Eötvös Loránd University

Faculty of Informatics

Department of Numerical Analysis

Unfolding the Variable Projection Method

Supervisor:
Dr. Gergő Bognár
Assistant Professor

Author:
András Salamon
Computer Science MSc

Budapest, 2023

Contents

Chapter 1

Introduction

Neural networks have revolutionized the field of machine learning in the recent decade, and have enabled significant advancements in fields such as computer vision, natural language processing and autonomous systems. They have shown remarkable success in a wide range of applications, from speech recognition to game playing, and have even surpassed human-level performance in some cases.

However, one of the main challenges associated with neural networks is their *black box* nature, which refers to the difficulty in understanding how these models arrive at their decisions. This lack of explainability is a significant issue in fields where interpretability is crucial, such as healthcare. Given that incorrect predictions can have severe consequences, having an understanding of the reasons behind them can assist professionals in identifying and rectifying such erroneous predictions.

An emerging technique called *unfolding* or *unrolling* provides a method of creating new neural network architectures that are inherently explainable because of their connection to iterative algorithms. This approach has several advantageous properties and has the potential to pave the way for the development of more robust and reliable models in the future. However, it is not a one-size-fits-all technique, and each iterative algorithm unfolding has its own set of challenges.

In my thesis, I explore the application of the technique of unfolding to the Variable Projection method [1]. I describe the necessary components for successful unfolding and verify the network's performance on two classification tasks. I used the PyTorch Python framework for my implementation.

Chapter 2

Fundamentals

This chapter covers the fundamentals of deep learning and neural networks that are necessary for understanding the rest of the thesis.

2.1 Machine learning

Machine learning is a powerful approach to address multidimensional prediction problems in a data-driven manner. Unlike traditional mathematical or statistical models, machine learning models are built based on observed data, enabling efficient solutions to complex prediction tasks. While classification and regression are common examples, machine learning encompasses a wide range of tasks. For instance, in classification, the goal is to assign discrete class labels to data instances, such as classifying images of various animals. On the other hand, regression focuses on predicting continuous or numerical values, like estimating housing prices based on factors such as area, location, and number of rooms. These examples highlight the versatility of machine learning, which finds applications in numerous domains due to its ability to tackle challenging prediction problems that would otherwise be difficult or impractical to solve.

In supervised learning tasks, the data is presented as (x, y) pairs, where the objective is to find a function $f(x) = \hat{y}$ that approximates the relationship between the input x and the corresponding output y. The process of arriving at such a function is called learning. The function f is a parameterized function with parameter vector θ, and the goal of training is to find the best values for θ, such that

$\hat{y} = f(x, \theta)$ is a good approximation of y. In order to measure how *close* $\hat{y}$ really is to y, a metric function is needed, commonly referred to as the loss function.

More precisely, a training dataset of m samples is defined as:

$$\{(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)\}, \quad m \in \mathbb{N},$$

where $x_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}^k$, $m, n, k \in \mathbb{N}\,(i = 1, 2, ..., m)$. The goal of training is to find an optimal parameter vector $\theta^* \in \mathbb{R}^q$, $q \in \mathbb{N}$, such that

$$\theta^* = \operatorname*{argmin}_{\theta} L(f(x, \theta), y),$$

where $f : \mathbb{R}^n \to \mathbb{R}^k$ is the hypothesis function, and $L : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}$ is the loss function.

2.2 Feedforward neural networks

Feedforward neural networks (*FFNs*) are the most foundational deep learning models [2]. These are called *feedforward* because information flows through the networks from the input to the output without any feedbacks. When networks contain feedbacks in their intermediate computations they are called recurrent neural networks. FFNs are represented by composing together many functions, where each function is called a layer. The first and last layers are called input and output layers, and anything in between is called a hidden layer (Figure 2.1). A layer consists of functions called *artificial neurons*. An artificial neuron with weights $w \in \mathbb{R}^n$, $n \in \mathbb{N}$, bias $b \in \mathbb{R}$ and activation function $g : \mathbb{R} \to \mathbb{R}$ is defined as the function $f : \mathbb{R}^n \to \mathbb{R}$

$$f(x) = g\left(\sum_{i=1}^{n} x_i w_i + b\right)$$

The activation function can be any nonlinear differentiable function, common choices include the $ReLU(x) = max(0, x)$ and the $sigmoid(x) = \frac{1}{1+e^{-x}}$ functions.

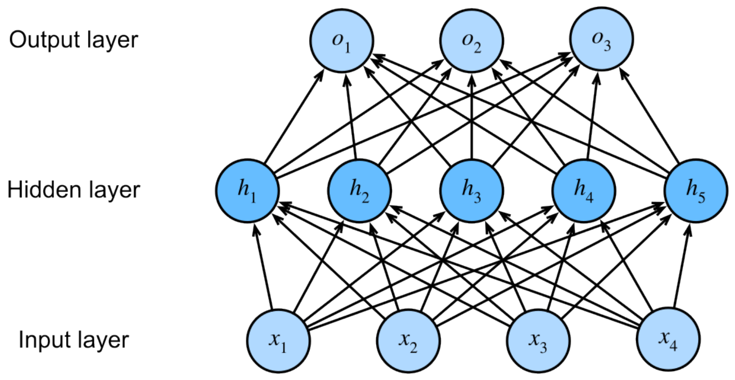


Figure 2.1: FFN with 5 hidden neurons. Figure is taken from `https://d2l.ai/chapter_multilayer-perceptrons`.

A layer with n inputs and k outputs is a stack of artificial neurons with weight matrix $W \in \mathbb{R}^{k\times n}$, bias vector $b \in \mathbb{R}^k$ and the same activation function. A layer's output can be written in a more compact matrix form:

$$g\left(Wx + b\right)$$

The FFN can be defined as a function $f = f_1(f_2(...(f_n(x))))$, where each f_i is a separate layer with its own weight matrix and bias vector with potentially different dimensions. The parameter vector θ is the collection of all values from the weight matrices and bias vectors.

2.3 Convolutional Neural Networks

Convolutional Neural Networks (*CNNs*) are a type of feedforward neural network architecture that have gained significant popularity in various fields, particularly in computer vision and signal processing. In CNNs, there are convolutional and pooling layers preceding the fully-connected layers. Mathematically, 1D convolution is a linear operation, defined as the element-wise multiplication of a filter/kernel with the input signal, followed by the summation of the multiplied values, for all values of the input signal (*zero-padded*):

$$y[n] = \sum_{k=0}^{K-1} x[n-k] \cdot h[k], \quad (n = 1, 2, ..., N),$$

where $y[n]$ is the output at index n, $x[n]$ is the input signal, $h[k]$ is the filter, K is the filter length, and N is the input signal length. This operation allows the network to capture local patterns and dependencies within the signal.

The pooling layer performs a nonlinear downsampling operation, reducing the dimensionality of the signal. Popular pooling techniques in CNNs include max pooling, where the maximum value within a region is retained, and average pooling, where the average value is computed. This combination of convolution and pooling enables CNNs to effectively capture and represent complex, multidimensional features in the input data. CNNs are commonly referred to as feature extractors for this reason.

2.4 Training procedure

As mentioned earlier, during training, the objective is to find the optimal parameter value that minimizes the loss function on the data. In regression tasks, the mean squared error is commonly used as the loss function, which is defined as:

$$\text{MeanSquaredError} = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

where y_i represents the true value and $\hat{y}_i$ represents the predicted value for the i-th data point. In classification tasks, the cross-entropy loss is often employed, which is defined as:

$$\text{CrossEntropy} = -\frac{1}{m}\sum_{i=1}^{m}\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c})$$

where $y_{i,c}$ is the true probability of the i-th data point belonging to class c, $\hat{y}_{i,c}$ is the predicted probability of the i-th data point belonging to class c, and C is the total number of classes.

Due to the intricate structure of neural networks, closed-form solutions for the minimizer are generally not attainable [3]. Instead, iterative optimization algorithms are used in general. In this regard training a neural network is not much different from any other numerical optimization problem. The main difference is that the presence of nonlinear activation functions makes the loss functions non-convex. As a result, optimizer algorithms will tend to converge to a locally low value, but finding the global minimizer becomes highly unlikely. Another consequence is that training is sensitive to initialization, and there are no guarantees of convergence [3].

A popular optimization algorithm that forms the basis of many other modern algorithms is *gradient descent*. The idea is simple:

1. Begin with an initial guess for the parameters.
2. Calculate the gradient of the loss function with respect to the parameters.
3. Update the parameters in a manner that reduces the value of the loss function.
4. Repeat steps 2 and 3 until convergence, or until step limit reached.

The algorithm's update formula can be expressed in a short equation.

$$\theta_{i+1} = \theta_i - \alpha \frac{\partial L}{\partial \theta}$$

The α hyperparameter is called the learning rate, and it controls the speed of the training process.

Manual computation of gradients becomes increasingly complicated as the neural network architecture grows in size and complexity. Numerically approximating the gradient with algorithms like finite differences [4] is also a highly inefficient solution, because it requires several function evaluations. *Automatic differentiation* addresses these challenges by providing a systematic and efficient way to compute gradients. It breaks down complex functions into elementary operations and applies the chain rule to compute derivatives efficiently.

The *backpropagation algorithm* [5] combines automatic differentiation and stochastic gradient descent to optimize neural networks efficiently. It computes the gradients of the cost function with respect to each parameter using the chain rule and updates the parameters accordingly. The algorithm consists of two phases: forward propagation and backward propagation. In forward propagation, the input data propagates through the layers of the neural network, and activations are computed. These activations are then used to compute the output of the network and the corresponding loss. During backward propagation, the algorithm starts from the output layer and works backward to compute the gradients of the parameters. It calculates the gradient of the loss with respect to each parameter by applying the chain rule and propagating the gradients backward through the layers.

Chapter 3

Model based neural networks

Fields such as signal processing, communications, and control have traditionally relied on classical statistical modeling techniques. These methods utilize mathematical formulations that capture the underlying physics, prior information, and domain-specific knowledge. While simple classical models are useful, they are sensitive to inaccuracies and may underperform when real systems exhibit complex or dynamic behavior [6]. Alternatively, data-driven approaches like deep neural networks with generic architectures have shown exceptional performance, particularly in supervised problems, as they learn to operate directly from the data. However, neural networks typically require large amounts of data and substantial computational resources, limiting their applicability in certain scenarios. Another issue is explainability. Even if a large enough training data is available, many fields are not willing to adopt the usage of models where the reasons behind the predictions are not easily understandable.

