1}$. We need $D$ floating point numbers to represent each of these points.

If we can find a $k$-dimensional hyperplane in $\mathbb{R}^D$, with $k \ll D$, such that our points are never very far from this subspace, we can substitute the $D$-dimensional parametrization of the points for a $k$-dimensional one by approximating each point by its orthogonal projection onto the $k$-dimensional hyperplane. We will make a certain error in this process: specifically, if $P_k(x_i)$ denotes the projection of the point onto this hyperplane, the error (computed according to the Euclidean distance among points) can be quantified by

$$\text{error}(k) = \sum_{i=0}^{N-1} \left\| x_i - P_k(x_i) \right\|^2 . \tag{1.2.1}$$

The algorithm of PCA allows us to determine which hyperplane minimizes this error.

The first step is to center the data: we compute their mean $\overline{x}$, and work with the dataset $y_i = x_i - \overline{x}$. Because of this, we can say that the $k$-dimensional hyperspace is now a *subspace* with respect to $y$. Computationally, we keep the mean $\overline{x}$ saved and add it to the reconstructed data $y$.

Let us now consider the $k = 1$ case: we want to project the data onto a single line, which we can parametrize as the span of a unit vector $w$. Therefore, what we want to minimize is $\sum_i \left\| y_i - (y_i \cdot w)w \right\|^2 = \sum_i \left( \left\| y_i \right\|^2 - (y_i \cdot w)^2 \right)$, and we can do so by maximizing $\sum_i (y_i \cdot w)^2$.

Therefore, the best 1-dimensional subspace is:

$$w = \underset{w \in \mathbb{S}^{D-1}}{\operatorname{argmax}} \sum_i (y_i \cdot w)^2 . \tag{1.2.2}$$

Now comes the clever idea of **PCA!**: we can reformulate this argmax problem as an eigenvalue problem for the covariance matrix of the data:

$$C = \frac{1}{N} \sum_i y_i y_i^\top . \tag{1.2.3}$$

A unit eigenvector $w$ of this matrix will satisfy $Cw = \lambda w$ for its eigenvalue $\lambda$, and we can recover the eigenvalue $\lambda$ from this equation by computing $w^\top C w = \lambda w^\top w = \lambda$; making the covariance matrix explicit allows us to see that

$$\lambda = w^\top C w = \frac{1}{N} \sum_i (y_i \cdot w)^2 ; \tag{1.2.4}$$

see broad use once availability of computers became widespread.

which is precisely the quantity we wanted to maximize: therefore, the best one-dimensional subspace is precisely the largest eigenvector of the covariance matrix, the direction of maximum variance.

If we make the further observation that the covariance matrix is symmetric and positive definite, and can therefore be orthogonally diagonalized, we are almost done: we can generalize to arbitrary $k$ moving one vector at a time. To find the second vector to span the subspace we can restrict ourselves to the subspace $w^{\perp}$ and apply the same procedure as before, this tells us that the optimal two-dimensional subspace is the span of the first two eigenvectors of the covariance matrix, and so on.

In order to perform a reduction onto a $k$-dimensional subspace, then, we need to calculate the unit eigenvectors $\{w_i\}_{i=0}^{k-1}$ corresponding to the $k$ largest eigenvalues; we can understand these as the columns of a $D \times k$ matrix $V$, which we can then use to construct the projection matrix onto the $k$-dimensional subspace.

In terms of the $D$-dimensional coordinates, the projection matrix is $VV^{\top}$: its application to a vector $y$ can be written as

$$P_k(y) = VV^{\top}y = \begin{bmatrix} w_1 & \cdots & w_k \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_k \end{bmatrix} y = \sum_{i=0}^{k-1} (w_i \cdot y) w_i \in \mathbb{R}^D . \tag{1.2.5}$$

For the purpose of dimensionality reduction, however, we are not interested in this projection, which still yields a $D$-dimensional vector: all the "reduced" vectors lie in a $k$-dimensional subspace, therefore we express them with only $k$ coordinates, by computing $V^{\top}y \in \mathbb{R}^k$.

