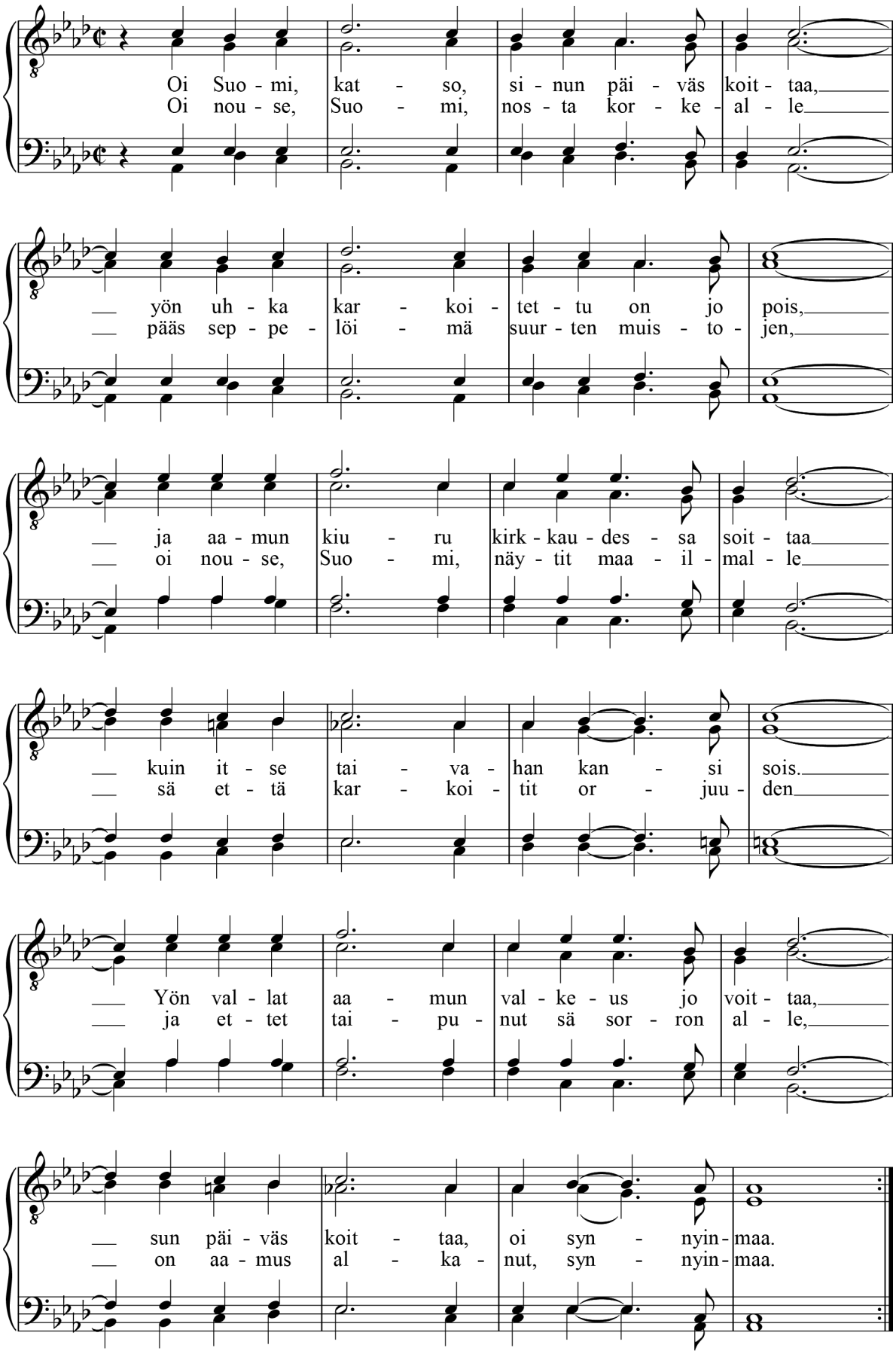
Universidade de Aveiro
2016
Departamento de Eletrónica, Telecomunicações e Informática

PEDRO MIRASSOL TOMÉ

PRÉ-DISTORÇÃO NEURONAL ANALÓGICA DE AMPLIFICADORES DE POTÊNCIA

ANALOG NEURAL PREDISTORTION OF POWER AMPLIFIERS

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações.



In *Finlandia*, J. Sibelius

o júri

presidente	Prof. Doutor Paulo Miguel Nepomuceno Pereira Monteiro professor associado da Universidade de Aveiro
arguente externo	Prof. Doutor Henrique Manuel de Castro Faria Salgado professor associado da Faculdade de Engenharia da Universidade do Porto
vogal	Prof. Doutor Telmo Reis Cunha professor auxiliar da Universidade de Aveiro

agradecimentos

I would like to express my gratitude to my thesis supervisors, Doctor Mikko Valkama, from the Tampere University of Technology, Finland, and Doctor Telmo Cunha, from the University of Aveiro, Portugal, for their always immediate support and for putting up with my very self-reliant work methodology.

I would like to thank Doctor Olli-Pekka Lundén, from the Tampere University of Technology, for his teachings and the interest he showed in my work; Doctor Tapio Elomaa, from the Tampere University of Technology, for his contribution to my understanding of the Artificial Intelligence concepts contained within this thesis; Mahmoud Abdelaziz, doctoral student at the Tampere University of Technology, for his resources on behavioral modeling; and Doctor Nuno Lau, from the University of Aveiro, as well as Abbas Abdolmaleki, doctoral student at the University of Aveiro, for their knowledge on Evolution Strategies.

I would like to acknowledge the Erasmus+ exchange program, without which I would not have had the opportunity to study at the Tampere University of Technology and to learn about the Finnish people and their customs.

My most sincere appreciation to the Teekkarikuoro, the student choir of the Tampere University of Technology. Of all the good things that happened to me in Finland, being a part of you was the best. There are no words which express how grateful I am for our time together and the feelings of us, through music, being one. I hope my playing of Rachmaninoff's *Élégie* was to your liking, and I hope it touched you as you did to me.

To my friends, Diogo Saraiva and William Robert. Keepo

Finally, to my parents and my sister. Thank you.

palavras-chave

linearização, pré-distorção, pré-distorção analógica, pré-distorção neuronal, rede neuronal, aprendizagem automática, temporal difference, otimização, estratégias de evolução, cma-es.

resumo

As especificações das redes de telecomunicações de quinta geração ultrapassam largamente as capacidades das técnicas mais modernas de linearização de amplificadores de potência como a pré-distorção digital. Por esta razão, esta tese propõe um método de linearização alternativo: um pré-distorçor analógico, à banda base, constituído por uma rede neuronal artificial. A rede foi treinada usando três métodos distintos: avaliação de política através de TD(λ), otimização por estratégias de evolução como CMA-ES, e um algoritmo original de aproximações sucessivas. Apesar do TD(λ) não ter produzido resultados de simulação satisfatórios, os resultados dos outros dois métodos foram excelentes: um NMSE entre as funções de transferência pretendida e efetiva do amplificador pré-distorcido até -70 dB, e uma redução total das componentes de distorção do espetro de frequência de um sinal GSM de teste. Apesar das estratégias de evolução terem alcançado este nível de linearização após cerca de 4 horas de execução contínua, o algoritmo original consegue fazê-lo numa questão de segundos. Desta forma, esta tese abre caminho para que se cumpram as exigências das redes de nova geração.

keywords

linearization, predistortion, analog predistortion, neural predistortion, neural network, reinforcement learning, temporal difference, optimization, evolution strategies, cma-es.

abstract

Fifth-generation telecommunications networks are expected to have technical requirements which far outpace the capabilities of modern power amplifier (PA) linearization techniques such as digital predistortion. For this reason, this thesis proposes an alternative linearization method: a base band analog predistorter consisting of an artificial neural network. The network was trained through three very distinct methods: policy evaluation using TD(λ), optimization using evolution strategies such as CMA-ES, and an original algorithm of successive approximations. While TD(λ) proved to be unsuccessful, the other two methods produced excellent simulation results: an NMSE between the target and the predistorted PA transfer functions up to -70 dB, and the complete elimination of distortion components in the frequency spectrum of a GSM test signal. While the evolution strategies achieved this level of linearization after about 4 hours of continuous work, the original algorithm consistently does so in a matter of seconds. In effect, this thesis outlines a way towards the meeting of the specifications of next-generation networks.