There are multiple motivating factors for combining the two approaches. The goal is to produce models that leverage the high performance of deep learning solutions while maintaining interpretability. These models should make it easy to incorporate domain knowledge and be trainable even with smaller-scale datasets.

3.1 Inference systems

In order to highlight the conceptual differences between model-based and data-driven solutions, a mathematical formulation of a generic inference problem is required. The term *inference* encompasses the capability to reach conclusions by considering evidence and employing reasoning, specifically it means the ability to make predictions based on a set of observed variables [6]. The system is required to map input variables $x \in \mathcal{X}$ to label variables $s \in \mathcal{S}$, where $\mathcal{X}$ is the input set and $\mathcal{S}$ is the label set. The output of the mapping is called a *prediction* and is denoted by $\hat{s}$. Thus, an inference rule is a function $f : \mathcal{X} \to \mathcal{S}$. The space of all possible inference mappings is $\mathcal{F}$. The *cost measure* is denoted by $\ell(\cdot)$ and is defined over the set $\mathcal{F} \times \mathcal{X} \times \mathcal{S}$. The fidelity of the model is measured by the *risk function* given by $\mathbb{E}_{x,s\sim p_{x,s}}\{\ell(f, x, s)\}$, where $p_{x,s}$ is the underlying statistical model which relates the input and the label. The goal of both approaches is to define an inference rule f to minimize the risk.

Model-based algorithms set the inference rule based on domain knowledge [6]. Domain knowledge in this context refers to the underlying knowledge of the statistics $p_{x,s}$. More specifically, an analytical expression of $p_{x,s}$ is required, in order to implement a provably correct risk minimization algorithm (e.g. based on maximum likelihood estimation). While computing the minimization rule is often computationally intractable, various model-based algorithms can approximate it sufficiently. The quality of a model-based solution is largely contingent upon precise knowledge of the statistical model, which is often unavailable. Consequently, certain assumptions are commonly made to compensate for this limitation.

A data-driven system on the other hand relies solely on observed data in the form of (x, y) pairs: $\{(x_t, y_t)\}_{t=1}^{m}\, m \in \mathbb{N}$. Since data-driven systems do not have access to the underlying distribution they are not able to compute the risk function, only the *empirical* risk function which is commonly referred to as the *loss function*: $L(f) = \frac{1}{m}\sum_{t=1}^{m} \ell(f, x_t, s_t)$. A leading strategy is to set up a highly expressive parameterized function f_θ (e.g. deep neural network) and train it on the dataset using numerical optimization algorithms. The universal approximation theorem implies that sufficiently large networks have the capability to effectively approximate any continuous mapping [3].

Model-based deep learning schemes, as a third approach, establish their f based on a combination of domain knowledge and observed data. Existing techniques in this area can be broadly categorized into two groups. The first category consists of *model-aided networks*, which employ neural networks for inference but adopt problem specific model-based architectures instead of conventional ones. The second category encompasses *neural network-aided inference systems*, where model-based approaches are used for inference, but neural networks are incorporated to enhance specific components of the algorithm. These approaches represent different ways of leveraging both model-based and data-driven techniques to address the task at hand [6].

3.2 Deep unfolding

Deep unfolding is a technique which converts an iterative algorithm into a neural network architecture by designing each layer to resemble a single iteration [7]. The method was originally proposed by Gregor and LeCun in [8], where they used it to unfold the iterative shrinkage and thresholding algorithm (ISTA) to solve the *sparse coding* problem.

The problem of sparse coding is the following. Given an input vector $y \in \mathbb{R}^n$ and an over-complete set of basis vectors $W \subset \mathbb{R}^{n \times m}$ where $m > n$, sparse coding refers to the task of finding a sparse representation x of y using W such that $y \approx Wx$, while forcing as many coefficients as possible in x to be zero [9]. A common approach to finding x is to solve the following convex optimization problem.

$$\min_{x \in \mathbb{R}} \frac{1}{2} \|y - Wx\|_2^2 + \lambda \|x\|_1$$

The λ parameter controls the sparsity of the solution. The iterations of the ISTA algorithm are roughly equivalent to solving the above problem with a step of gradient descent and a projection to the ℓ_1 ball [9]. The ISTA update formula is the following equation:

$$x_{i+1} = \mathcal{S}_\lambda \left(\left(I - \frac{1}{\mu} W^\top W \right) x_i + \frac{1}{\mu} W^\top y \right),$$

where I is the identity matrix, μ is the step size parameter, and $\mathcal{S}_\lambda(x) = sign(x) \cdot max\{|x| - \lambda, 0\}$. The update formula can be recast into a network layer by applying two generalizing substitutions: $W_t = I - \frac{1}{\mu} W^\top W$ and $W_e = \frac{1}{\mu} W^\top$. This yields the following equation for a single layer.

$$x_{i+1} = \mathcal{S}_\lambda \left(W_t x_i + W_e y \right),$$

where λ and the matrices W_t and W_e are learnable parameters. Then the unrolled network can be trained on real data using the mean squared error as a loss function. Experiments show that the unrolled network achieves convergence with substantially fewer iterations [8].

The unrolling procedure can be summarized in the general case with few steps.

1. Select an iterative optimization algorithm that is suitable for the specific problem.
2. Convert the update formula h into a parameterized version h_θ while preserving the function's structure.
3. Fix the number of iterations l, and create an l-layer deep network.
4. Train the network end-to-end using observed data.

Figure 3.1 illustrates the concept. It is worth noting, that *untying* the weights of the layers allows for a greater level of expressivity [7].

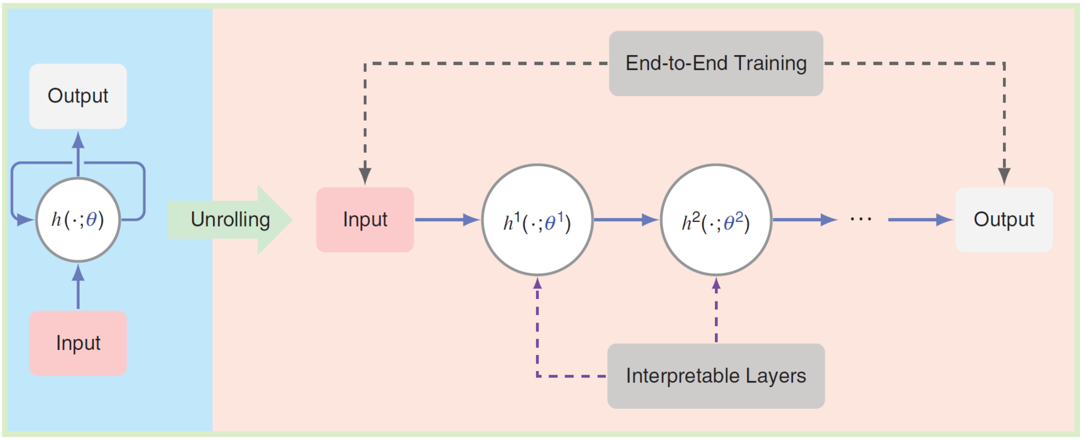


Figure 3.1: High-level overview of algorithm unrolling. Figure is taken from [9].

One of the best performing examples of unfolding, and the main inspiration for the current work, is the *DetNet* architecture. [10]. The DetNet architecture unfolds the projected gradient descent algorithm in order to solve the *multiple-input multiple-output* (MIMO) symbol detection problem. The task is to recover the unknown vector $x \in \mathcal{S}^k$ from the received vector $y \in \mathbb{R}^n$ which was sent on a noisy channel with known channel matrix $H \in \mathbb{R}^{n \times k}$ and unknown Gaussian noise $w \in \mathbb{R}^n$. The relationship between the sent and received signal can be described by the following model:

$$y = Hx + w$$

The set $\mathcal{S} \subset \mathbb{R}$ is the finite set of symbols. The optimal symbol detector is based on maximum likelihood estimation, but has an exponential time complexity which makes it impractical in most situations [10]. However, using the iterative projected gradient descent algorithm it is solvable with affordable computational complexity. The update formula for the algorithm is as follows.

$$\begin{aligned} x_{i+1} &= \Pi \left(x_i - \delta \frac{\partial \|y - Hx_i\|_2^2}{\partial x_i} \right) \\ &= \Pi \left(x_i + \delta H^\top y - \delta H^\top H x_i \right) \end{aligned}$$

where Π is a nonlinear projection operator, and δ is the step size parameter. Each iteration of the algorithm computes a linear combination of x_i, $H^\top y$, and $H^\top H x_i$, and then applies a nonlinear transformation. This is already very similar to the operations of FFNs, enriching these steps with learnable weight matrices results in the basic DetNet architecture:

$$x_{i+1} = W_2\, g(W_1(x_i - \delta_1 H^\top y + \delta_2 H^\top H x_i) + b_1) + b_2$$

where W_1, W_2, b_1, b_2, δ_1, and δ_2 are learnable parameters, and g is nonlinear activation function. To enhance clarity and better highlight the concept, this representation of DetNet is simplified. The original DetNet architecture contains a residual connection between layers, but for the sake of readability, this connection is omitted in the above formula. Each layer has its own weight matrices and bias vectors to enhance the expressive power of the network. The unrolled network is trained on data using the mean squared error with additional auxiliary losses on each hidden layer, similar to [11].

Chapter 4

Variable Projection method

This chapter provides an overview of the Variable Projection method, along with some necessary linear algebra concepts, drawing from [12]. For an extended introduction see Appendix A.

4.1 Pseudoinverse

A *pseudoinverse*, or a *Moore-Penrose inverse* is generalized matrix with additional conditions. A matrix $A \in \mathbb{R}^{m \times n}$ has a pseudoinverse matrix $B \in \mathbb{R}^{n \times m}$ if

1. $ABA = A$ (B is generalized inverse of A)
2. $BAB = B$ (A is generalized inverse of B)
3. $(AB)^\top = (AB)$ (AB is symmetric)
4. $(BA)^\top = (BA)$ (BA is symmetric)

The pseudoinverse of A is denoted by A^+.