If we approach the problem by diagonalizing the covariance matrix $C$, the computational complexity in the worst case scenario is $\mathcal{O}\left(D^3\right)$, since it involves the diagonalization of a $D \times D$ matrix.

### 1.2.2 PCA for waveforms

After downsampling and computing the residuals from the **PN!** (**PN!**) waveform, we are left with a waveform described by a vector of a few hundred points for the amplitude and similarly for the phase.

It would be possible to perform **PCA!** separately for these two vectors; however if we combine them into a single one we can exploit any existing correlation between the amplitude and phase residuals. In the worst case scenario — amplitude and phase residuals being completely uncorrelated — this procedure will perform exactly like separating the **PCA!** into two. So, we want to combine amplitude and phase residuals into a single vector.

A simple way to do so is to simply "append" one vector to the other, and therefore consider the waveform as a vector in $\mathbb{R}^{D_A + D_\varphi}$. There is an issue with this procedure: **PCA!** optimizes Euclidean distance, so the scaling of the amplitude and phase residuals becomes relevant. Fortunately, this carries no physical meaning: because of the way we define the residuals **??**, it is equivalent to a change in the basis for the logarithm.

Heuristically, we want the typical scale of the phase residuals to roughly match that of the amplitude residuals. This can be simply achieved by multiplying either part of the vector by a constant, which can be tuned in order to optimize the reconstruction error.

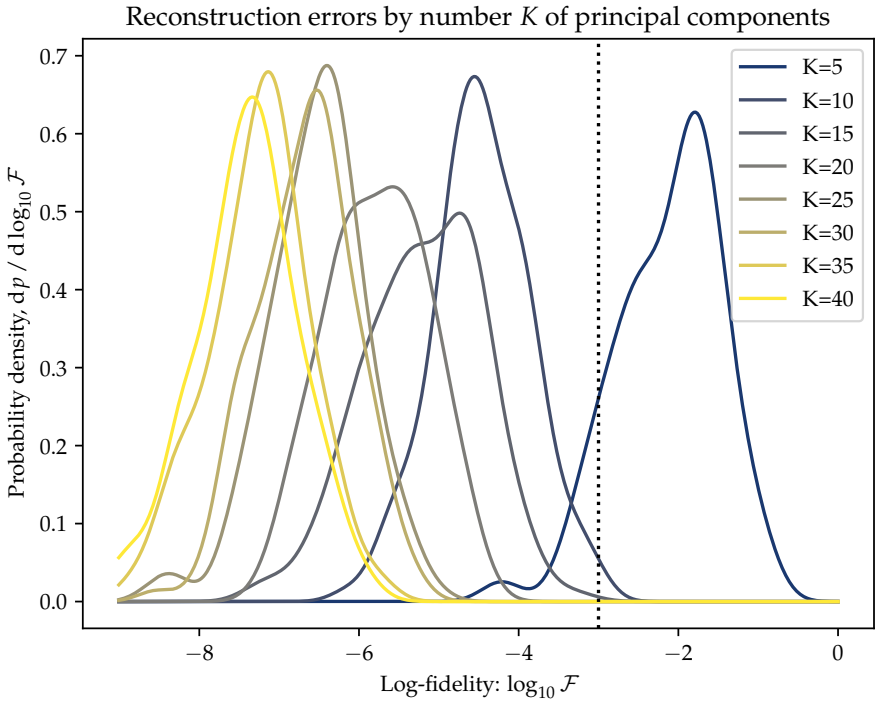


Figure 1.3: The reconstruction error is computed after projecting onto a $K$-dimensional subspace a number $N_w = 64$ waveforms. The parameters for the **PCA!** model are fitted with an independent, $N_{\mathrm{PCA}} = 128$ waveform dataset. A **KDE!** is performed for each set of corresponding log-fidelities, in order for the probability curve to look smoother. Here the **EOB!** waveforms are calculated with a **SPA!**. A vertical dashed line indicates a "desirable" maximal reconstruction error, $\mathcal{F} \sim 10^{-3}$, which roughly corresponds to the accuracy of the underlying **EOB!** model to **NR!** simulations.