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

5G	Fifth generation of mobile telecommunications
AM-AM	Amplitude-to-Amplitude modulation
AM-PM	Amplitude-to-Phase modulation
ANN	Artificial Neural Network
APD	Analog Predistortion
CMA-ES	Covariance Matrix Adaptation Evolution Strategy
CMOS	Complementary Metal-Oxide Semiconductor
DC	Direct Current
DPD	Digital Predistortion
GSM	Global System for Mobile Communications
IMD	Intermodulation Distortion
NMSE	Normalized Mean Square Error
PA	Power Amplifier
PD	Predistorter
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase Shift Keying
RC	Resistance-Capacitance
RF	Radio Frequency
STA	Successive Target Approximation
TD	Temporal Difference
TD(λ)	Generalized Temporal Difference
TDMA	Time Division Multiple Access
TDNN	TD(λ) Neural Network
VSPA	Virtual Static Power Amplifier

LIST OF FIGURES

1. INTRODUCTION

The work reported in this dissertation was done, in part, under the supervision of Doctor Mikko Valkama, of the Tampere University of Technology, Finland. A significant portion of the text within this document was also presented to the same institution as a dissertation [1].

While the requirements and specifications for fifth-generation (5G) mobile systems and services have yet to be fully defined, some goals of the next generation of mobile networks are already very clear: a tremendous increase in connection density and speed (over 1 Gb/s downlink bit rate) and a similarly significant decrease in connection latency (under 1 ms roundtrip delay) [2].

However desirable, these advancements impose changes not only on the hardware that constitutes cellular networks, but on their topology as well. To be able to yield such high bit rates at such low latencies, cellular base station transmitters will need to have wider operational bandwidths – on the order of 500 to 1000 MHz [3], in contrast to the few tens of MHz that current base stations possess –, and their center frequencies will have to be adjusted to higher regions of the spectrum – reportedly as high as 6 to 300 GHz [2].

Radiation at such high frequencies will evidently have limiting effects on the propagation of radio frequency (RF) signals through buildings and objects, thus leading to a structural change in network architectures: instead of network coverage being provided by central, hugely encompassing, high power transmitters, it will instead be done through the deployment of swarms of small, low power, distributed transmitters [2,4].

Ultimately, all of these changes, from the higher signal bandwidths to the lower power levels of the transmitting amplifiers, contribute to one critical outcome: the downfall of digital predistortion (DPD) as a viable linearization technique. Not only will the bandwidth of 5G power amplifiers (PAs) be too wide for the limited processing speed of state-of-the-art digital processors, but also their own power consumption (proportional to their switching frequency) will be too great compared to the power level of the PAs they linearize, thus defeating any sort of effort for increased power efficiency – in other words, it would not be sensible to linearize a 1 W power amplifier with a 20 W digital processor.

Naturally, the need for a means of PA linearization will remain: without it, achieving any of the next-generation (or even current-generation) goals would be impossible. New ideas must, therefore, be proposed and explored, and that is what this dissertation is all about.

1.1. The Dissertation

Extraordinary needs require extraordinary measures, and thus a new line of thinking must begin. The aim of this dissertation is not to solve the problem of replacing 20 years' worth of research and technological development on digital predistortion, but to start the discussion on one way in which it might be possible to do so – eventually.

This dissertation builds upon analog predistortion (APD), the precursor to digital predistortion. Due to very significant technical advancements in digital electronics at the turn of the century, APD has been mostly put aside in favor of DPD. However, a small set of researchers have realized that the requirements for next-generation telecommunications will prove to be insurmountable for DPD, thus promoting the authoring of new literature on APD [5–7], albeit at a still relatively slow pace.

Another topic this dissertation builds upon is the use of artificial neural networks (ANNs) as predistortion devices, which has also been explored in the past. Most existing publications on neural predistortion are about DPD [8–10], since only recently has it been possible to implement ANNs as analog circuits. For this reason, the literature on this topic is still lacking [11,12].

The headline of this work is the linearization of power amplifiers using the predistortion technique, performed at base band using analog implementations of artificial neural networks (ANNs). Three very distinct methods of training the predistorting ANNs were tested: policy evaluation using TD(λ) learning, which proved to be unsuccessful; optimization using evolution strategies such as CMA-ES, which proved to be very successful, yet slow; and a novel, custom-made algorithm which proved to be very successful and exceptionally fast.

2. LINEARITY AND THE LACK THEREOF

Power amplifiers are some of the most fundamentally important devices in radio frequency telecommunications, since they are that which guarantees an information-carrying signal is of sufficiently high power level to be successfully transmitted by an antenna as small as a cell phone's or as large as a broadcasting radio station's.

Power amplifiers typically handle large amounts of power (for varying degrees of “large” – power ratings can vary by several orders of magnitude depending on the application), which means that power efficiency is of the highest importance: if efficiency is low, a cell phone's battery life may be severely compromised or the operational cost of a base station’s cooling system may become unreasonably high.

On the other hand, if an amplifier is not perfectly linear – that is, if it does *anything* to the input signal other than to increase its power level (besides introducing a constant delay) –, the information that is supposed to be transmitted through the succeeding antenna may be corrupted.

And therein lies the problem. In general, the more linear an amplifier is, the less efficient it is [13]. For example, a class A amplifier (such as the textbook common emitter, single transistor amplifier) has very high linearity, but a theoretical (absolute maximum) efficiency limit of 50%. This isn't as unintuitive as it might seem – consider a class D amplifier, which is ideally a switch: because it is a switch, it can either be *on* or *off*, making it extremely nonlinear; but also because it is a switch, its theoretical efficiency is 100%, since “an ideal switch in its *on* state conducts all the current but has no voltage loss across it and therefore no heat is dissipated, and when it is *off* it has the full supply voltage across it but no leak current flowing through it, and again no heat is dissipated”.

In short, typical applications demand high efficiency power amplifiers; because they are highly power efficient, they are very nonlinear, and because they are very nonlinear, the amplified signals – as well as the information they carry – are distorted. To solve this, these

amplifiers are linearized in a variety of ways, resulting in a system that is both highly power efficient and highly linear: the best of both worlds.

2.1. Linearity: An Intuitive View

Static linearity can be formally defined through two distinct properties: superposition, $F(x_1 + x_2) = F(x_1) + F(x_2)$, and first-degree homogeneity, $F(\alpha x) = \alpha F(x)$. Essentially, this means that the net response of a linear system to a number of simultaneous inputs is the sum of the responses of the system to each individual input.

It is much easier, however, to think of a static linear system as one whose input/output response is, as the name implies, *linear*: a line. This line cannot have an offset, however, as there should be no output when there is no input. See Figures 2.1 and 2.2 for examples of linear and nonlinear static input/output responses.

On a more general and formal note, a linear system – be it static or dynamical –, is one whose variation of its state vector x is defined as in (2.1), where A is a constant matrix, b is a constant vector, and u is the input vector.

$$\dot{x} = Ax + bu \tag{2.1}$$

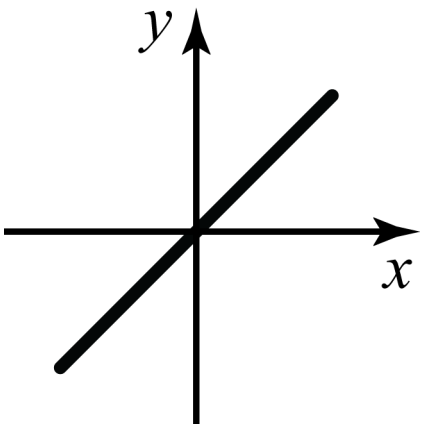


Figure 2.1. A linear static system.

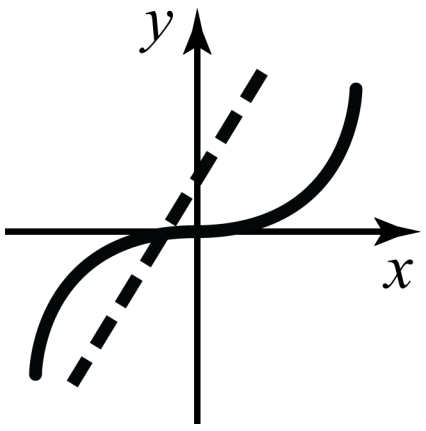


Figure 2.2. Nonlinear static systems.