The pseudoinverse can be computed for any *tall* matrix $A \in \mathbb{R}^{m \times n}$, where $m > n$, with full column rank by the following formula:

$$A^+ = (A^\top A)^{-1} A^\top.$$

However, directly computing the formula involves inverting the matrix $A^\top A$, which can be computationally expensive and numerically unstable. Instead, in practice usually the *Singular Value Decomposition* (*SVD*) is used.

The SVD of a matrix A is given by:

$$A = U\Sigma V^\top,$$

where $U \in \mathbb{R}^{m\times m}$ is an orthogonal matrix ($UU^\top = I$) representing the left singular vectors of A, $\Sigma \in \mathbb{R}^{m\times n}$ is a rectangular diagonal matrix containing the singular values of A, and $V \in \mathbb{R}^{n\times n}$ is an orthogonal matrix representing the right singular vectors of A. Using the SVD the computation of the pseudoinverse is given by:

$$A^+ = V\Sigma^+U^\top = V\Sigma^{-1}U^\top.$$

See Appendix A for a short proof.

4.2 Nonlinear least squares fitting with Variable Projection

The *Variable Projection* (VP) method was introduced by Golub and Pereyra in [1] in order to solve a specific class of nonlinear least squares problems: *separable nonlinear least squares* problems. A separable nonlinear least squares problem is one for which the model is a linear combination of nonlinear functions that can depend on multiple parameters [13].

Given a set of data points $\{(t_i, y_i)\}$ $t_i, y_i \in \mathbb{R}$ $i = 1, 2, ..., m;$ $m \in \mathbb{N}$, a nonlinear model is of the form

$$\eta(\alpha, c, t) = \sum_{j=1}^{n} c_j \phi_j(\alpha, t)$$

where ϕ_j are called the *basis functions*. The task is to find the optimal linear parameters $c = (c_1, c_2, ..., c_n) \in \mathbb{R}^n$ $n \in \mathbb{N}$ and nonlinear parameters $\alpha = (\alpha_1, \alpha_2, ..., \alpha_q) \in \mathcal{S}_\alpha \subseteq \mathbb{R}^q$ $q \in \mathbb{N}$ that minimize the nonlinear functional

$$r_1(\alpha, c, t) = \|y - \eta(\alpha, c)\|_2^2,$$

where $\eta(\alpha, c) = (\eta(\alpha, c, t_1), \eta(\alpha, c, t_2), ..., \eta(\alpha, c, t_m)) \in \mathbb{R}^m$. Note that the nonlinear parameters can be constrained ($\mathcal{S}_\alpha$), but the linear parameters must be unconstrained.

The fundamental idea of VP is to separate the linear and nonlinear parameters, and reformulate the objective in terms of only the nonlinear parameters, greatly reducing the problem size. The linear problem can be solved efficiently with direct numerical methods (e.g. with *SVD*), while the nonlinear problem can be solved with any general minimization algorithm (e.g. *Broyden–Fletcher–Goldfarb–Shanno algorithm*), or any Gauss-Newton type algorithm which takes advantage of the fact that the functional to be minimized is a sum of squares of functions (e.g. *Levenberg–Marquardt algorithm*).

Let $c(\alpha)$ denote the linear coefficients for any fixed α. Minimizing the r_1 functional is equivalent to minimizing the modified functional

$$r_2(\alpha, t) = \|y - \eta(\alpha, c(\alpha))\|_2^2,$$

which only depends on the nonlinear parameters [1]. This functional is known as the *variable projection functional*. The advantage of using VP is that derivatives can be given directly for r_2 based on r_1, simplifying the optimization process.

Now consider the discrete case. Let $\Phi(\alpha)$ denote the matrix where each column corresponds to the nonlinear functions $\phi_j(\alpha, t_i)$ evaluated at all t_i values, thus $\eta(\alpha, c) = \Phi(\alpha)c$. With this notation r_1 can be rewritten as:

$$r_1(\alpha, c) = \|y - \Phi(\alpha)c\|_2^2.$$

If the nonlinear parameters α are known, then the linear parameters c can be obtained by $c = \Phi(\alpha)^+ y$. By replacing c in r_1, the variable projection functional is obtained:

$$r_2(\alpha) = \|y - \Phi(\alpha)\Phi(\alpha)^+ y\|_2^2 = \|P(\alpha)y\|_2^2,$$

where $P(\alpha) = I - \Phi(\alpha)\Phi(\alpha)^+$ which is a projection onto the orthogonal complement of the range of $\Phi(\alpha)$ (hence the name of the method).

In order to utilize Gauss-Newton type optimization algorithms that rely on derivative information, the Jacobian matrix $J(\alpha)$ of $P(\alpha)y$ is required. To express the Jacobian matrix, a numerically efficient decomposition of $\Phi(\alpha)$ is needed to compute the pseudoinverse. This was done by *QR decomposition* in the original paper [1], but more recent formulations of the method use the *SVD decomposition* instead [14] to cover rank deficient cases as well. The Jacobian will be given in two parts: $J = -(A + B)$, where the definition for both A and B will be formulated by their columns (a_k and b_k respectively) to avoid dealing with 3-tensors.

Suppose that the *reduced* singular value decomposition of the *tall* matrix Φ is $U\Sigma V^\top$, where $U \in \mathbb{R}^{m\times rank(\Phi)}$, $V \in \mathbb{R}^{rank(\Phi)\times rank(\Phi)}$, and $\Sigma \in \mathbb{R}^{rank(\Phi)\times rank(\Phi)}$. The matrix Σ is a diagonal matrix of the nonzero singular values. The matrix U is not orthogonal in the reduced case thus $UU^\top$ is not the identity matrix, however V is orthogonal and $V^\top V = I$. As stated in the previous section the pseudoinverse of Φ can be computed with $\Phi^+ = V\Sigma^+U^\top = V\Sigma^{-1}U^\top$ [14]. Given these matrices, A and B can be derived using the product rule for differentiation.

$$\begin{aligned} a_k &= PD_k\Phi^+y \\ &= PD_kc \\ &= D_kc - U(U^\top(D_kc)) \quad k = 1,2,...,q \end{aligned}$$

$$\begin{aligned} b_k &= (PD_k\Phi^+)^\top y \\ &= (\Phi^+)^\top D_k^\top P^\top y \\ &= (\Phi^+)^\top D_k^\top r \\ &= U(\Sigma^{-1}(V^\top(D_k^\top r))) \quad k = 1,2,...,q \end{aligned}$$

where r contains the residuals $r = y - \Phi c = Py$, the matrix $P = I - \Phi\Phi^+ = I - (U\Sigma V^\top)(V\Sigma^{-1}U^\top) = I - UU^\top$, and the (i,j) element D_k contains the partial derivatives of Φ: $\frac{\partial\Phi_j}{\partial\alpha_k}$ evaluated at t_i. This formulation is efficient to compute, because the products only involve matrix-vector products, and the only inverse involves a diagonal matrix [14].

However, the optimization problem can also be solved with general purpose optimization algorithms as well. In this case the derivative of r_2 is necessary. This can be computed column-by-column using the product rule for the dot product:

$$\begin{aligned} \frac{\partial\|P(\alpha)y\|_2^2}{\partial\alpha_k} &= 2(P(\alpha)y)\frac{\partial(P(\alpha)y)}{\partial\alpha_k} \\ &= -2r^\top j_k = 2r^\top(a_k + b_k) \end{aligned}$$

where j_k is k-th column of the Jacobian, and r is the residual.

Chapter 5

Unfolded Variable Projection network

5.1 Unfolding the Variable Projection method

The unfolding of the Variable Projection method can be achieved similarly to the DetNet architecture. Since the VP method itself is not an optimization algorithm, it is necessary to select one for the unfolding process. Like with the DetNet architecture, the projected gradient descent algorithm is suitable for this task due to its similarity to neural network operations. However, there are a couple of important differences between the MIMO detection problem and the nonlinear least squares problem, which complicate matters.

Recall that solving the separable nonlinear least squares problem with Variable Projection means minimizing the functional

$$r_2(\alpha) = \|y - \Phi(\alpha)\Phi(\alpha)^+ y\|_2^2 = \|P(\alpha)y\|_2^2.$$

To apply the projected gradient descent optimization algorithm, it is necessary to calculate the derivative of r_2 with respect to the α parameters. This derivation was provided in the previous section:

$$\frac{dr_2(\alpha)}{d\alpha} = -2J^\top r.$$

Using the derivative, the update formula for the projected gradient descent algorithm is as follows:

$$\begin{aligned} x_{i+1} &= \Pi\left(x_i - \delta\frac{dr_2(\alpha)}{d\alpha}\right) \\ &= \Pi\left(x_i + \delta J^\top r\right), \end{aligned}$$

where Π is a nonlinear projection operator, and δ is the step size parameter. Replacing the Π operator with a single layer of a feedforward neural network results in the basic unfolded VP architecture:

$$ReLU(W(x_i + \delta J^\top r) + b),$$

where W, b and δ are learnable parameters for each layer separately.

One difference between the training processes of DetNet and the unrolled VP is that the DetNet model is designed to be trained with the actual nonlinear parameters. The model outputs the estimated nonlinear parameters, and the mean squared error measures its proximity to the true nonlinear parameters. Given the unavailability of these parameters in real world tasks, my objective was to train the unrolled VP network as a component of a larger classifier or regressor network for a signal processing task. In this way, the model can output the most appropriate nonlinear parameters for the specific task and dataset, without relying on knowledge of the true parameters. This greatly enhances the applicability of the network.

5.2 Adaptive Hermite system

The selection of basis functions is a crucial aspect in solving a specific problem with VP. Following prior work on ECG heart rate signal classification with VP [15], I also opted for the *adaptive Hermite system* as my choice of function system. The Hermite polynomials can be defined by a recurrence relation:

$$H_{k+1}(t) = 2tH_k(t) - 2kH_{k-1}(t) \quad (k \in \mathbb{N}^+, t \in \mathbb{R}),$$

$$H_0(t) = 1 \qquad H_1(t) = 2t.$$

These orthogonal polynomials can be parameterized by two parameters: *translation* ($\tau \in \mathbb{R}$) and *dilation* ($\lambda \in \mathbb{R}$):

$$\Phi_k(t, \tau, \lambda) = \sqrt{\lambda}\Phi_k(\lambda(t - \tau)),$$

where

$$\Phi_k(t) = \frac{H_k(t)e^{-\frac{t^2}{2}}}{\sqrt{\sqrt{\pi}2^k k!}} \quad (k \in \mathbb{N}^+).$$

The functions $\Phi_k(t, \tau, \lambda)$ are the *adaptive Hermite functions* (Figure 5.1).