## 1.3 Neural Networks

After the reduction of dimensionality through **PCA!**, we are left with the task of approximating the map between the parameters of the system generating the waveform, $\theta_i$, and the $k$ principal components.

In the work of Schmidt et al. [Sch+20] this was accomplished through a Mixture of Experts model, which amounts to a fit to a polynomial expression of the parameters; here instead we reconstruct the function with a neural network.

**NN!**s are known to be able to approximate arbitrary functions [Nie15], and in practice they are quite versatile and usually not prone to overfitting. In the next section we will describe the architecture of a multi-layer perceptron regressor, the kind of network used by **mb!**.

### 1.3.1 Multi-layer perceptrons

This architecture is built in order to solve the problem of reconstructing a map from an input $x_i \in \mathbb{R}^n$, to an output $y_i \in \mathbb{R}^k$.

We introduce a *hidden layer* between the input and the output. This consists of a

certain number $m$ of "neurons", which can be more or less activated as a function of the inputs. Specifically, the $j$-th neuron in the hidden layer will have an activation level given by the expression:

$$z_j = \sigma\left(w_{ji}^{(1)} x_i + b_j^{(1)}\right), \tag{1.3.1}$$

where $\sigma\colon \mathbb{R} \to \mathbb{R}$ is called the *activation function*, while the parameters $w_{ji}^{(1)}$ and $b_j^{(1)}$ are respectively called the *weights* and the *biases* (for the first hidden layer).

We shall discuss the reasons one might have for choosing different activation functions later; for now let us say that it is typically differentiable almost everywhere, and it achieves low values for low inputs and high values for high inputs. A common choice, for example, is a logistic sigmoid in the form $\sigma(z) = (1 + e^{-z})^{-1}$.

The weights and biases of the network are free parameters, real numbers which will need to be tuned by the training process.

Once the network has computed the activations $z_j$ for our single hidden layer, it can compute the activations for the output: in this last stage we use no activation function, and the output of the network is simply

$$y_\ell = w_{\ell j}^{(2)} z_j + b_\ell^{(2)} = w_{\ell j}^{(2)} \sigma\left(w_{ji}^{(1)} x_i + b_j^{(1)}\right) + b_\ell^{(2)}. \tag{1.3.2}$$

In the end, therefore, the number of free parameters of the network is $nm + m$ for the first layer and $mk + k$ for the second, so $m + k + m(n + k)$ in total.

Adding more layers is not conceptually different, it only amounts to applying the procedure described by equation 1.3.1 again to the result of the first layer, and so on; each time we use the activation function, except for the output layer.

The activation function is crucial for our network to be able to capture nonlinearities: if we were to remove it, with any number of layers the network would still be a linear function of the inputs.

The implementation we use for **mb!** is the one provided by `scikit-learn` [Ped+11]; specifically, we choose an `MLPRegressor`.[3]

### 1.3.2 Training

The network is able to reconstruct our function as long as the weights and biases are appropriately set: how do we train it to ensure this?

We need to assign a loss function to the output of the network. Here, simplicity trumps accuracy — the "true" error we might like to work with is given by the Wiener distance among waveforms reconstructed from their PCA components, but in order to efficiently train the network we need something easier to compute. The typical error chosen is quadratic in the Euclidean distance, since its analytical derivative is easy to compute:

$$\text{error}(y) \propto \sum_{\text{training data}} \left\| y_{\text{predicted}} - y_{\text{true}} \right\|^2. \tag{1.3.3}$$

We also add an error term in the form $\alpha \|W\|^2$, where $\alpha$ is a non-negative hyperparameter (typically chosen to be small) and $\|W\|^2$ is the L2 norm of the weight tensor:

[3] The documentation for this network can be found at the url https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html.

this term is known as a "regularizer", it penalizes complex models. The reason why the L2 norm specifically is often chosen is because of the simplicity and efficiency of its implementation.[4] The parameter $\alpha$ is optimized in the hyperparameter training procedure.

