1 Machine Learning

1.1 Numerical waveforms

1.1.1 Greedy approximants

Galley and Schmidt [GS16] introduced `romspline`.

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!** (**PCA!**) is quite general,[1] so let us describe it in general terms, and then apply it to our specific problem.

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[2]) can be quantified by

$$\text{error}(k) = \sum_{i=0}^{N-1} \left\| x_i - P_k(x_i) \right\|^2 . \tag{1.2}$$

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

[1] The utility and relative simplicity of this technique might make one think it quite old, while in fact it was developed only in 1901 by Karl Pearson [Pea01; MN17], ànd it started to see wider use once availability of computers became widespread.

[2] One might object here: the Euclidean distance among points is hardly relevant for our practical application! Fortunately, as we will later discuss, the PCA reconstruction of the points is efficient according to the Wiener distance as well as according to the Euclidean one. This might be understood heuristically by thinking of the fact that, when looking at waveforms which are quite close in terms of both distances, the linear approximation

$$\text{Wiener distance}(a_i, b_j) \approx \sqrt{g_{ij} a_i b_j} \tag{1.1}$$

for some metric g_{ij}, where a_i and b_j denote the vectors representing the two waveforms. Now, we do not know the precise form of g_{ij} (and, of course, we could not give a unique expression since it varies with detector noise), but as long as the metric is not pathological convergence in the Euclidean distance will imply convergence for this alternative distance.

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 $\sum_i \left\| y_i - (y_i \cdot w)w \right\|^2 = \sum_i \left(\|y_i\|^2 - (y_i \cdot w)^2 \right)$, and we can do so by maximizing $\sum_i (y_i \cdot w)^2$.

Therefore, the best 1-dimensional subspace is the direction of maximum variance:

$$w = \operatorname*{argmax}_{w \in \mathbb{S}^{D-1}} \sum_i (y_i \cdot w)^2 \,. \tag{1.3}$$

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

$$C = \frac{1}{N} \sum_i y_i y_i^\top \,. \tag{1.4}$$

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

$$\lambda = w^\top C w = \frac{1}{N} \sum_i (y_i \cdot w)^2 \,; \tag{1.5}$$

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

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

$$P_k(y) = VV^\top y = \begin{bmatrix} w_1 & \cdots & w_k \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_k \end{bmatrix} y = \sum_{i=0}^{k-1} (w_i \cdot y) w_i \in \mathbb{R}^D \,. \tag{1.6}$$

For the purpose of dimensionality reduction, however, we are not interested in this projection, which still yields a D-dimensional vector: this vector lies in a k-dimensional subspace, therefore we express it 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 have defined the residuals [INSERT REF], 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. It was found that, after using natural logarithms for the amplitude residuals, dividing the phase residuals by roughly 200 yields the best results.

[INSERT FIGURE FOR PCA RECONSTRUCTION ACCURACY]

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, θ_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, in $x_i \in \mathbb{R}^n$, to an output in $y_i \in \mathbb{R}^k$.

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:

$$z_j = \sigma\left(w^{(1)}_{ji} x_i + b^{(1)}_j\right), \tag{1.7}$$

where $\sigma\colon \mathbb{R} \to \mathbb{R}$ is called the *activation function*, while the parameters $w^{(1)}_{ji}$ and $b^{(1)}_j$ are respectively called the *weights* and the *bias* (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

$$y_\ell = w^{(2)}_{\ell j} z_j + b^{(2)}_\ell = w^{(2)}_{\ell j} \sigma\left(w^{(1)}_{ji} x_i + b^{(1)}_j\right) + b^{(2)}_\ell . \tag{1.8}$$

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 1.7 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:

$$\text{error}(y) \propto \sum_{\text{training data}} \left\| y_{\text{predicted}} - y_{\text{true}} \right\|^2 . \tag{1.9}$$

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

We also add an error term in the form $\alpha\|W\|^2$, where α is a non-negative hyperparameter (typically chosen to be small) and $\|W\|^2$ is the L2 norm of the weight tensor: this term is known as a "regularizer", it penalizes complex models. The parameter α is optimized in the hyperparameter training procedure, and its prior distribution is taken to be a log-uniform one between 10^{-7} and 10^{-5}.

We are then using the Euclidean distance among the ys (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

$$\mathrm{PC}_i \lambda_i^\kappa \,, \tag{1.10}$$

where λ_i are the eigenvalues corresponding to the principal components, while $\kappa > 0$ is a hyperparameter. The prior distribution for κ 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[4] according to the variation of the cost function. We would like to implement some sort of gradient descent algorithm, updating weights and biases by

$$\Delta w_{ij}^l = -\eta \frac{\partial C}{\partial w_{ij}^l} \qquad \text{and} \qquad \Delta b_j^l = -\eta \frac{\partial C}{\partial b_j^l} \,, \tag{1.11}$$

where η 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

$$\frac{\partial C}{\partial w_{ij}^l} \approx \frac{C(w + \epsilon e_{ij}^l) - C(w)}{\epsilon} \,, \tag{1.12}$$

where w is the full weight tensor, while ϵe_{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] 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]:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{1.13}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{1.14}$$

$$\delta^l_j = \left(\sum_i w^{l+1}_{ij} \delta^{l+1}_i \right) \sigma'(z^l_j) \tag{1.15}$$

$$\delta^L_j = \frac{\partial C}{\partial a^L_j}, \tag{1.16}$$

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 1.9.[5]

The second-to-last equation tells us how to compute the error δ^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 δ^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 δ^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 1.11 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

[5] 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.

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.

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.

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 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 γ, say 15%, and select a fraction γ 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:

$$p(\vec{x}|y) = \begin{cases} \ell(\vec{x}) & y < y^* \\ g(\vec{x}) & y \geq y^* \,. \end{cases} \tag{1.17}$$

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 $\vec{x}$:

$$\mathrm{EI}_{y^*}(\vec{x}) = \int_{-\infty}^{\infty} \max\left(y^* - y, 0\right) p(y|\vec{x})\, \mathrm{d}y \tag{1.18}$$

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

$$= \int_{-\infty}^{y^*} (y^* - y) p(y|\vec{x}) \, \mathrm{d}y \tag{1.19}$$

$$= \int_{-\infty}^{y^*} (y^* - y) \underbrace{p(\vec{x}|y)}_{=\ell(\vec{x})} p(y) \frac{1}{p(\vec{x})} \, \mathrm{d}y \tag{1.20}$$

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

$$= \frac{\ell(\vec{x})}{\gamma\ell(\vec{x}) + (1-\gamma)g(\vec{x})} \underbrace{\int_{-\infty}^{y^*} (y^* - y) p(y) \, \mathrm{d}y}_{\text{independent of } \vec{x}} \tag{1.21}$$

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

$$\propto \frac{1}{\gamma + (1-\gamma)\frac{g(\vec{x})}{\ell(\vec{x})}} , \tag{1.22}$$

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 ℓ and not by the "bad" distribution g, so that g/ℓ is small, which means that the expected improvement will be large. The formula also says that we should prefer smaller values of γ, 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 γ.

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/ℓ, really) and pick the one with the largest expected improvement as the one for which to evaluate the **NN!** (**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 Gaussian Mixture Model to estimate $\ell(\vec{x})$ and $g(\vec{x})$ [BYC13, section 5].