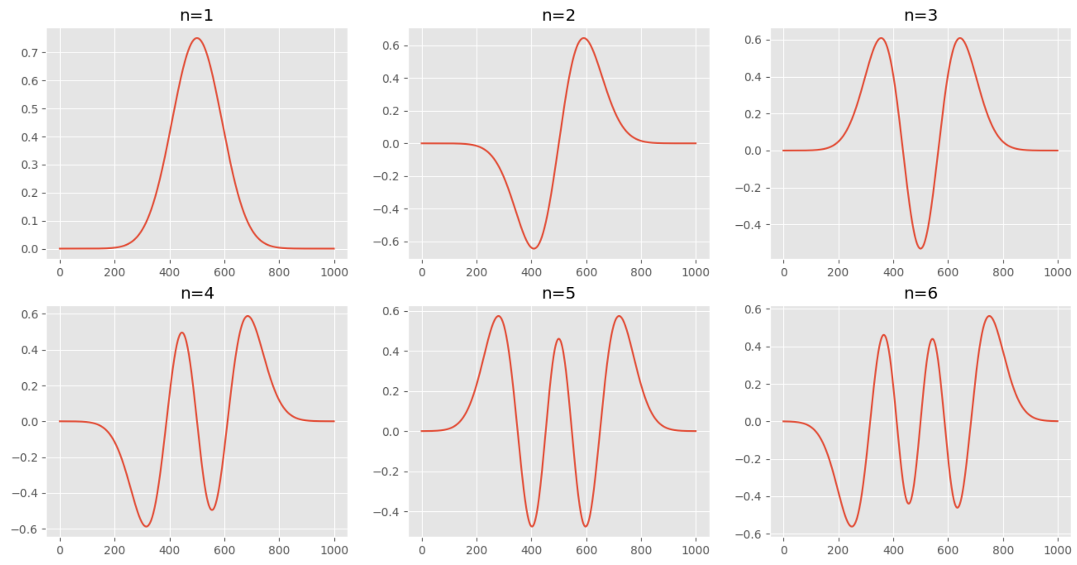


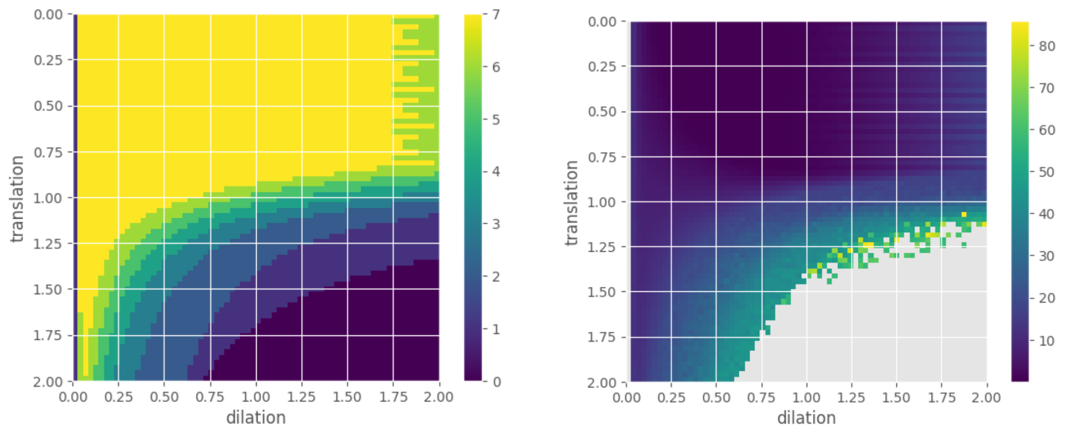
Figure 5.1: The first 6 adaptive Hermite functions.

5.3 Numerical challenges

Unfortunately, training the basic unrolled VP architecture proved to be challenging due to several numerical difficulties. These shortcomings and their solutions will be addressed in this section, in order of importance.

5.3.1 Consequences of using the Hermite system

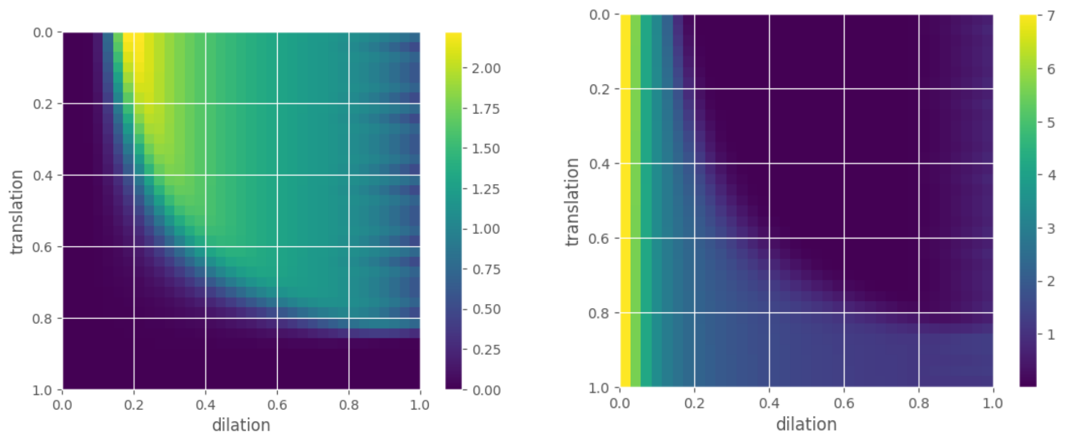
The Hermite system has various desirable properties. It is an orthogonal system, and the functions are either odd or even. When sampled uniformly however, the resulting matrix is not always orthogonal. Let the matrix $\Phi(\tau, \lambda) \in \mathbb{R}^{m \times k}$ denote the matrix $\Phi(\tau, \lambda) = \Phi_k(t_i, \tau, \lambda)$ which is computed by uniformly sampling the first k Hermite functions at the t_i data points. The Hermite functions are zero almost everywhere, except around 0. This means that certain parameter values *"push off"* the function values, resulting in the zero matrix.



(a) Rank (b) Log of condition number

Figure 5.2: Rank and condition number of the matrix $\Phi(\tau, \lambda)$ as a function of the translation and dilation parameters.

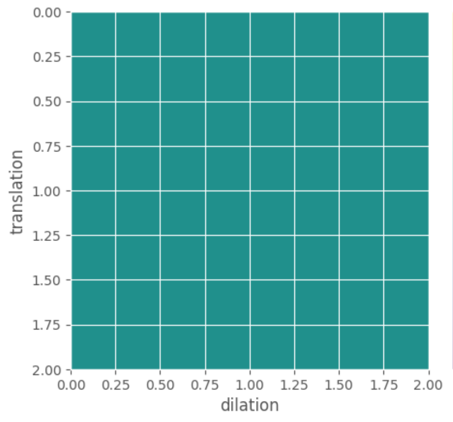
By examining the rank and condition number of the $\Phi(\tau, \lambda)$ matrix for fixed k and m values but across different translation and dilation parameters, the range of values where its usage is reasonable can be determined (Figure 5.2). The k and m values were chosen based on the ECG signal classification task.



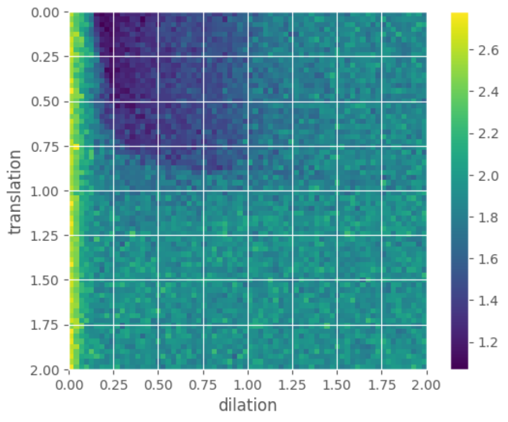
(a) Minimum singular value (b) Maximum distance between the singular values

Figure 5.3: Singular values of the matrix $\Phi(\tau, \lambda)$ as a function of the translation and dilation parameters.

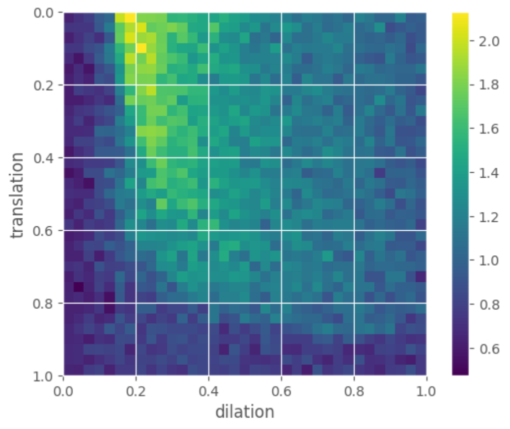
A crucial component of the unrolled VP is the computation of the singular value decomposition of the basis function matrix. However, within the range of the reasonable translation and dilation parameters the differentiation of the decomposition is numerically rather challenging. The derivative of SVD contains the term $\frac{1}{s_i^2-s_j^2}$, where s contains the singular values [16]. This means that the derivative will be numerically unstable if the singular values are too close to each other, or if the singular values are too close to zero. By examining the magnitudes and relative distances of the singular values within the reasonable range, it becomes apparent that they collectively cover nearly the entire range (Figure 5.3).



