1 Machine Learning

1.1 Principal Component Analysis

After downsampling, a waveform used by `mlgw_bns` 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.

1.1.1 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.

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 $\sum_i (y_i \cdot w)^2$.

[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.

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 the

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.

1.1.2 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]

1.2 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 `mlgw_bns`.

1.2.1 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:

$$z_j = \sigma\left(w_{ji}^{(1)} x_i + b_j^{(1)}\right), \tag{1.7}$$

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 *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

$$y_\ell = w_{\ell j}^{(2)} z_j + b_\ell^{(2)} = w_{\ell j}^{(2)} \sigma\left(w_{ji}^{(1)} x_i + b_j^{(1)}\right) + b_\ell^{(2)} . \tag{1.8}$$

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 `mlgw_bns` is the one provided by `scikit-learn` [Ped+11].

1.2.2 Training

The network captures