We are then using the Euclidean distance among the $y$s (which for us will be **PCA!** component vectors) as a measure of the performance of the network; this works well enough, but we can make an improvement by noticing that the first PCA components are responsible for more of the variance (and thus more of the distance) between data points. Therefore, we can improve the performance by having the network learn the distance among the rescaled

$$\mathrm{PC}_i \lambda_i^{\kappa}\,, \tag{1.3.4}$$

where $\lambda_i$ are the eigenvalues corresponding to the principal components, while $\kappa > 0$ is a hyperparameter. The prior distribution for $\kappa$ is a log-uniform one between $10^{-3}$ and 1.

### 1.3.3 Backpropagation

Once we have our cost function, we need a rule to change the weights $w_{ij}^l$ and biases $b_j^l$ of our network[5] according to the variation of the cost function. We would like to implement some sort of gradient descent algorithm, updating weights and biases by

$$\Delta w_{ij}^l = -\eta \frac{\partial C}{\partial w_{ij}^l} \qquad \text{and} \qquad \Delta b_j^l = -\eta \frac{\partial C}{\partial b_j^l}\,, \tag{1.3.5}$$

where $\eta$ is called the *learning rate*.

So, we need to compute the gradients $\partial C \Big/ \partial w_{ij}^l$ and $\partial C \Big/ \partial b_j^l$. The first idea one might have to do so is to approximate them to first order, doing something like

$$\frac{\partial C}{\partial w_{ij}^l} \approx \frac{C(w + \epsilon e_{ij}^l) - C(w)}{\epsilon}\,, \tag{1.3.6}$$

where $w$ is the full weight tensor, while $\epsilon e_{ij}^l$ represents a small increment to that particular weight, $e$ being a "unit tensor".

This strategy turns out to be unfeasible because of its computational complexity: the computation of the cost requires a full pass-through of the network, requiring at least $M$ floating point operations (additions or multiplications) where $M$ is the number of free parameters. This would need to be done to compute the update of each of these parameters, so the number of operations needed to perform a single step of the gradient descent for the full network would be at least $M^2$.

The backpropagation algorithm is a clever idea which allows us to compute the gradient with only a forward pass through the network followed by a backward pass, exploiting the chain rule and the way the weight matrices affect each other iteratively.

[4] In the backpropagation equations, which we will shortly introduce, this eventually translates to a penalty on the update of each weight proportional to the weight itself. The overhead needed in order to compute this is very small.

[5] We have added an index $l$ for the layer: so, $w_{ij}^l$ is the weight that the $j$-th neuron in the $l$-th layer gives to input $i$.

The algorithm can be summarized by the following equations [Nie15, chapter 2]:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{1.3.7}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{1.3.8}$$

$$\delta^l_j = \left( \sum_i w^{l+1}_{ij} \delta^{l+1}_i \right) \sigma'(z^l_j) \tag{1.3.9}$$

$$\delta^L_j = \frac{\partial C}{\partial a^L_j}, \tag{1.3.10}$$

where $a^l_j = w^l_{ji} z_i + b^l_j$ is the activation for the $j$-th neuron of layer $l$, while $z^l_j = \sigma(a^l_j)$ is the output of the activation function for $a^l_j$.

It is important to note that the backpropagation equations are written without the Einstein summation convention: "$x_j = y_j z_j$" means that the $j$-th component of the vector $x$ is calculated by multiplying the $j$-th components of the vectors $y$ and $z$. This means that we are not computing matrix products, but instead the element-wise Hadamard product.

Working backwards, the last equation tells us how to compute the error corresponding to the output layer, denoted with $L$. We can compute the derivative of the cost function with respect to the activation analytically if, for example, we are using a simple quadratic cost function as described in equation 1.3.3.[6]

The second-to-last equation tells us how to compute the error $\delta^l_j$ of a layer if we know the error of the following layer. It is an application of the chain rule. We compute the $\delta^l_j$ iteratively going backwards through the network, and having done so we can recover the derivative of the cost with respect to the weights and biases by combining the error $\delta^l_j$ with the activations of each layer, $a^{l-1}_k$, as described by the first two equations.

### 1.3.4 Stochastic gradient descent

The method for gradient descent described by equation 1.3.5 is still slow if we try to compute the gradient of the cost function by using all the training data we have. It turns out to be more efficient to compute the gradient by only looking at a single example or a small batch of them, chosen randomly: this idea is known as *stochastic gradient descent*.