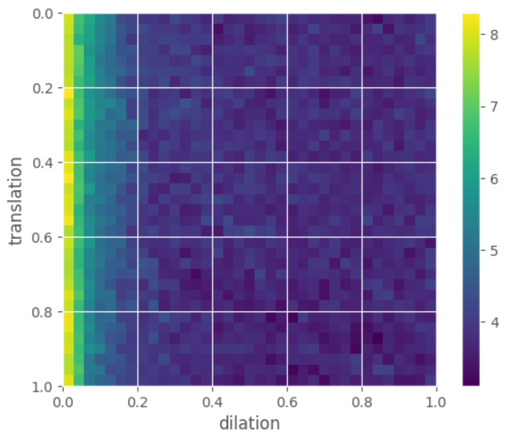
(a) Rank



(b) Log of condition number



(c) Minimum singular value



(d) Maximum distance between the singular values

Figure 5.4: Rank, condition number and singular values of the perturbed matrix $\Phi(\tau, \lambda) + R_{uniform}$ as a function of the translation and dilation parameters.

In order to solve this problem, it is customary to perturb the matrix with a small random noise as a form of regularization. This ensures that the singular values are neither close to zero nor to each other. Since random matrices are likely to have full rank this perturbation will likely improve the rank and the condition number as well when possible [17]. Figure 5.4 shows the improvements made by adding a small random matrix $R_{uniform} \in [0, 0.5]^{m \times k}$ to Φ.

5.3.2 Gradient norm clipping

Since optimization is not done against the actual nonlinear parameters but on a real signal processing task, it is imperative to ensure that the nonlinear parameters are always meaningful. To make convergence likely on a correct set of parameters, it is essential to avoid large jumps in the parameter space. Experiments show that the fastest convergence can be achieved if each layer is only slightly able to modify the nonlinear parameters. To achieve this my implementation uses explicit gradient clipping on the expression $J^\top r$ inside the VP layers.

Gradient clipping is a technique used widely in deep learning to mitigate the issue of exploding gradients during training. It involves setting a threshold value, and if the gradient exceeds this threshold, it is scaled down to prevent it from growing uncontrollably. By clipping the gradient, the technique helps stabilize the learning process and prevents gradients from becoming too large, which can hinder convergence or cause numerical instability. This allows the model to make more consistent updates to its parameters and facilitates smoother optimization.

To some extent, scaling the gradient is the task of the δ parameters in each layer. However, at the beginning of training, the norm of $J^\top r$ changes too rapidly for the δ parameters to accurately track.

5.3.3 Feedforward initialization

The standard way of initializing neural networks is by sampling from either uniform or standard normal distributions. One of the most popular initialization scheme is the *Xavier initialization*, also known as Glorot initialization [18]. It aims to address the challenge of finding appropriate initial weights that facilitate efficient learning and prevent issues such as vanishing or exploding gradients. Xavier initialization sets the initial weights of the network's neurons using a uniform distribution scaled by the square root of the input dimension. This scaling factor helps ensure that the activations and gradients during the forward and backward passes of the network remain within a desirable range.

Using standard methods however may not be optimal in the case of the unrolled VP network. A drawback of these initialization schemes is that, despite the relatively stable norm of the nonlinear parameters during the forward pass, they tend to produce jumps across the parameter space. My experiments show that initializing the layers between the VP layers with diagonal matrices which have their values sampled from a normal distribution $\mathcal{N}(1, s)$ where s is sufficiently small (e.g. 0.01) but keeping a modern initialization scheme for the rest of the network smooths out the optimization process and speeds up training. This initialization approach can be seen as initializing with the identity matrix and then enhancing it by incorporating small random perturbations.

5.3.4 VP loss

The unrolled VP network can be viewed as a feature extractor that captures and propagates essential characteristics of the input signal. By significantly reducing its dimensionality, the network efficiently extracts relevant information. Prior work on neural networks and the Variable Projection method introduced a secondary regularization loss term known as *percent root-mean-square difference (PRD)* [15], which is essentially the normed variable projection functional:

$$\frac{\|y - \Phi(\alpha)\Phi(\alpha)^{+}y\|_2^2}{\|y\|_2^2}.$$

The motivation behind this term arises from the objective of the VP layer, which is to extract relevant information from the input signal in a manner that facilitates accurate signal reconstruction. This term serves as a means to enforce high-quality signal approximations.

5.3.5 Constraints

Despite these efforts it is still possible that the value of the nonlinear parameters may venture into regions where the computation of the linear least squares, SVD, or the gradients of either becomes infeasible or unattainable. As a last-ditch effort my implementation includes bounds on the nonlinear parameters (similar to [19]), that are applied as maximum and minimum operations before every use of these parameters.

Chapter 6

Qualitative results: regression

In this section, I will briefly explore the Unrolled VP network in its filtering form as a *pure optimization technique* for fitting nonlinear functions. This analysis of the algorithm is primarily qualitative, aiming to compare it with other optimization techniques on synthetic data. The following three algorithms were used for comparison.

1. *LM*: Levenberg–Marquardt algorithm
2. *VP+LM*: Variable Projection with Levenberg–Marquardt algorithm
3. *VP+BFGS*: Variable Projection with Broyden–Fletcher–Goldfarb–Shanno algorithm

The difference between using the VP method with LM or BFGS lies in whether to rely on the fact that the objective is a sum of squares of functions. The derivative formulas for both cases were developed in Section 4.2.

The nonlinear parameters predicted by the Unrolled VP network can be used in two ways: *feature extraction* and *filtering*. Filtering is often more appropriate for regression problems, whereas feature extraction is well-suited for classification tasks.

Feature extraction refers to the process in which relevant information is extracted from raw data to create a compact representation that captures the essential characteristics of the data. It involves transforming the data into a set of features that are more informative and suitable for a specific task or analysis while reducing the dimensionality. In the case of the unrolled VP network, the features correspond to the coefficients of the basis functions. The VP feature extractor is the following function:

$$f(x) = \Phi(\alpha)^+ x = c, \quad x \in \mathbb{R}^m,$$

where α denotes the output of the unrolled network: the nonlinear parameters. The assumption behind using the coefficients (linear parameters) as features, is that in certain signal processing tasks it has been shown that the coefficients capture fine details, while the nonlinear parameters reflect the coarse changes in the signal morphologies [19].

Filtering on the other hand refers to the process of modifying a signal to enhance or remove unwanted features, aiming to improve signal quality and reduce noise. Since the VP method aims to approximate the input signal with a set of predefined parameterized functions, using the approximation in place of the input signal can be considered a form of filtering. The VP filter is the following function:

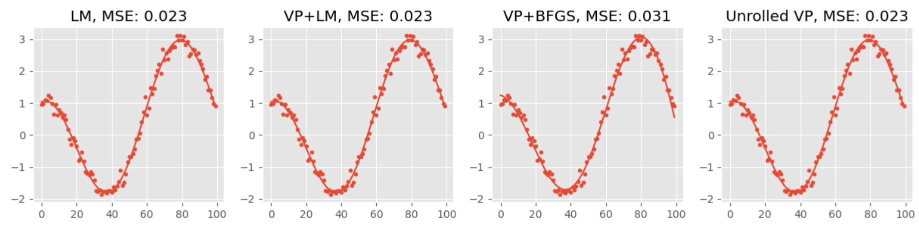
$$f(x) = \Phi(\alpha)\Phi(\alpha)^+ x, \quad x \in \mathbb{R}^m,$$

where again α denotes the nonlinear parameters.

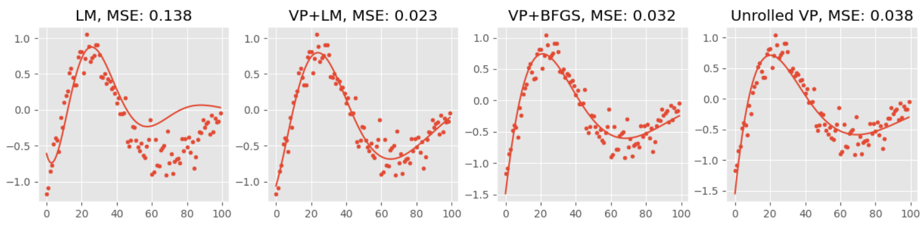
For the comparisons three different basis function systems were used.

1. *Hermite*: Adaptive Hermite functions, see Section 5.2 for the definitions.
2. *Exp*: Products of exponential and trigonometric functions: $e^{-\beta_j t} \cdot cos(\gamma_j t)$.
3. *Sin*: Parameterized *sin* functions: $sin(\beta_j + \gamma_j t)$.

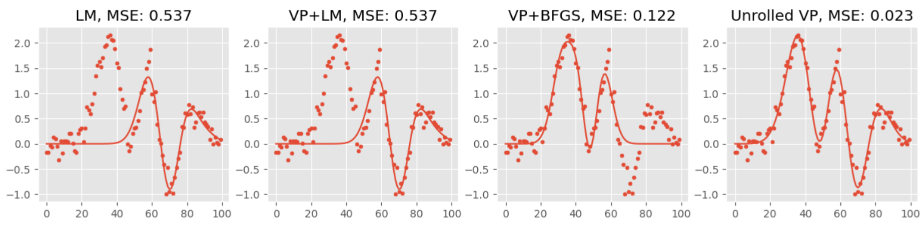
Figure 6.1 shows an example of each function system. The *Hermite* example highlights the advantage of unfolding as compared to the other two VP methods. Further quantitative investigation is necessary to precisely assess the capabilities of the unrolled VP network in regression tasks.



(a) example *Sin*



(b) example *Exp*



(c) example *Hermite*

Figure 6.1: Nonlinear function fitting examples.

Chapter 7

Quantitative results: classification

The performance of the unrolled VP network was measured on two classification datasets. The first dataset is a synthetic dataset which was made using linear combinations of adaptive Hermite functions [15]. The dataset consists of three classes each with $10,000$ samples. I used a 50-50% split for training and testing. Figure 7.1 shows some examples from the dataset.

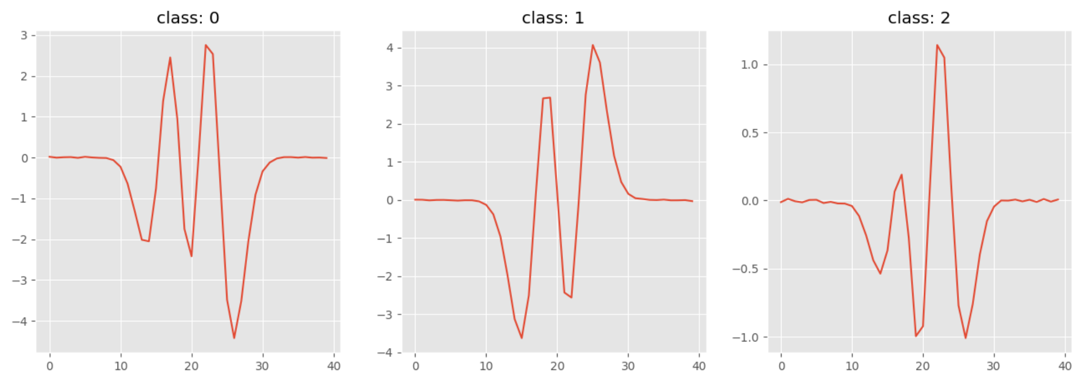


Figure 7.1: Examples from each class of the synthetic dataset.