2.2. Effects of Nonlinearity

It has been established that nonlinearity produces distortion in signals and has the potential to corrupt the information they carry. But how so? How can that be quantified?

Consider an amplifier whose behavior can be modeled by a simple third-order (nonlinear) polynomial with input $x(t)$ and output $y[x(t)]$: $y[x(t)] = a_1x(t) + a_2x(t)^2 + a_3x(t)^3$. Consider also a signal composed of two close tones, one at frequency ω_1 and amplitude X_1 and another at frequency ω_2 and amplitude X_2: $x(t) = X_1\cos(\omega_1 t) + X_2\cos(\omega_2 t)$. The response of the amplifier to the signal is the sum of various tones at the following frequencies [14]:

- Base-band: $\omega_2 - \omega_1$
- Coincident with the signal: ω_1 , ω_2
- In-band distortion: ω_1 , ω_2 , $2\omega_1 - \omega_2$, $2\omega_2 - \omega_1$
- 2nd harmonic: $2\omega_1$, $\omega_1 + \omega_2$, $2\omega_2$
- 3rd harmonic: $3\omega_1$, $2\omega_1 + \omega_2$, $\omega_1 + 2\omega_2$, $3\omega_2$

Clearly, the response of the amplifier is not an amplified version of its input, otherwise the output tones would only be those coincident in frequency with the input ones; the spectrum has, therefore, expanded – see Figures 2.3 and 2.4 for a graphical example of a slightly more complex PA model (fifth-degree polynomial), showing only the fundamental frequency band.

High order harmonics and base band distortion are not exactly the problem, because they can be easily filtered out by the amplifier's output matching network. The real problem is in having to deal with spurious (unwanted) tones very near the input tones, because they would require filters with extremely high Q-factors (sharp frequency responses) to be eliminated, and those are not at all trivial to design. Also, filtering would not be reasonable for transceivers operating with multiple channels (at distinct frequency locations, although in nearby regions of the spectrum). Thus, intermodulation distortion (IMD) tones cannot be filtered – they have to be suppressed resorting to a variety of linearization techniques.

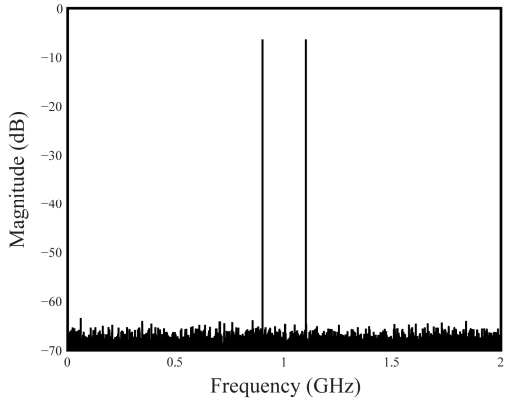


Figure 2.3. The spectrum of the input signal of a nonlinear device.

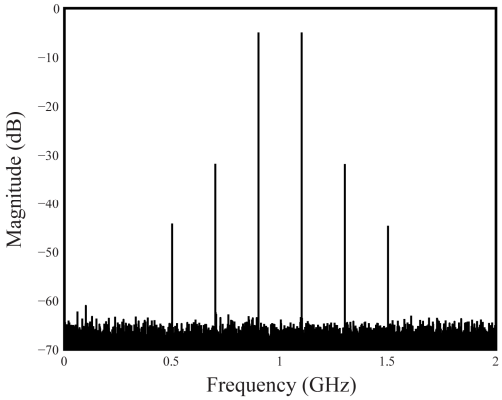


Figure 2.4. The spectrum of the output signal of a nonlinear device.

2.3. Linearization Techniques

Most linearization techniques fall into the four different categories explained in this section. Naturally, one can take advantage of a combination of them, producing fairly complex linearization circuits, but each of them may be used separately to great effect.

2.3.1. Power Back Off

Most power amplifiers have three operation regimes: at low powers, the amplifier is linear, with constant gain; when the amplifier approaches its saturation point, the device starts behaving nonlinearly and the gain starts decreasing; finally, when either the maximum rail voltage is reached or the maximum current is drawn, the amplifier fully saturates and its gain reaches its minimum – the amplifier cannot produce any more output power.

Power back off simply consists in operating an amplifier in its linear regime, “backing off” (or “away”) from the nonlinear ones; see Figure 2.5. Generally, the amount of back off power (say, 3 dB) is in respect to the device's 1 dB compression point, which is the point at which the power gain is 1 dB lower than its maximum value (the gain in the linear region, in the case of single-transistor class-A amplifiers).

The advantage of the employment of this technique is its extreme simplicity: either the input power is lowered so the amplifier operates exclusively in its linear region, or the supply voltage is increased so that the amplifier's linear region is extended. The disadvantage, however, is that the efficiency rapidly decreases with the increase of the back off power, since a linear amplifier is (usually) an inefficient one. Also, as a general rule, the higher the maximum power rating of an amplifier, the more expensive it is, so using a 200 W amplifier to produce a 100 W signal (3 dB back off) would certainly be more expensive than using a 100 W amplifier to produce the same signal.

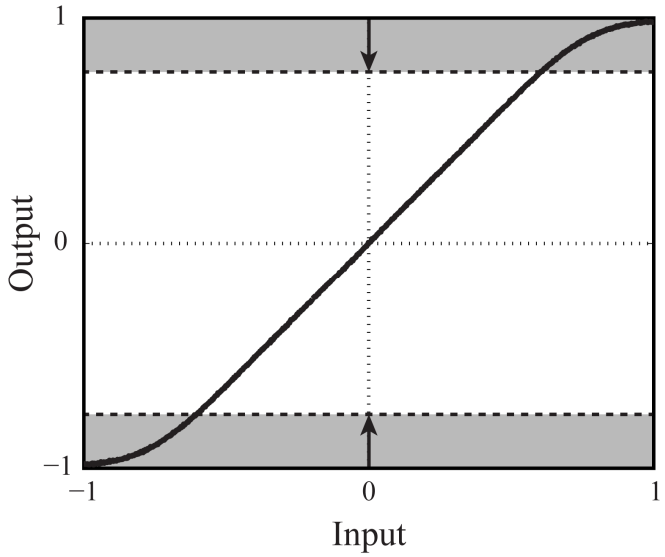


Figure 2.5. Power back off from the perspective of an amplifier's normalized voltage input/output response.

2.3.2. Cartesian Feedback

Most RF signals are generated through the modulation of a high frequency carrier signal using lower frequency data signals, called the in-phase (*I*) and quadrature (*Q*) signals. It is these *I* and *Q* components that define a system as "Cartesian", since they directly relate to a Cartesian representation of the transmitted signal (composition of two orthogonal vectors, *I* and *Q*), rather than a polar one (magnitude and phase).

The most distinguishing feature of Cartesian feedback [15] – and the fundamental concept behind it – is the use of a negative feedback loop to control each of the *input I and Q components* so that the *output I and Q components* of the amplifier correspond to an output composite signal that is a linearly amplified version of the input composite signal. In Cartesian terms, a system is said to be linear if its output (*I*, *Q*) vector is a scaled version of its input (*I*, *Q*) vector – their phases should, therefore, be equal.

The output of an RF amplifier is an RF signal, so, in order to perform the feedback of its *I* and *Q* output components, these must be extracted with a demodulator which reverses the up-conversion done by the modulator that mixes the input *I* and *Q* signals with the carrier signal. After extracting the output *I* and *Q* components, *I* and *Q* error signals (the difference between the respective *I* and *Q* input and output components) are fed to control systems that guarantee the linearity of the overall system. These control systems, represented as "H(s)" blocks in Figure 2.6, may be designed with classical techniques such as dominant pole compensation [15].