This typically allows for much faster convergence of the training process.

The algorithm used for the training of the network in **mb!** is Adam [KB17; Rud16], short for "Adaptive Moment Estimation", as implemented in `scikit-learn` [Ped+11].

The general idea of this algorithm is, first, to not move directly in the direction of the gradient, but instead to keep a running, exponentially weighted average of it, and move in *that* direction. An example where we can imagine this could be useful is if the cost function landscape exhibits a "canyon", with a low slope in a long and narrow central region and steep walls. A direct move in the direction of the gradient might mean we "bounce" between the walls a lot without being able to settle in the middle, while averaging allows the direction of our movement to be smoothed and perhaps fall in the middle region.

[6] The expression in [Nie15] differs from this one since he applies the activation function to the last layer, which one should do for a classification algorithm but not for a regression algorithm.

The second aspect is to make the step-size adaptive, as opposed to it being strictly proportional to the gradient. A "signal-to-noise ratio" is estimated through the square of the gradient, and it is used to scale the step based on how confident we might be in it being meaningful or not.

Both of these exponential decays are regulated by a hyperparameter — $\beta_1$ and $\beta_2$ respectively, which are commonly set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$, but which can be optimized.

## 1.4 Hyperparameter optimization

Our network will depend on several hyperparameters, such as the number and size of the layers or the learning rate; we want to get as close as possible to the optimal choice of these for the reconstruction of the function mapping the binary system parameters to the principal components to be fast as well as accurate.

This optimization is accomplished through a **MOTPE!** (**MOTPE!**), as described in Ozaki et al. [Oza+20] and as implemented through the Optuna API [Aki+19].

Here we will summarize the mechanism through which a single objective **TPE!** (**TPE!**) works, as originally described in Bergstra et al. [Ber+11, section 4], since the generalization to the multi-objective case [Oza+20] is rather mathematically involved but not too conceptually dissimilar from the single-objective case.

We can abstract away the neural network as a function $f$ which, after being given a set of hyperparameters $\vec{x}$, outputs a cost $y$, which we want to minimize. The evaluation of $f$ is quite costly: it involves the training of the network and a full evaluation of its performance (in our case, the reconstruction of the waveforms and a computation of the Wiener distance to their true counterparts).

So, we want to find a value of $\vec{x}$ which minimizes $y$ with as few evaluations of $f$ as possible. The parameters $\vec{x}$ will be given certain prior distributions initially, from which their values will be drawn randomly. Let us then suppose we already have a set of observations $\{(\vec{x}_i, y_i)\}_i$, and we want to find the best possible new value of $\vec{x}$.

We choose a certain quantile $\gamma$, say 15%, and select a fraction $\gamma$ of the best observations we have. This allows us to find a $y^*$ such that $\mathbb{P}(y < y^*) = \gamma$.

Then, we approximate the probability density $p(\vec{x}|y)$ as follows:

$$p(\vec{x}|y) = \begin{cases} \ell(\vec{x}) & y < y^* \\ g(\vec{x}) & y \geq y^* \,. \end{cases} \tag{1.4.1}$$

We are condensing the $y$-dependence onto a binary choice between "good" observations, modelled by $\ell(\vec{x})$, and "bad" observations, modelled by $g(\vec{x})$. These two functions can then be estimated by making use of the observations we have in the $y < y^*$ and $y \geq y^*$ cases.

This, crucially, allows us to calculate the **expected improvement** associated with a certain parameter set $\vec{x}$:

There is no improvement if $y > y^*$.

$$\mathrm{EI}_{y^*}(\vec{x}) = \int_{-\infty}^{\infty} \max\left(y^* - y, 0\right) p(y|\vec{x})\, \mathrm{d}y \tag{1.4.2}$$

$$= \int_{-\infty}^{y^*} (y^* - y)\, p(y|\vec{x})\, \mathrm{d}y \tag{1.4.3}$$

$$= \int_{-\infty}^{y^*} (y^* - y) \underbrace{p(\vec{x}|y)}_{=\ell(\vec{x})} p(y) \frac{1}{p(\vec{x})} \, \mathrm{d}y \tag{1.4.4}$$