The second dataset is a subset of the MIT-BIH Arrhythmia Database [20]. The task is to correctly classify the two largest arrhythmia classes. It contains $8,520$ samples for training and $6,440$ samples for testing with perfect class balance. Figure 7.2 shows two examples from the dataset.

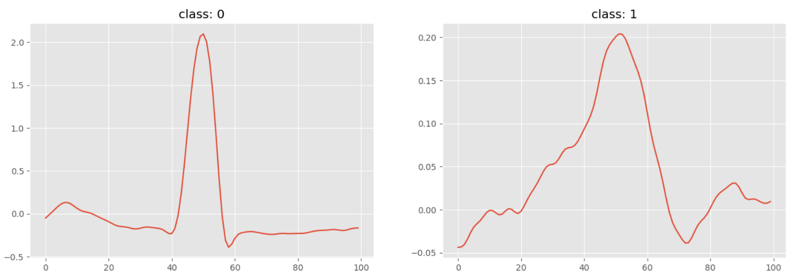


Figure 7.2: Examples from each class of the ECG dataset.

7.1 Performance of the Unrolled VP network

First, I will present my experimental findings on the performance of the Unrolled VP network along with the contributions made by the various improvements to the basic formula which were detailed in Section 5.3. Then I will compare the unrolled VP network to various other architectures.

Number of VP iterations

For the classification task, I found it beneficial for the Unrolled VP network to include a single hidden layer after the unfolded iterations and before the final output layer. Table 7.1 summarizes the results for varying numbers of unrolled iterations (depth) and different numbers of neurons in this hidden layer. From this point onward, the term *hidden neurons* will refer to the neurons in this particular hidden layer, unless stated otherwise. The results align with the intuition that increasing either the number of hidden neurons or the number of VP iterations increases performance.

No. hidden neurons	**depth = 1**	**depth = 3**	**depth = 5**	**depth = 7**
3	81.85%	82.00%	81.07%	82.03%
5	97.13%	97.78%	99.39%	99.83%
10	99.84%	99.48%	99.43%	99.82%

Table 7.1: Results on synthetic data: best test accuracies for different depth and no. hidden neurons.

Nonlinear parameter initialization

Since each unfolded iteration starts from an initial guess, it is crucial to initialize this hyperparameter reasonably. Nevertheless, verifying successful convergence from multiple initial guesses is a reliable approach for assessing the algorithm's robustness.

Initial dilation	Initial translation	Best accuracy
0.2	0.0	99.39%
0.3	0.0	97.78%
0.4	0.0	97.73%
0.2	0.1	81.29%
0.2	0.2	97.69%
0.2	0.3	80.31%

Table 7.2: Results on synthetic data: best test accuracies for 5 hidden neurons and different initial nonlinear parameter values.

Table 7.2 shows the results for a 5 iteration deep network with 5 hidden neurons. The initial parameters result in substantially different Hermite functions (Figure 7.3), and the model appears to struggle with some of them.

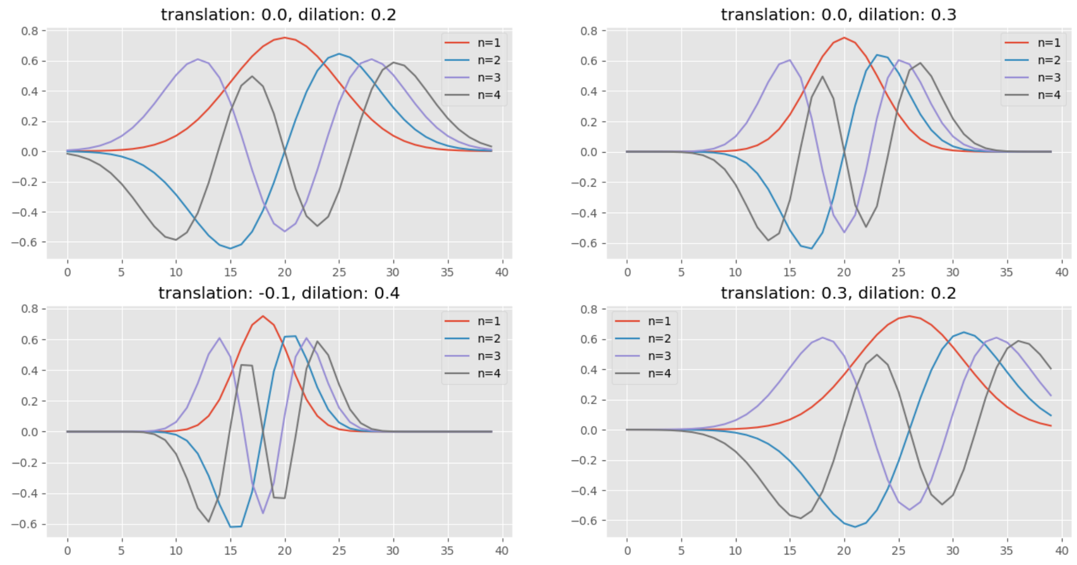


Figure 7.3: The first 4 adaptive Hermite functions with different translation and dilation values.

Table 7.3 shows that increasing the number of hidden neurons after the unrolled part can offset the more difficult initializations.

Initial dilation	Initial translation	Best accuracy
0.2	0.0	99.43%
0.3	0.0	99.68%
0.4	0.0	99.37%
0.2	0.1	99.35%
0.2	0.2	99.85%
0.2	0.3	99.92%

Table 7.3: Results on synthetic data: best test accuracies for 10 hidden neurons and different initial nonlinear parameter values.

Regularization

Regularizing the model function matrix with a small random matrix is essential for increasing the numerical stability of the algorithm. However, excessive regularization can cause the derivative approximations to become too imprecise, thereby hindering performance. Table 7.4 shows the results of a 5 iteration deep network with 5 hidden neurons. Surprisingly, the performance exhibits only a slight decrease even with relatively high levels of regularization. The impact of introducing noise to the Hermite functions is illustrated in Figure 7.4.

Regularization weight	Best accuracy
0.005	99.23%
0.01	99.39%
0.1	99.39%
0.3	97.12%
0.5	97.06%

Table 7.4: Results on synthetic data: best test accuracies for different levels of regularization.

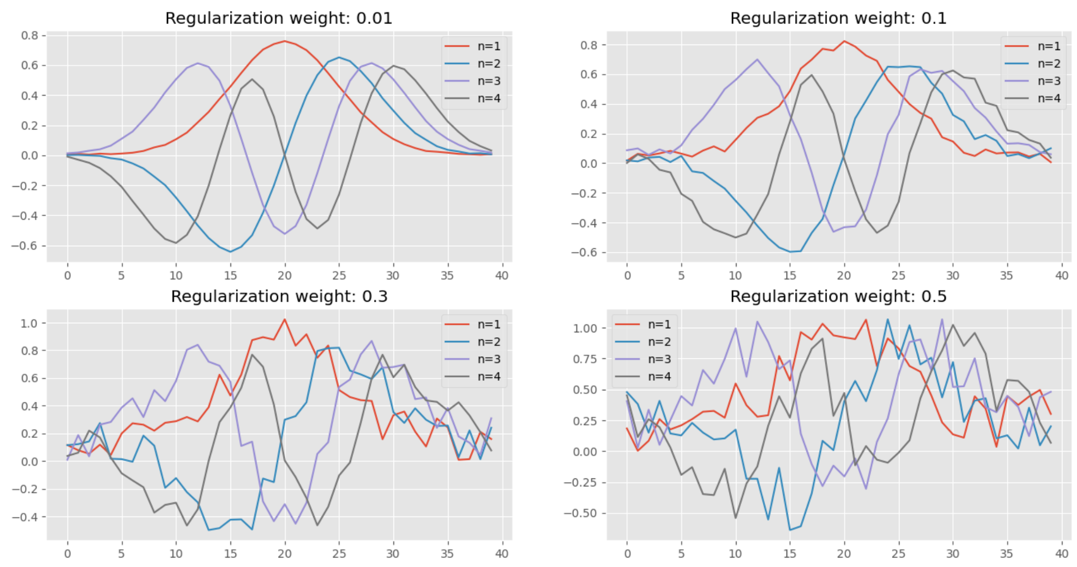


Figure 7.4: The first 4 adaptive Hermite functions with different levels of random noise.

Feedforward initialization

Adequately initializing the feedforward weight matrices and bias vectors in between the VP layers has a large impact on performance. Using standard initialization results in poor performance, as shown in Table 7.5. Using identity matrices and zero bias vectors is a much better solution, since it means that at the beginning of training only the VP layers will influence the nonlinear parameters. An even better solution is to introduce small random perturbations into the diagonal of the identity matrix. Using an identity matrix is similar to constant initialization, and as such it fails to break the symmetry between neurons. This symmetry can lead to symmetric gradients, causing neurons to update their weights in the same way during training. As a result, the neurons become highly correlated and lose the ability to learn independent features, negatively impacting the network's performance [18]. However, increasing the number of hidden neurons can alleviate the effects of suboptimal initialization.

Initialization type	No. hidden neurons	Best accuracy
Random uniform	5	49.84%
Random uniform	10	51.93%
Identity	5	97.84%
Identity	10	99.83%
Identity + noise	5	99.39%
Identity + noise	10	99.43%

Table 7.5: Results on synthetic data: best test accuracies for different feedforward initialization schemes.

VP loss

Recall that the VP loss is defined as $\frac{\|y-\Phi(\alpha)\Phi(\alpha)^{+}y\|_2^2}{\|y\|_2^2}$. Its usage is based on the assumption that precise VP approximations may lead to informative features and therefore to high classification accuracy [15]. Table 7.6 demonstrates that incorporating the VP loss alongside the cross-entropy loss results in improved performance across a wider range of nonlinear parameter initializations.