The advantage of the Cartesian feedback linearization technique is, similarly to the power back off technique, its fair simplicity and reasonable IMD suppression. Feedback systems are inherently slow, though, so this technique is only reliable for low base band frequencies – up to hundreds of kHz at most [16] –, so RF feedback is not even attempted: any phase shift from the feedback path would ruin the system's stability.

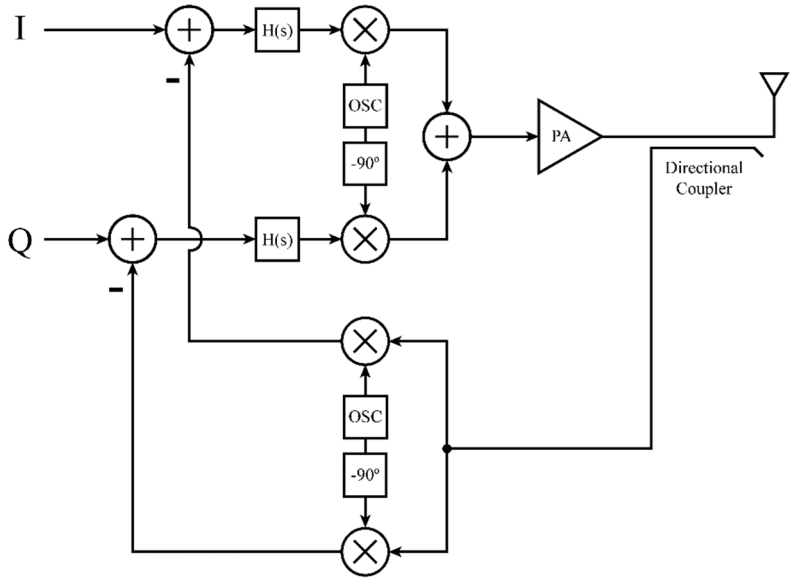


Figure 2.6. Cartesian feedback.

2.3.3. Feedforward Linearization

In a feedback loop, a sample of the controlled system's output is subtracted from a reference input signal, producing an error signal. Likewise, in a feedforward scheme a sample of the controlled system's output is also subtracted from a reference input signal, producing an error signal as well. (Naturally, if the system has a gain of A W/W then the sampled output should be attenuated by A W/W to achieve a proper difference or error signal; see Figure 2.7.)

The difference between the two architectures – feedback and feedforward – is how they use the error signal which carries the information of how exactly the actual system output differs from the intended, target output: in a feedback topology, the error signal is used as the input of a controller which adjusts the controlled system's output so it matches the reference signal, i.e., the error signal has an indirect consequence on the system's output; in a feedforward topology, the error signal is *directly subtracted* from the system's output, producing a new, error-free signal further down the road.

Consider the following example:

- An amplifier has a power gain of 10 and introduces some spurious signals, whose power shall be named D ("D" for "distortion"). [e.g., $D = 0.2$ W]
- Let X be the input of the amplifier. Then, the output of the amplifier is $Y = 10X + D$, that is, a 10 times amplified version of the input signal plus some D amount of distortion. [e.g., $X = 7$ W; $Y = 70.2$ W]
- Now, to get the error signal, E, the input and output signals are subtracted while taking into account the gain of the amplifier (so both signals are at the same power level), so $E = X - Y/10 = X - (10X + D)/10 = -D/10$. [$E = -0.02$ W]
- Finally, the feedforward part: the error signal is coupled (added) to the amplifier's output; again, the amplifier's gain has to be taken into consideration, so the error signal has to be multiplied by 10. The overall output of the linearized system is therefore $Y + 10E = 10X + D - D = 10X$, a perfectly amplified, distortion-free version of the input signal. [$Y + 10E = 70.2\text{ W} + 10 \times (-0.02\text{ W}) = 70$ W]

The main advantages of feedforward linearization are the wide operating bandwidth and the compensation of any sort of distortion produced by an amplifier – even that which is caused by the device's memory effects. The tradeoff, though, is the high complexity and the requirement of automatic adaptation to maintain performance specifications [16].

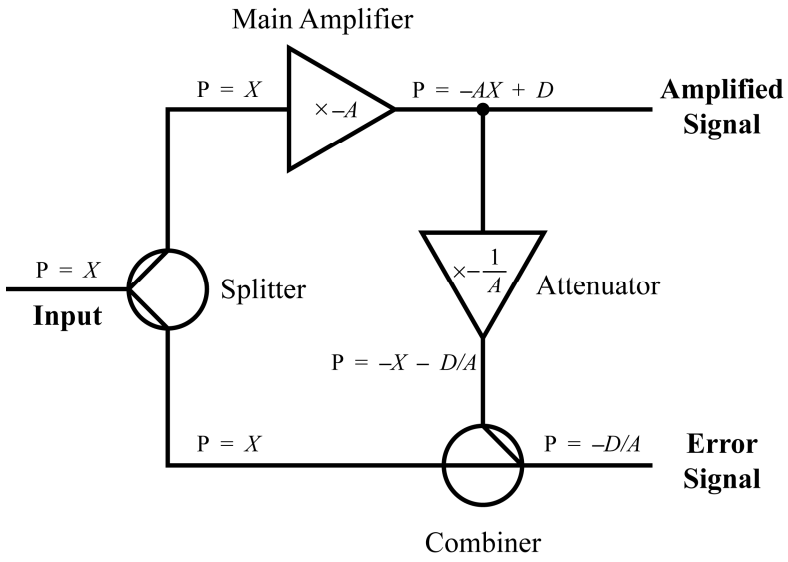


Figure 2.7. Error signal generation through signal cancellation.

A typical feedforward linearization system, schematized in Figure 2.8, consists of two circuits: a signal cancellation circuit and an error cancellation circuit.

The first circuit implements steps 1 to 3 of the previous example, that is, it produces a signal that *only contains the distortion* created by the power amplifier; it does this by attenuating the output of the amplifier (by an amount equal to the amplifier's gain) and combining the resulting signal with a copy of the input signal. Because these two signals have opposite phases, this essentially results in a subtraction, rather than an addition.

Finally, the second circuit implements step 4 of the previous example, that is, it amplifies the distortion signal extracted by the first circuit and couples it to the output of the amplifier. Similarly to the previous case, these two signals have opposite phases, so this essentially results in a subtraction. This means that the distortion generated by the amplifier is subtracted from the amplifier's own output signal, leaving a signal that is free of distortion and, by definition, a linearly amplified version of the input signal.

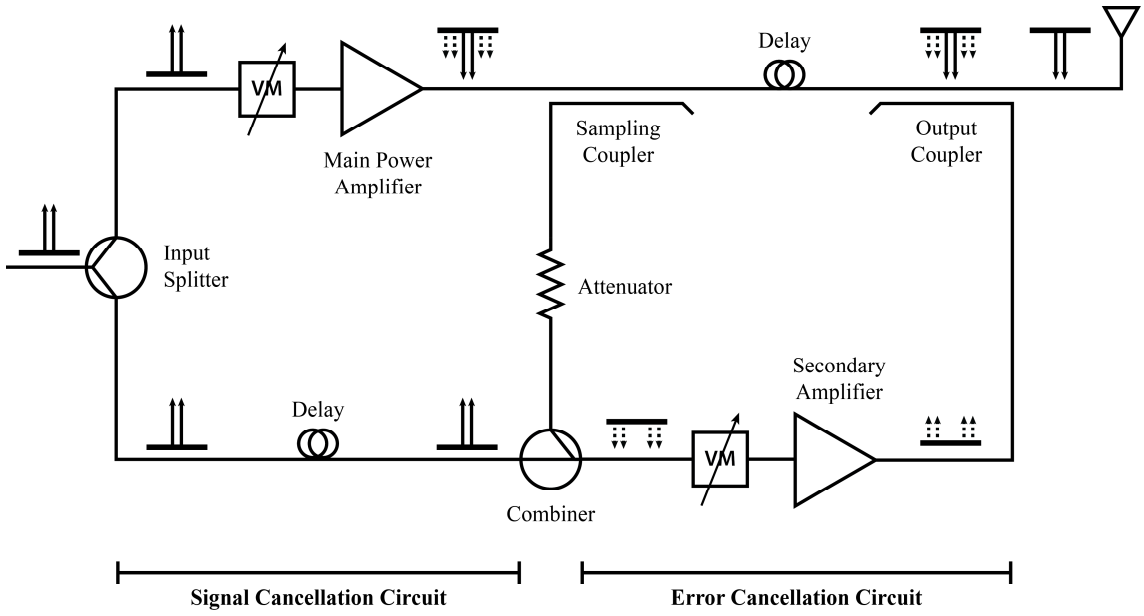


Figure 2.8. Feedforward linearization [16].