Used $p(x,y) = p(x|y)p(y) = p(y|x)p(x)$.

$$= \frac{\ell(\vec{x})}{\gamma\ell(\vec{x}) + (1-\gamma)g(\vec{x})} \underbrace{\int_{-\infty}^{y^*} (y^* - y) p(y) \, \mathrm{d}y}_{\text{independent of } \vec{x}} \tag{1.4.5}$$

Expanded $p(\vec{x})$ as $\int p(x|y)p(y)\,\mathrm{d}y$ and split the $\ell$ and $g$ cases.

$$\propto \frac{1}{\gamma + (1-\gamma)\frac{g(\vec{x})}{\ell(\vec{x})}}, \tag{1.4.6}$$

where the term we neglected can be interpreted as the average improvement over all choices of $\vec{x}$, which we hope is large but which we cannot affect with a good choice of $\vec{x}$.

What this tells us is the rather intuitive fact that we want to select points which are favored by the "good" distribution $\ell$ and not by the "bad" distribution $g$, so that $g/\ell$ is small, which means that the expected improvement will be large. The formula also says that we should prefer smaller values of $\gamma$, but it does not take into account the fact that we need a reasonably large selection of points in order to properly model $\ell(\vec{x})$ without a large error. The balance among these two contrasting desires will dictate our choice of $\gamma$.

Once we are able to compute the expected improvement, the task is simple: we just need to generate a selection of points $\vec{x}_i$ from the distribution $\ell(\vec{x})$, evaluate their expected improvement (or just $g/\ell$, really) and pick the one with the largest expected improvement as the one for which to evaluate the **NN!**.

The one aspect still missing in the algorithm is how to estimate a probability distribution from a small set of samples, and how to calculate a new sample from it. There are different ways to do so; the one implemented in the **TPE!** algorithm fits a gaussian mixture model to estimate $\ell(\vec{x})$ and $g(\vec{x})$ [BYC13, section 5].

### 1.4.1 Greedy downsampling

The residual waveforms generated with the **EOB!** / **PN!** models have on the order of a few times $10^5$ sampling points. This is too large a number to effectively use **PCA!** on: as we will see, it requires writing a covariance matrix which would have $\sim 10^{11}$ entries, way beyond the RAM of most computers.

Therefore, we need to start by applying a simple dimensionality reduction technique: downsampling, applied to the unwrapped phase and amplitude of waveforms (see section **??**) after the subtraction of the "baseline" **PN!** ones.

Doing so on a uniform grid, however, is suboptimal: there is more detail to be captured in some areas than in others. It is difficult to determine *a priori* which areas will be more relevant than others, therefore **mb!** implements a greedy algorithm which selects a set of indices which optimize the reconstruction accuracy, inspired by `romspline`, which was introduced by Galley and Schmidt [GS16].

The algorithm is detailed as algorithm 1; it includes certain improvements over `romspline`. First, it allows for a full training dataset as opposed to reconstruction of a single waveform. The points are greedily chosen until the error over the whole dataset is below the threshold. This allows us to be more confident of the fact that the indices are not capturing the specifics of a single waveform, but instead they are representative of the whole dataset.

**Algorithm 1** Greedy downsampling algorithm.

**Require:** $x_i$, $y_i^{(j)}$, tolerance $\epsilon$.
**Require:** "Seed" indices $I$
  Error $e \leftarrow \infty$
  **while** $e > \epsilon$ **do**
    **for** each $j$ **do**
      $I' \leftarrow \emptyset$
      $e_j \leftarrow 0$
      $y^{(j)}_{\mathrm{rec},i} \leftarrow$ interpolation of $x_I$, $y_I^{(j)}$
      **for** $k_1, k_2$ successive pair in $I$ **do**
        $n \leftarrow \mathrm{argmax}_{i \in [k_1,k_2]} \left| y^{(j)}_{\mathrm{rec},i} - y_i^{(j)} \right|$
        $e_j \leftarrow \max\left(e_j, \left| y^{(j)}_{\mathrm{rec},n} - y_n^{(j)} \right|\right)$
        **if** $\left| y_{\mathrm{rec},n} - y_n^{(j)} \right| > \epsilon$ **then**
          $I' \leftarrow I' \cup n$
        **end if**
        $I \leftarrow I \cup I'$
      **end for**
      $e \leftarrow \min(e, e_j)$
    **end for**
  **end while**
  **return** indices $I$