Initial dilation	Initial translation	Best accuracy with VP loss	Best accuracy without VP loss
0.2	0.0	99.43%	83.03%
0.3	0.0	99.37%	97.71%
0.4	0.0	99.54%	51.00%
0.2	0.1	99.25%	97.33%
0.2	0.2	99.88%	97.32%
0.2	0.3	99.88%	81.86%

Table 7.6: Results on synthetic data: best test accuracies for different losses and initial nonlinear parameter values.

Using the VP loss also speeds up the speed of convergence. Figure 7.5 shows the best results from the above table: 0.3 dilation and 0.0 translation.

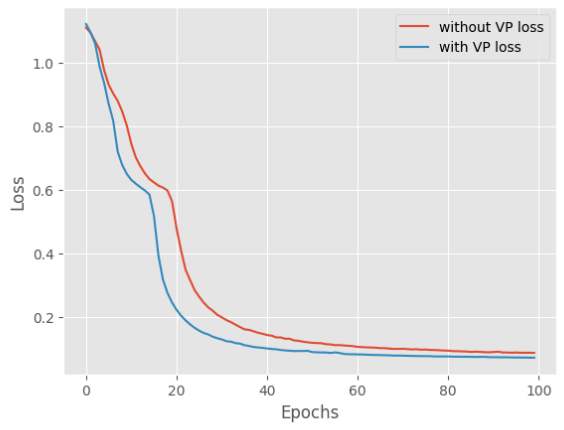


Figure 7.5: Best training curves with and without the VP loss.

Residual connections

The DetNet architecture inspired by ResNet [21] contained residual connections between each unrolled layer. Residual connections address the issue of vanishing gradients by adding the layer's output to the subsequent layer's input. This creates a shortcut connection that allows the network to retain information from previous layers, and it makes training much deeper networks possible [21]. As shown in Table 7.7, the Unrolled VP network did not seem to benefit from residual connections. Further investigation is needed to uncover why this is the case.

No. hidden neurons	**depth = 1**	**depth = 3**	**depth = 5**	**depth = 7**
3	82.50%	81.94%	81.64%	80.17%
5	81.86%	97.87%	97.53%	99.56%
10	99.90%	99.38%	99.76%	99.93%

Table 7.7: Results on synthetic data: best test accuracies for different depth and no. hidden neurons with residual connections.

7.2 Comparative analysis of performance

The following three networks were used for comparison.

1. *FFN*: Simple feedforward neural network.
2. *CNN*: A convolutional neural network.
3. *VPNet*: A model-based feedforward neural network that contains a Variable Projection based feature extraction layer.

The feedforward network consists of one or two layers and uses the ReLU activation function. It is meant to be a baseline model. The CNN contains a 1D convolutional and pooling layer followed by a simple feedforward network. The convolutional layer can be thought of as a feature extractor. The VPNet model is a custom implementation of the model first published by Kovács et al in [15]. The key idea of the VPNet model is to use the Variable Projection as a feature extractor, but with learnable nonlinear parameters. The VPNet model is more compact and parameter efficient compared to the unrolled VP network. However, after training, the VPNet model utilizes fixed global values for the nonlinear parameters. In contrast, the Unrolled VP network has the capability to predict different nonlinear parameters for each input signal through the unfolding process.

No. learnable parameters	**FFN**	**CNN**	**VPNet**	**Unrolled VP**
30 - 45			57.17%	81.85%
45 - 60			97.51%	82.00%
60 - 75			96.55%	97.13%
75 - 90		96.78%	99.61%	97.78%
90 - 105		98.37%	99.82%	99.39%
105 - 120		82.07%	99.65%	**99.84%**
120 - 135	34.11%	98.47%	**99.90%**	99.48%
135 - 150	83.26%		99.81%	99.43%
150 - 165		**99.84%**		99.82%
165 - 180	81.35%	99.42%	99.84%	
195 - 210	85.75%	97.54%		
210 - 225		97.81%	99.83%	
225 - 240	96.87%			
240 - 255	98.95%			
285 - 300	98.46%	99.39%		
300 - 315	99.63%	99.53%		
435 - 450	99.78%			
480 - 495	99.56%			
540 - 555	**99.81%**			

Table 7.8: Results on synthetic data: best test accuracies for the different models as a function of parameter count.

Table 7.8 summarizes the results for the synthetic dataset, and Table 7.9 for the ECG dataset, for the different models with different number of learnable parameters. The results achieved by the VPNet model are slightly lower than those reported in the original paper. This discrepancy is likely due to the fact that the original implementation precisely defines the gradients for the VP layer, whereas my implementation relies on PyTorch's *autograd* functionality.

No. learnable parameters	**FFN**	**CNN**	**VPNet**	**Unrolled VP**
30 - 45			**93.47%**	91.74%
45 - 60			92.37%	94.10%
60 - 75			93.07%	92.86%
75 - 90			93.30%	**94.74%**
90 - 105			92.49%	91.62%
105 - 120			91.56%	93.09%
120 - 135			92.49%	93.79%
135 - 150				94.27%
150 - 200		91.59%	92.73%	93.53%
200 - 250	92.70%		93.35%	
250 - 300		**93.74%**		
300 - 350	93.79%			
350 - 400		92.89%		
400 - 450	93.68%			
500 - 550	93.57%	93.40%		
550 - 600	93.50%			
700 - 750	93.83%	92.64%		
1000 - 1050	**94.04%**			
1050 - 1100	92.91%			
1100 - 1150	91.98%			

Table 7.9: Results on ECG data: best test accuracies for the different models as a function of parameter count.

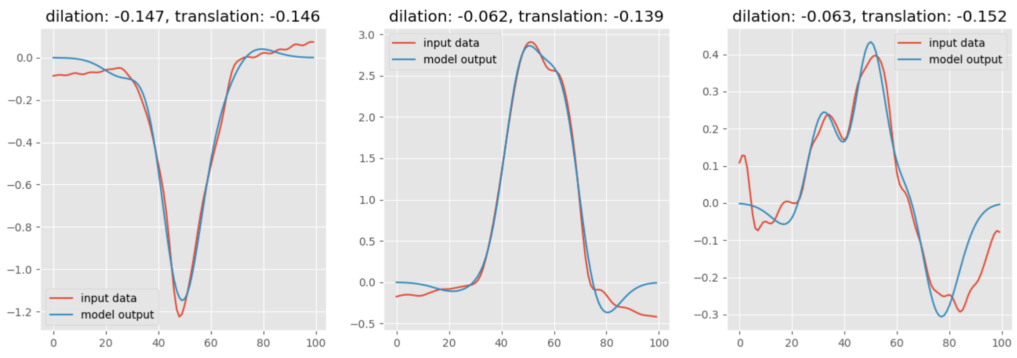
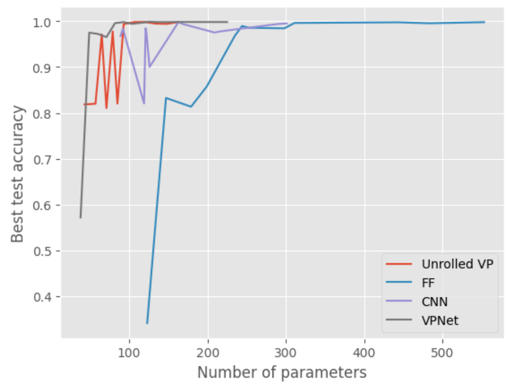


Figure 7.6: Examples from the ECG dataset and the output of the trained model.

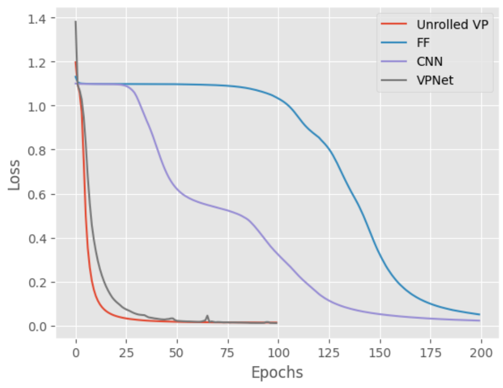
The results show that both VPNet and the Unrolled VP networks are more parameter efficient than the CNN and FFN models. Both of them are also highly explainable due to their architectures closely resembling the VP iterative algorithm.

Figure 7.6 shows some examples from the ECG dataset and the Hermite functions fitted by the Unrolled VP model.

Another advantage of using model-based networks is the accelerated speed of training. Both VP based models achieved convergence much faster than the traditional architectures (Figure 7.7).



(a) Best test accuracies as a function of parameter count



(b) Best training curves

Figure 7.7: Evaluation on the synthetic dataset.

One key distinction between the Unrolled VP model and the VPNet model is that the unrolled model has the ability to predict nonlinear parameters that are dependent on the input signal. Figure 7.8 shows the different translation and dilation values predicted by the Unrolled VP network on the synthetic dataset.

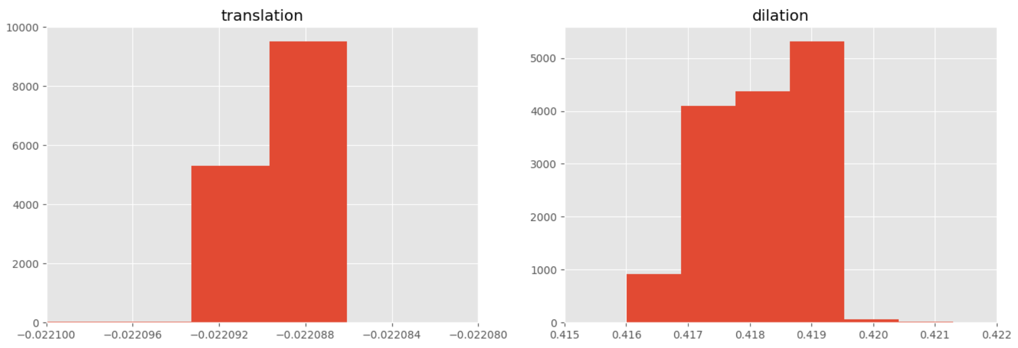


Figure 7.8: Histogram of the translation and dilation values from the trained model's output.