2.3.4. Predistortion

Predistortion [17], illustrated in Figure 2.9, is the act of distorting a signal before it is fed to a nonlinear system in such a way that the distortion generated by the system is exactly canceled by the distortion synthesized by the predistorter (PD), resulting in an overall linear cascade of two devices. As an example, consider a system that has an input/output transfer function of $y = x^3$, which is clearly nonlinear. If a predistorter with an input/output transfer function of $y = \sqrt[3]{(x)}$ is used, then the *cascade* of the PD and the system is $y = \left[\sqrt[3]{(x)}\right]^3 = x$ and the overall system is perfectly linear.

The main advantage of predistortion is its potential to achieve fantastic intermodulation distortion suppression, i.e., very high linearity. However, predistortion usually requires the physical modeling of the amplifier, which is extremely complex, since most amplifiers exhibit memory effects, that is, their outputs depend not only on the current input, but the input at previous times as well. These models, as well as the predistortion of the input signals, are usually implemented using digital processors, which means that the bandwidth of the input signals is either limited by the sampling rate or the processing speed of the digital predistorter.

A common modification of the basic concept of predistortion is Cartesian predistortion (Figure 2.10), which is the predistortion of the base band (low frequency) in-phase and quadrature components (*I* and *Q*) instead of the predistortion of the RF (high frequency) composite signal. Among other things, this greatly reduces the required bandwidth of the predistorter. While this is a welcome relaxation of performance specifications in the case of APD, it is the very basis of DPD, since the predistortion of the RF signal would require extremely fast analog/digital conversion units and even faster processing units.

Finally, a very common way of simplifying the modeling of an amplifier and the resulting predistortion algorithm is to forgo the modeling of the amplifier's non-electrical characteristics, like temperature dependence, ageing, and other very slow phenomena. These can be compensated by recalculating the parameters of the amplifier's model based on the measurement of its response to a set of test signals. This way, the slow drifts of the input/output response of the PA due to changing temperature and other causes can be compensated. This is called "adaptive predistortion".

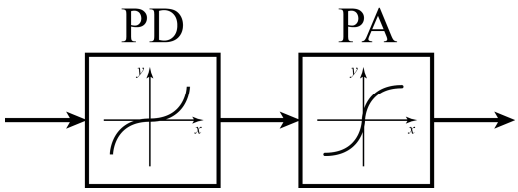


Figure 2.9. RF predistortion.

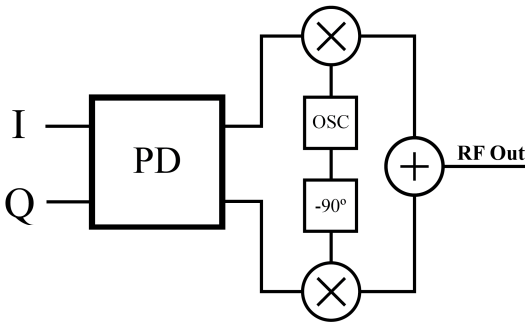


Figure 2.10. Cartesian predistortion.

3. ANALOG PREDISTORTION

Following Arthur C. Clarke's 1945 article on "Extra-Terrestrial Relays" [18] and John R. Pierce's 1955 article on "Orbital Radio Relays" [19], efforts towards global communications escalated along with a demand for higher transmission bandwidths at lower costs, leading to an increased interest in high order modulation techniques such as QPSK (Quadrature Phase Shift Keying) or QAM (Quadrature Amplitude Modulation) and multiple-access schemes such as TDMA (Time Division Multiple Access).

In order to achieve acceptable bit error rates and to meet the increasingly stringent spectral purity requirements of these data rate-increasing schemes, much attention was given between the late 1970s and the early 1980s to problems such as the linearization of high power microwave amplifiers used in satellite earth stations [20] and traveling wave tube amplifiers used in satellite transponders [21].

Because of the high power levels of these amplifiers, most linearization circuits consisted in the analog realization of the predistortion technique, applied not only to the microwave signals [21], but also (though less frequently) to the base band signals [20]. Regardless of the idiosyncrasy of each implementation, the great majority of the linearizers adhered to two main classes of predistortion circuits: cubic predistorters, and series diode predistorters [22].

In essence, cubic predistorters (Figure 3.1) couple the input signal to a distortion generator, a pair of antiparallel diodes, which produces exclusively odd-order harmonics of the input signal [23]. A variable phase shifter is used to guarantee a 180º phase difference between the input signal and the distortion signal, and a delay line is used to equalize the group delays of the two signals. Finally, a variable attenuator ensures the amplitude of the generated distortion matches that of the harmonic distortion produced by the predistorted device (such as an amplifier). This amplitude matching, along with the 180º phase difference between the clean signal and the generated distortion, results in an appreciable suppression of the spurious odd-order tones produced by the nonlinear predistorted device.

Series diode predistorters (Figure 3.2) consist of a single forward-biased series diode, which may be modeled as a nonlinear resistor with a parasitic capacitance – an RC phase shift network. The principle of operation is fairly straightforward: as per Shockley's diode equation, an increase in forward (RF) power results in a decrease in the diode's series resistance; this, in turn, provided that the series resistance is not too high [23], results in an expanding gain and a decreasing phase shift, effectively countering the predistorted amplifier's undesired AM-AM and AM-PM characteristics: amplitude compression and phase advance.

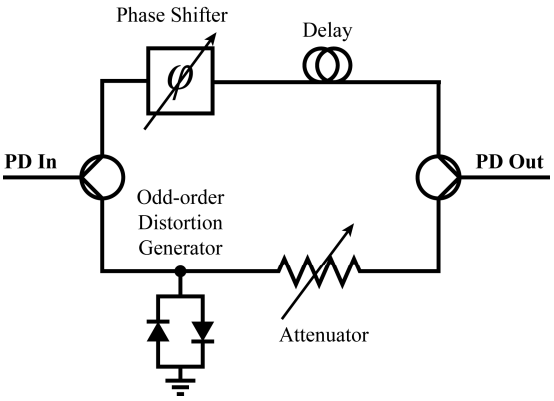


Figure 3.1. Cubing predistorter.

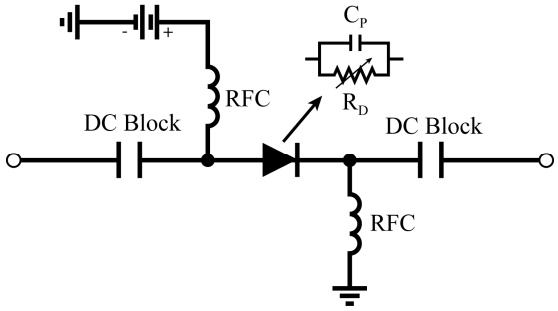


Figure 3.2. Series diode predistorter.

With the advent of high speed digital computing, analog predistortion plummeted into near oblivion and was swiftly replaced by more capable and more configurable digital predistortion schemes. Still, some research was done, mainly in the early 2000s, and not only did old analog predistortion technology improve, some new interesting ideas even came to light.

The first great advancement in analog predistortion was the refinement of the cubing predistorter, which led to the development of fully configurable, independently controllable "IMD generators" [24–27], that is, branched versions of the cubing predistorter that generate 3rd- and 5th-order (and higher) intermodulation distortion tones that can be independently scaled in magnitude and shifted in phase. See Figure 3.3 for an example of such a scheme.

The second great advancement – perhaps the most noteworthy, due to its novelty – was the realization that the AM-AM and AM-PM characteristics of a moderately nonlinear amplifier can be modelled by complex-valued polynomials of low order [28–30]. These polynomials, in turn, – or, rather, their inverse – can be approximated by transistor circuits based on the Gilbert cell [31] (Figure 3.4): a cascode circuit used as an analog four-quadrant multiplier and frequency mixer. A new class of CMOS circuits was therefore designed to implement high order polynomials (as high as 11th-order, for instance) with freely configurable coefficients and thus synthesize the inverse transfer characteristic of an amplifier – an almost ideal predistorter.