Whether this is actually happening can be checked with a validation dataset: the validation errors are shown in figure 1.4 as the size of the training dataset increases. The number of indices found corresponding to a fixed tolerance increases with the size of the training dataset, as can be seen in figure 1.5.

The size of the training dataset is limited by memory availability, more so than computational complexity: each training waveform must be fully kept in memory, with its $\sim 10^6$ floating point numbers, each needing 8 bytes of memory. Therefore, we need $\sim 10\,\mathrm{MiB}$ of memory per stored waveform, and a hundred of them will take up $\sim 1\,\mathrm{GiB}$ of storage.

The interpolation method choice is important: higher-order interpolants are slower but need less points. Here we use cubic splines, which have been found to be a good compromise for surrogate models [Lac+19]. As we shall see in section **??**, the interpolation accounts for the most significant part of the overhead **mb!** exhibits over the 3.5PN TaylorF2 approximant it uses (roughly 65 % of a 40 % overhead) in the case of high numbers of points to interpolate. This may seem like a reason to switch to a lower-order interpolation mechanism, but it is balanced by the fact that the lower number of points allows for a larger training dataset.

Care must also be put into the choice of the tolerances, since they have a secondary, perhaps unexpected effect: the greedy downsampling algorithm implemented in **mb!** yields different numbers of downsampling points for amplitude and phase based on how many are needed in order to get the training dataset below the tolerance, therefore the combined amplitude+phase vector will have $N_{\mathrm{amp}} + N_{\mathrm{phi}}$ components, where the numbers of points $N_{\mathrm{amp}}$ and $N_{\mathrm{phi}}$ depend on the tolerances $\epsilon_{\mathrm{amp}}$ and $\epsilon_{\mathrm{phi}}$.