Chapter 8

Conclusion

In my thesis I explored the application of the deep unfolding technique to the Variable Projection method. By leveraging the properties of unfolding, I aimed to create an interpretable neural network architecture. Understanding how a neural network arrives at its decisions is of great importance, particularly in domains where transparency and explainability are vital.

I described the essential components required for successful unfolding and showcased its capabilities in both regression and classification tasks. Additionally, I examined the contributions of the different aspects of the model, including the network depth, different levels of regularization, and the effects of various initialization schemes. I also compared it to both conventional and model-based architectures. This comparative analysis allowed for the evaluation of the strengths and weaknesses of the unfolded model in relation to other existing approaches. In conclusion, the unrolled Variable Projection network not only demonstrated competitive performance but also showcased advantages such as requiring fewer parameters, shorter training time, and providing interpretability.

Appendix A

Generalized inverse and least squares

This section serves as an extended introduction to the Variable Projection method, discussed in Chapter 4.

Generalized inverse

Recall that a matrix $A \in \mathbb{R}^{n\times n}$ $(n \in \mathbb{N})$ is *invertible* if there exists a square matrix $B \in \mathbb{R}^{n\times n}$ such that $AB = I$ and $BA = I$, where I is the identity matrix. B is called the *inverse matrix* of A denoted by $B = A^{-1}$. Two equivalent ways of characterizing an invertible matrix A: A has full rank, A has nonzero determinant. Linear systems will play a significant role later on. It is important to recall that for any invertible matrix $A \in \mathbb{R}^{n\times n}$ and any vector $b \in \mathbb{R}^n$ the linear system $Ax = b$ has a unique solution $x^* = A^{-1}b$ [12].

For a general non-square matrix, the concept of an inverse matrix becomes slightly more intricate. Let $A \in \mathbb{R}^{m\times n}$ $m, n \in \mathbb{N}$ be any matrix, then $G \in \mathbb{R}^{n\times m}$ is called the *generalized inverse* of A if $AGA = A$. Note that if the matrix A is square and invertible it has only one generalized inverse which is A^{-1}. *Proof:* From $AGA = A$:

$$G = A^{-1}(AGA)A^{-1} = A^{-1}(A)A^{-1} = A^{-1}.$$

For a general matrix its generalized inverse always exists but might not always be unique [12].

The generalized inverse can be used to solve a consistent linear system (a system with at least one existing solution). In $Ax = b$ suppose that $b \in Col(A)$, where Col

denotes the column space of a matrix. Let G be the generalized inverse of A, then $x^* = Gb$ is a solution to the system. *Proof:* Multiply both sides by AG:

$$(AG)b = (AG)Ax = (AGA)x = Ax = b.$$

Projection matrix

A square matrix P is called a *projection matrix* if $P = P^2$, in other words if P is *idempotent.* A projection matrix *projects* any vector in $\mathbb{R}^n$ onto its column space (range). Due to its idempotency property, any vector already within its range will remain unchanged. Let $A \in \mathbb{R}^{m \times n}$ be a matrix with a generalized inverse $G \in \mathbb{R}^{n \times m}$, then $AG \in \mathbb{R}^{m \times m}$ is a projection matrix. *Proof:* From $AGA = A$:

$$(AG)(AG) = (AGA)G = AG.$$

Similarly, $GA \in \mathbb{R}^{n \times n}$ is also a projection matrix. It can be shows that AG and A have the same column space.

1. For any $y \in Col(AG)$ there exists an $x \in \mathbb{R}^m$ such that $y = (AG)x = A(Gx) \in Col(A)$. This means that $Col(AG) \subseteq Col(A)$.
2. For any $y \in Col(A)$ there exists an $x \in \mathbb{R}^n$ such that $y = Ax = (AGA)x = (AG)(Ax) \in Col(AG)$. This means that $Col(A) \subseteq Col(AG)$.

Therefore, AG and A have the same column spaces, and AG is projection matrix onto this space.

A square matrix P is called an *orthogonal projection* matrix if $P = P^\top$ and $P = P^2$. For any matrix $A \in \mathbb{R}^{m \times n}$ the matrix AA^+ is an orthogonal projection matrix (onto $Col(A)$). This follows directly from the definitions. For any matrix $P \in \mathbb{R}^{n \times n}$ and vector $x \in \mathbb{R}^n$, $x = Px + (I - P)x$. This implies that orthogonal projections produce orthogonal decomposition of vectors [12].

Computation of the pseudoinverse

Let $U \in \mathbb{R}^{m \times n}$ be any orthogonal matrix, then $U^+ = U^\top$ [12]. Let $A \in \mathbb{R}^{m \times n}$ be any matrix with a singular value decomposition $A = U\Sigma V^\top$, then $A^+ = V\Sigma^+U^\top$ *Proof:*

1. $AA^{+}A = U\Sigma V^{\top} \cdot V\Sigma^{+}U^{\top} \cdot U\Sigma V^{\top} = U\Sigma\Sigma^{+}\Sigma V^{\top} = U\Sigma V^{\top} = A$
2. $A^{+}AA^{+} = V\Sigma^{+}U^{\top} \cdot U\Sigma V^{\top} \cdot V\Sigma^{+}U^{\top} = V\Sigma^{+}\Sigma\Sigma^{+}U^{\top} = V\Sigma U^{\top} = A^{+}$
3. $AA^{+} = U\Sigma V^{\top} \cdot V\Sigma^{+}U^{\top} = U\Sigma\Sigma^{+}U^{\top}$ (symmetric)
4. $A^{+}A = V\Sigma^{+}U^{\top} \cdot U\Sigma V^{\top} = V\Sigma^{+}\Sigma V^{\top}$ (symmetric)

Least squares

Consider the system of linear equations $Ax = b$ where $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^{m}$. When b is in the column space of A there is at least on or more solutions to the system. If b is not in the column space of A then there are no exact solutions, but it may still be desirable to find x^{*} which is *closest* to the solution. In this case the problem instead is to find a vector x^{*} such that $\|Ax^{*} - b\| \leq \|Ax - b\|$ for all x. The solution x^{*} is the best approximate solution: $x^{*} = A^{+}b$. *Proof:* For any $x \in \mathbb{R}^{m}$,

$$Ax - b = Ax - b - AA^{+}b + AA^{+}b = A(x - A^{+}b) + (I - AA^{+})(-b).$$

From the previous results it follows that $I - AA^{+}$ is an orthogonal projection onto the complement of the range of A. This means that the right-hand side is a summation of two orthogonal vectors. Using Pythagorean theorem for the vector norms:

$$\begin{aligned}\|Ax - b\|^{2} &= \|A(x - A^{+}b)\|^{2} + \|(I - AA^{+})(-b)\|^{2} \\ &= \|A(x - x^{*})\|^{2} + \|Ax^{*} - b\|^{2} \leq \|Ax^{*} - b\|^{2}\end{aligned}$$

Therefore, the norm of the residual is at its minimum when $x = x^{*}$ [22].

Appendix B

Abbreviations

Abbreviation	**Description**
BFGS	Broyden–Fletcher–Goldfarb–Shanno algorithm
CNN	Convolutional neural network
FFN	Feedforward neural network
ISTA	Iterative shrinkage and thresholding algorithm
LM	Levenberg–Marquardt algorithm
MIMO	Multiple–input multiple–output
MSE	Mean squared error
SVD	Singular Value Decomposition
VP	Variable Projection

Bibliography

[1] Gene H Golub and Victor Pereyra. “The differentiation of pseudo-inverses and nonlinear least squares problems whose variables separate”. In: *SIAM Journal on numerical analysis* 10.2 (1973), pp. 413–432.

[2] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.

[4] Jorge Nocedal and Stephen J Wright. *Numerical optimization.* Springer, 1999.

[5] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), pp. 533–536.

[6] Nir Shlezinger et al. “Model-based deep learning”. In: *Proceedings of the IEEE* (2023).

[7] John R Hershey, Jonathan Le Roux, and Felix Weninger. “Deep unfolding: Model-based inspiration of novel deep architectures”. In: *arXiv preprint arXiv:1409.2574* (2014).

[8] Karol Gregor and Yann LeCun. “Learning fast approximations of sparse coding”. In: *Proceedings of the 27th international conference on international conference on machine learning.* 2010, pp. 399–406.

[9] Vishal Monga, Yuelong Li, and Yonina C Eldar. “Algorithm unrolling: Interpretable, efficient deep learning for signal and image processing”. In: *IEEE Signal Processing Magazine* 38.2 (2021), pp. 18–44.

[10] Neev Samuel, Tzvi Diskin, and Ami Wiesel. “Learning to detect”. In: *IEEE Transactions on Signal Processing* 67.10 (2019), pp. 2554–2564.

[11] Christian Szegedy et al. “Going deeper with convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[12] Avrim Blum, John Hopcroft, and Ravindran Kannan. *Foundations of data science*. Cambridge University Press, 2020.

[13] Gene Golub and Victor Pereyra. “Separable nonlinear least squares: the variable projection method and its applications”. In: *Inverse problems* 19.2 (2003), R1.

[14] Dianne P O’Leary and Bert W Rust. “Variable projection for nonlinear least squares problems”. In: *Computational Optimization and Applications* 54 (2013), pp. 579–593.

[15] Péter Kovács et al. “VPNET: Variable projection networks”. In: *International Journal of Neural Systems* 32.01 (2022), p. 2150054.

[16] James Townsend. *Differentiating the singular value decomposition*. Tech. rep. Technical Report 2016, 2016.

[17] Xinlong Feng and Zhinan Zhang. “The rank of a random matrix”. In: *Applied mathematics and computation* 185.1 (2007), pp. 689–694.

[18] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[19] Carl Böck et al. “ECG beat representation and delineation by means of variable projection”. In: *IEEE Transactions on Biomedical Engineering* 68.10 (2021), pp. 2997–3008.

[20] George B Moody and Roger G Mark. “The impact of the MIT-BIH arrhythmia database”. In: *IEEE engineering in medicine and biology magazine* 20.3 (2001), pp. 45–50.

[21] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[22] Ross MacAusland. “The moore-penrose inverse and least squares”. In: *Math 420: Advanced Topics in Linear Algebra* (2014), pp. 1–10.

List of Figures

List of Tables