Finally, in the present decade, various novel analog predistortion schemes have surfaced, possibly in anticipation of the 5G networking challenges already summarized. These schemes include, among others, the bandwidth reduction of error signals [32], the use of mirror amplifiers [33], and lookup table-based, combined digital/analog predistortion systems [34].

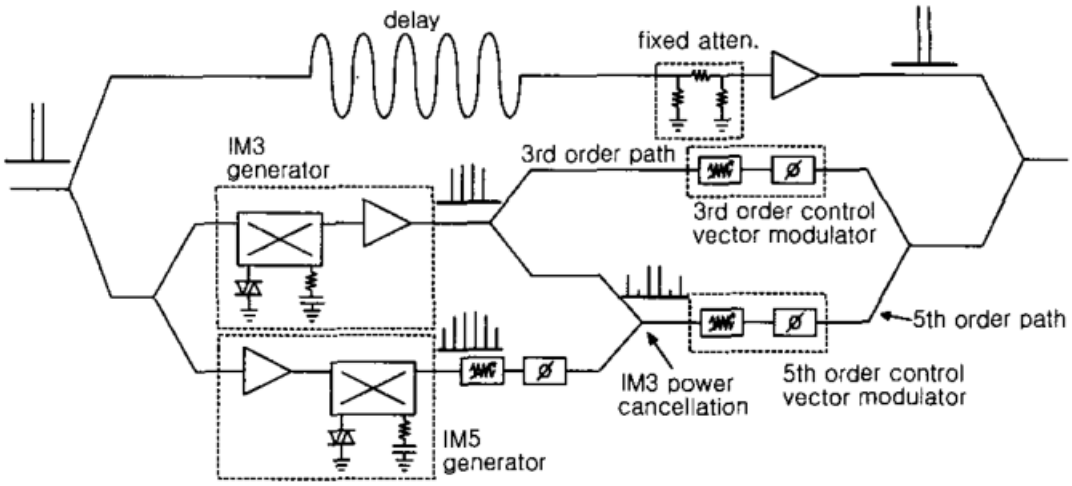


Figure 3.3. 5th-order IMD generating predistorter [25].

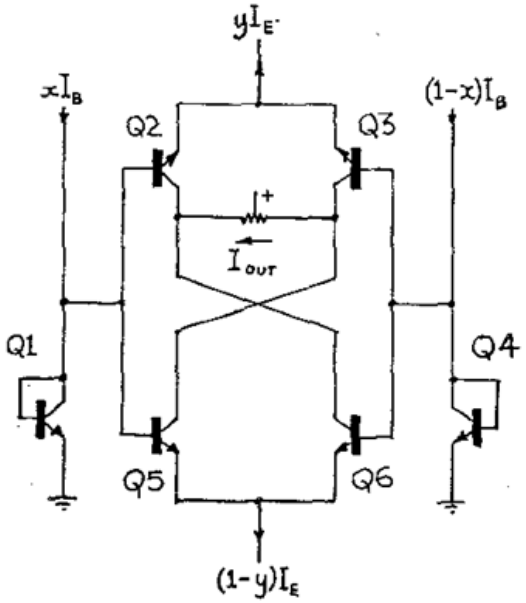


Figure 3.4. The Gilbert cell [31].

3.1. Proposed APD System Architecture

The system architecture of the proposed predistortion solution, schematized in Figure 3.5, consists of an analog feedforward artificial neural network that predistorts the base band *I* and *Q* components of a complex telecommunications signal. As usual, the predistorted signal is then transposed to a much higher frequency with an RF modulator and it is then fed to the PA that should be linearized. Naturally, some additional components – such as filters and intermediate amplification stages – are required for the successful implementation of the solution, but Figure 3.5 only illustrates the main blocks of the system for clarity purposes.

This base band architecture is ideal for an analog solution based on an artificial neural network because the bandwidth requirements of the ANN are much lower than they would be if it were used as an RF predistorter. An additional reason for having chosen a base band solution is the fact that the predistortion of the *I* and *Q* components of the complex signal is a matter of amplitude scaling, which means that the function the ANN is supposed to learn is real-valued. This contributes to a relatively simple model of the ANN-based predistorter and its learning algorithm. It should be noted that base band control is just as effective as RF control, because the scaling of the base band *I* and *Q* components results in both an amplitude and a phase change in the complex envelope RF signal that is fed to the PA.

The ANN is supposed to predistort the *I* and *Q* components of a telecommunications signal, so it should have at least two input neurons and two output neurons. The number of hidden neurons and layers can be adjusted to fit a variety of specifications. While only one hidden layer is required to approximate any function to an arbitrary level of precision [35], the number of neurons required to do so decreases with the number of layers, since the connection density (and the network's expressivity) also increases with the number of layers.

The ANN is intended to be an analog circuit, so the number of neurons and hidden layers should be carefully managed – not only because the former may be limited, but also because the number of input or output connections of each neuron may be constrained due to electrical loading and other practical aspects.

If the PA is assumed to be static, then a simple feedforward ANN with two input nodes should suffice. However, if the PA is assumed to be dynamic (that is, if it exhibits memory effects), then the ANN should exhibit a dynamic behavior as well. This can be achieved by using a recurrent ANN, in which the connections between neurons form directed cycles.

While a recurrent ANN would be able to implement the dynamic $\mathbf{R}^2 \rightarrow \mathbf{R}^2$ predistortion function, this is not an absolute necessity. Even though a PA's transfer function may be dynamic in an $\mathbf{R}^2 \rightarrow \mathbf{R}^2$ projection, it is, intuitively, static in an $\mathbf{R}^{2\times(M+1)} \rightarrow \mathbf{R}^2$ projection, where M is the memory depth (in samples) of the PA. Thus, the predistortion function can be a static $\mathbf{R}^{2\times(M+1)} \rightarrow \mathbf{R}^2$ function $[I_{PD}(k),\ Q_{PD}(k)] = f_{PD}[I(k),\ Q(k),\ I(k-1),\ Q(k-1),\ \cdots,\ I(k-M),\ Q(k-M)]$ implemented by a feedforward ANN with a pair of input neurons for each of the $M+1$ current and previous *I* and *Q* input states.

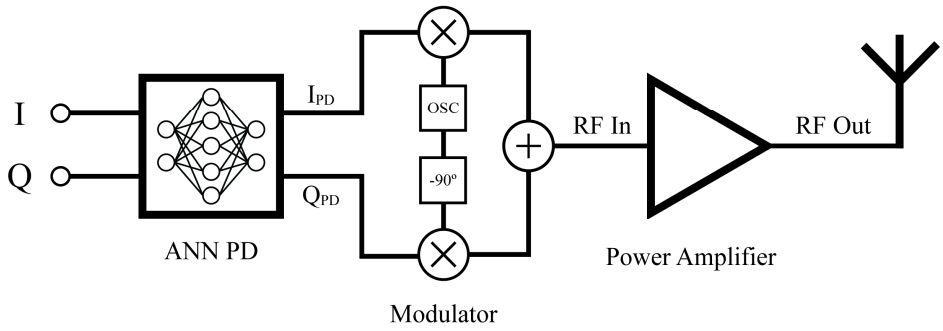


Figure 3.5. Predistortion system architecture.

3.2. Development and Test Setup

The proposed predistortion system was entirely simulated in Matlab. All predistortion efforts went towards the linearization of a model of a PA implemented as an obfuscated (P-code) Matlab function, called VirtualStaticPA (VSPA), which was provided by a third party.

This function models the static properties of a generic PA, such as gain compression and phase advance, and it focuses mainly on the distortion introduced by the PA – its maximum gain is just slightly above 0 dB. Moreover, this is a base band model, which means that the VirtualStaticPA function accepts the base band *I* and *Q* components of a signal as its input, denoted xI and xQ, and returns the base band *I* and *Q* components correspondent to its output amplified signal, denoted yI and yQ.

Figures 3.6 and 3.7 illustrate the transfer characteristics of the VSPA with respect to its input and output *I* and *Q* components. While both figures represent essentially same thing, the two distinct representations end up conveying different information.