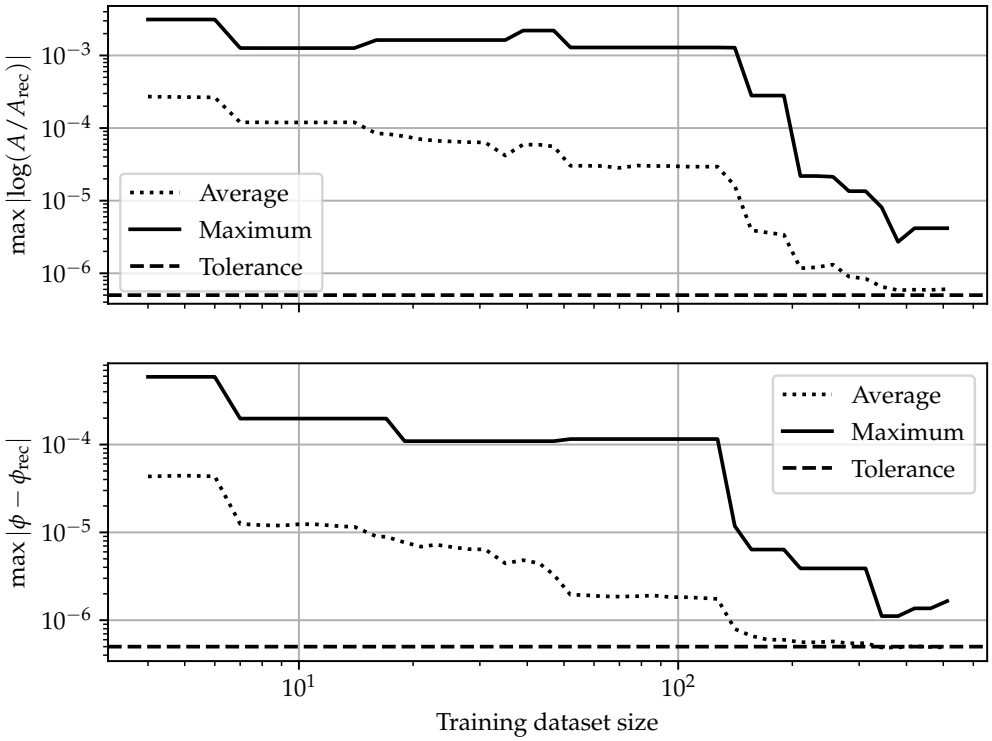


Figure 1.4: Validation errors for downsampled waveforms. Downsampling indices are calculated by applying algorithm 1 to a dataset of varying size, and the error is calculated by computing the maximum reconstruction error for each of $N_{\rm val} = 128$ waveforms. Note that "average" and "maximum" here refer to the difference between taking an average or a maximum over the $N_{\rm val}$-long array of maximum reconstruction errors per validation waveform. The validation waveforms are stochastically generated each time, hence the slight up-ticks one can see at certain points — still, the overall trend is downward.

There is no natural relation between $\epsilon_{\rm amp}$ ad $\epsilon_{\rm phi}$: one measures a variation in the logarithm of the amplitude, the other a variation in angle. However, there is more to the choice of these than just selecting the error we are willing to accept. The vectors whose distances are computed when doing **PCA!** will have more components for amplitude or for phase, and this will affect the reconstruction error, biasing the reconstruction towards one or the other.

Heuristically, it was found to be typically best to tune the tolerances so that the number of points describing the phase is slightly larger than the ones describing the amplitude.

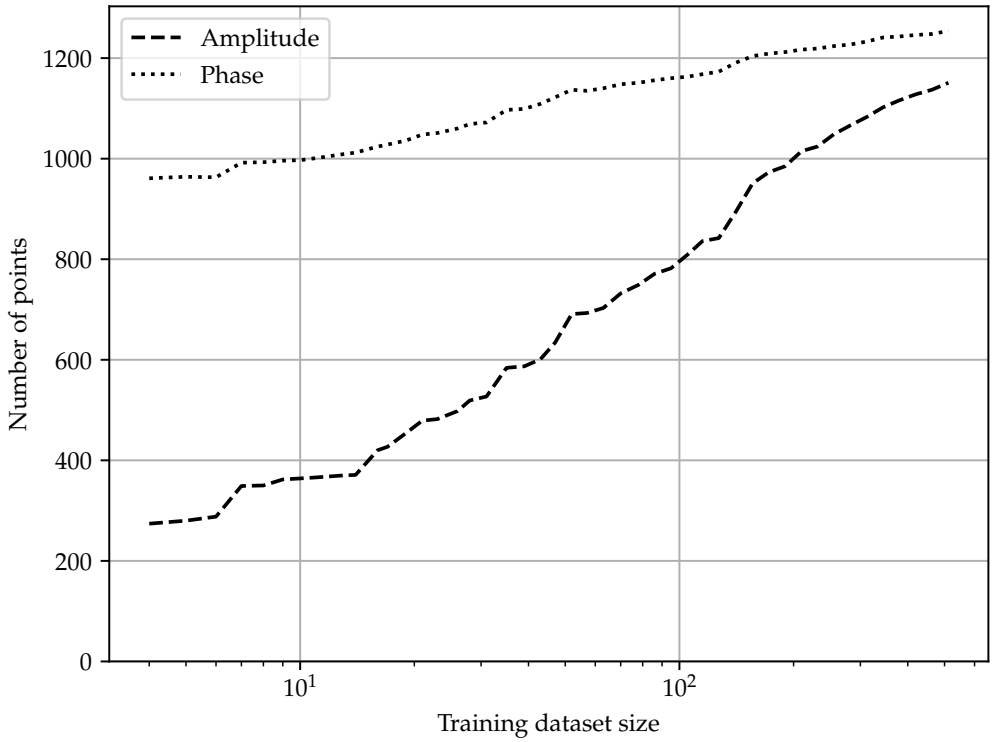


Figure 1.5: Size of the downsampling index array deriving from the application of algorithm 1 to training datasets of varying sizes.