The first figure makes it immediately clear that the transfer function of the modeled PA is a smooth $\mathbf{R}^2 \rightarrow \mathbf{R}^2$ projection, and provides insight into its amplitude modulation behavior: the PA saturates for values of xI and xQ close to 1 (one), and outputs a maximum value of yI and yQ of 1 (one).

The latter figure shows the same saturation effect, but it mainly addresses the representation of the phase modulation behavior of the PA, plotting the input and output (*I*, *Q*) vectors with connecting arrows which make the warping effect of the complex signal very noticeable.

Finally, Figure 3.8 illustrates the AM-AM (amplitude modulation) and AM-PM (phase modulation) behavior of the VSPA with respect to its input power.

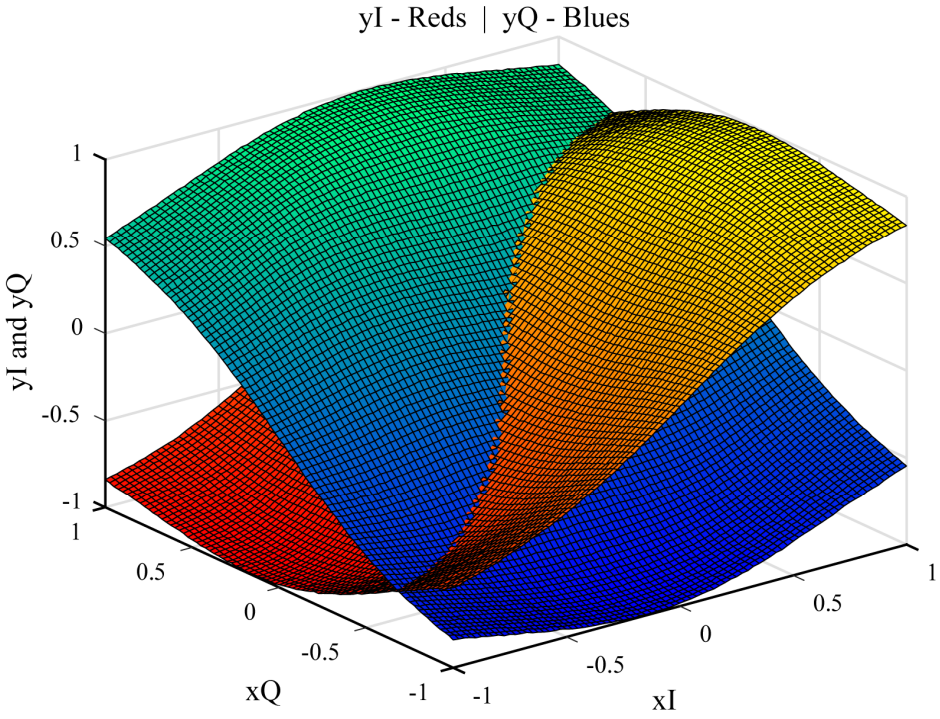


Figure 3.6. Transfer characteristics of the VSPA: view in the Cartesian space.

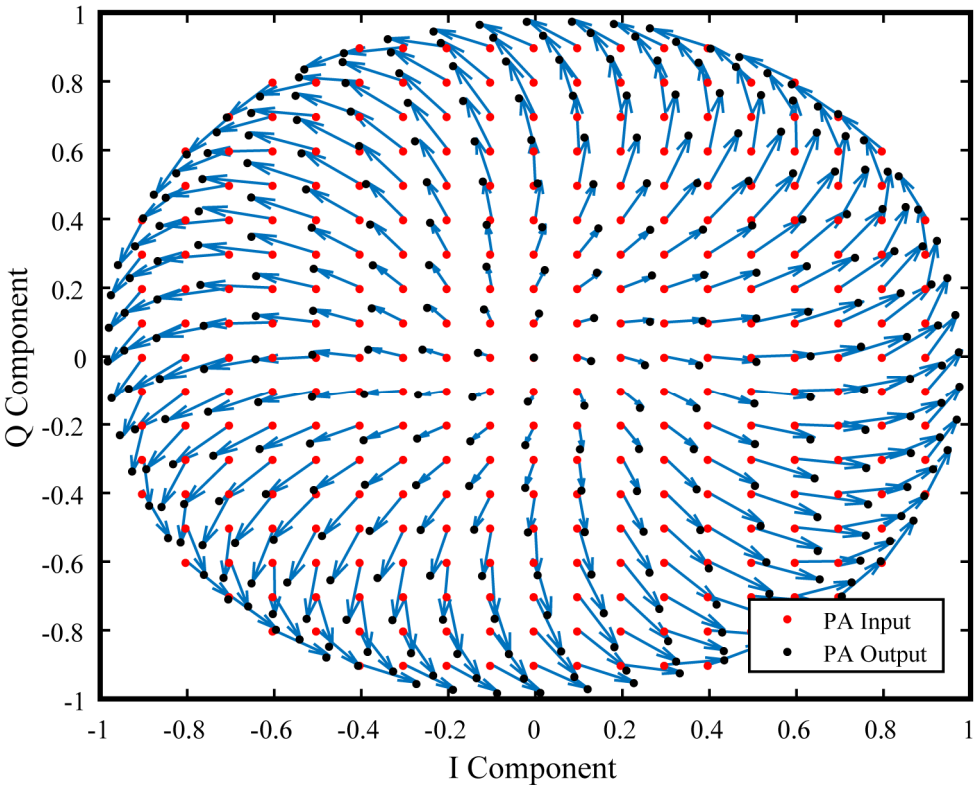


Figure 3.7. Transfer characteristics of the VSPA: view in the quadrature plane.

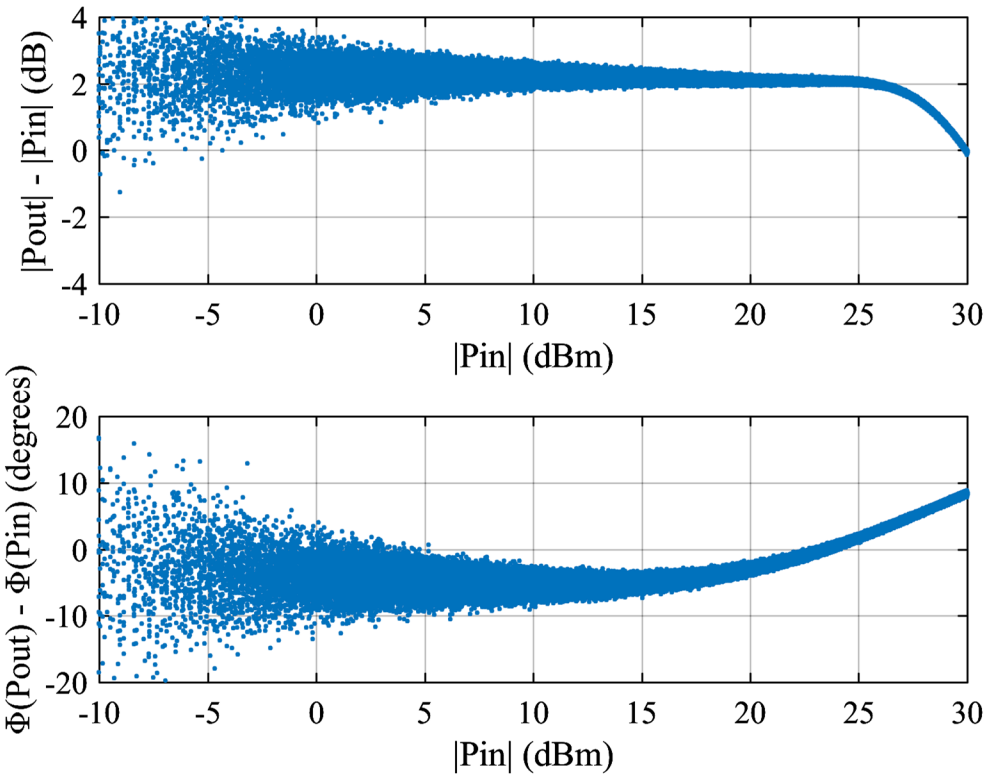


Figure 3.8. Transfer characteristics of the VSPA: gain and phase modulation.

As explained in a previous section, the most visible effect of the distortion introduced by a PA occurs in the frequency spectrum of its output signal. For this reason, a four-carrier GSM signal was used to monitor the spectral performance of the predistortion system. This signal, $RF(t)$, shown in Figures 3.9 and 3.10, is a composition of two base band signals, $I(t)$ and $Q(t)$, and is defined in (3.1).

$$RF(t) = I(t)\,cos(2\pi f_c t) - Q(t)\,sin(2\pi f_c t) \qquad (3.1)$$

It should be noted that the centering of the signal at $f_c = 10$ MHz was done merely for illustrative purposes. As stated, the VSPA is a base band model, so its inputs are the base band $I(t)$ and $Q(t)$ signals – not the compound $RF(t)$ signal. Similarly, its outputs are also base band quadrature signals; these are also shown modulated by a 10 MHz carrier signal throughout this document for illustrative purposes. Figure 3.11 contains the output spectrum of the natural response (i.e., without any sort of predistortion) of the VSPA to the GSM signal. Notice the presence of significant distortion tones, and the noise floor of –20 dBm.

The VSPA function also models the intrinsic noise of the amplifier using a function called random(), which explains the increased noise floor. This function can be bypassed by exploiting Matlab's function precedence order.

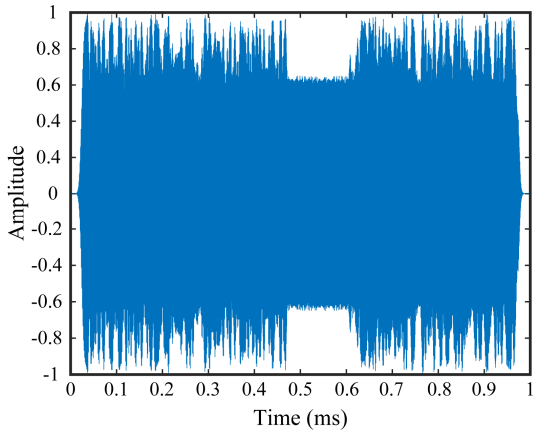


Figure 3.9. The input of the VSPA: a four-carrier GSM signal.

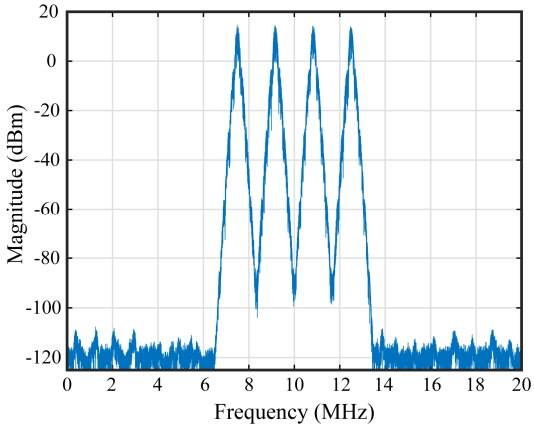


Figure 3.10. The spectrum of the input signal of the VSPA.

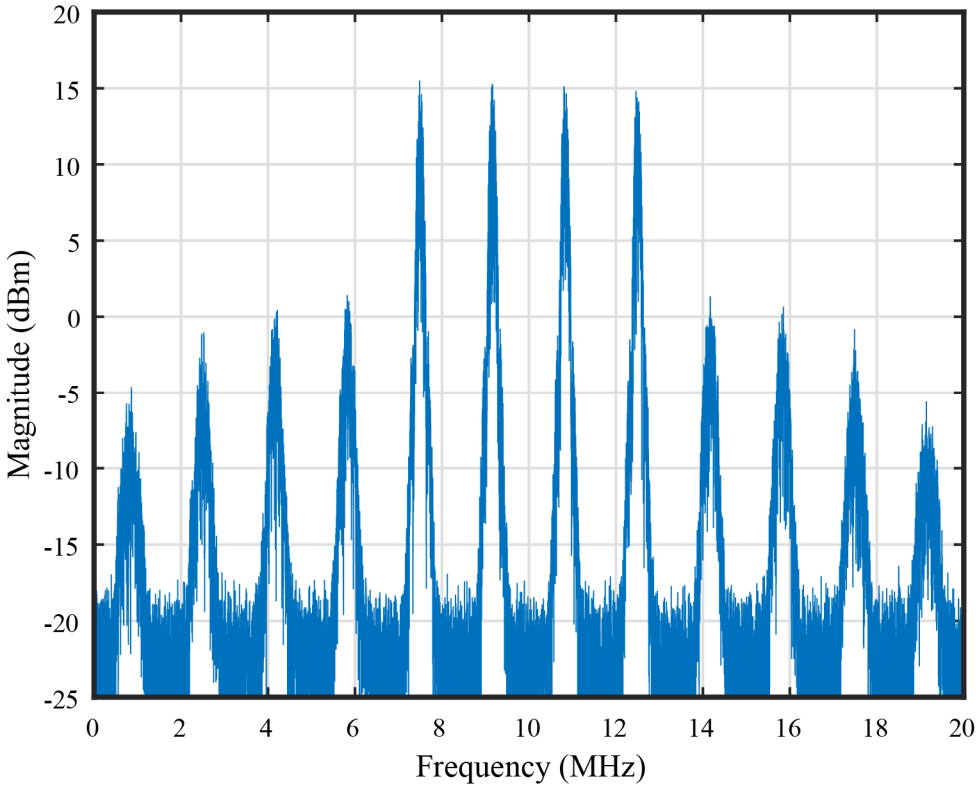


Figure 3.11. The output spectrum of the VSPA in response to the four-carrier input signal.

4. ARTIFICIAL NEURAL NETWORKS

Not unlike polynomials or Volterra series, artificial neural networks are a family of nonlinear function models which consist of a series of basic computational units, the neurons (akin to polynomials' power products), that are interconnected by means of model-defining weights (akin to polynomials' coefficients). Even though there are metrics such as the Vapnik-Chervonenkis dimension, the evaluation of the complexity of an ANN (similar to a polynomial's degree) has yet to be formally and unequivocally defined [36], though it is intuitive that it is related to the number of neurons it comprises and the way they are interconnected.

The basic computational unit of an ANN is the neuron, or node, illustrated in Figure 4.1. A neuron can have an arbitrary positive number of inputs x, one of which acts as a bias, and these are processed by an activation function Φ, which is selected by the ANN designer to calculate the neuron's activation a: its output. Typical activation functions include a purely linear transfer function (4.1) and the (logistic) sigmoid function (4.2), and these can be used at will throughout an ANN. A variety of sigmoid (meaning s-shaped) functions can be used for different levels of algorithmic optimization.

$$\Phi(z) = z \tag{4.1}$$

$$\Phi(z) = \frac{1}{1 + e^{-z}} \tag{4.2}$$

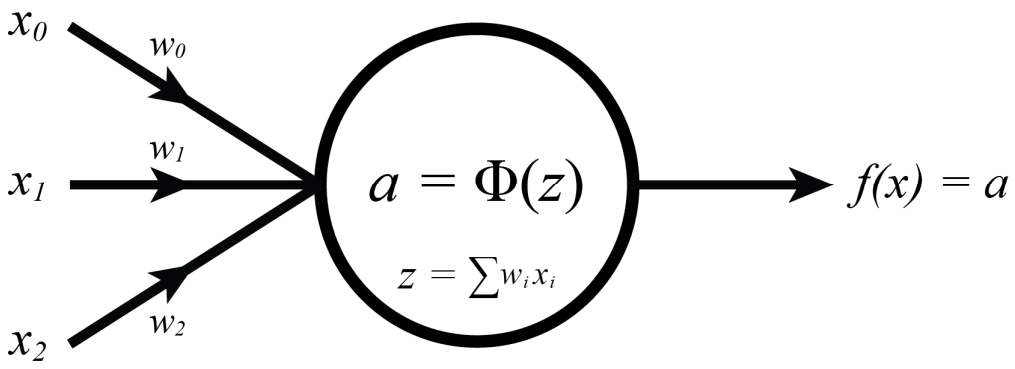


Figure 4.1. A neuron with three inputs.

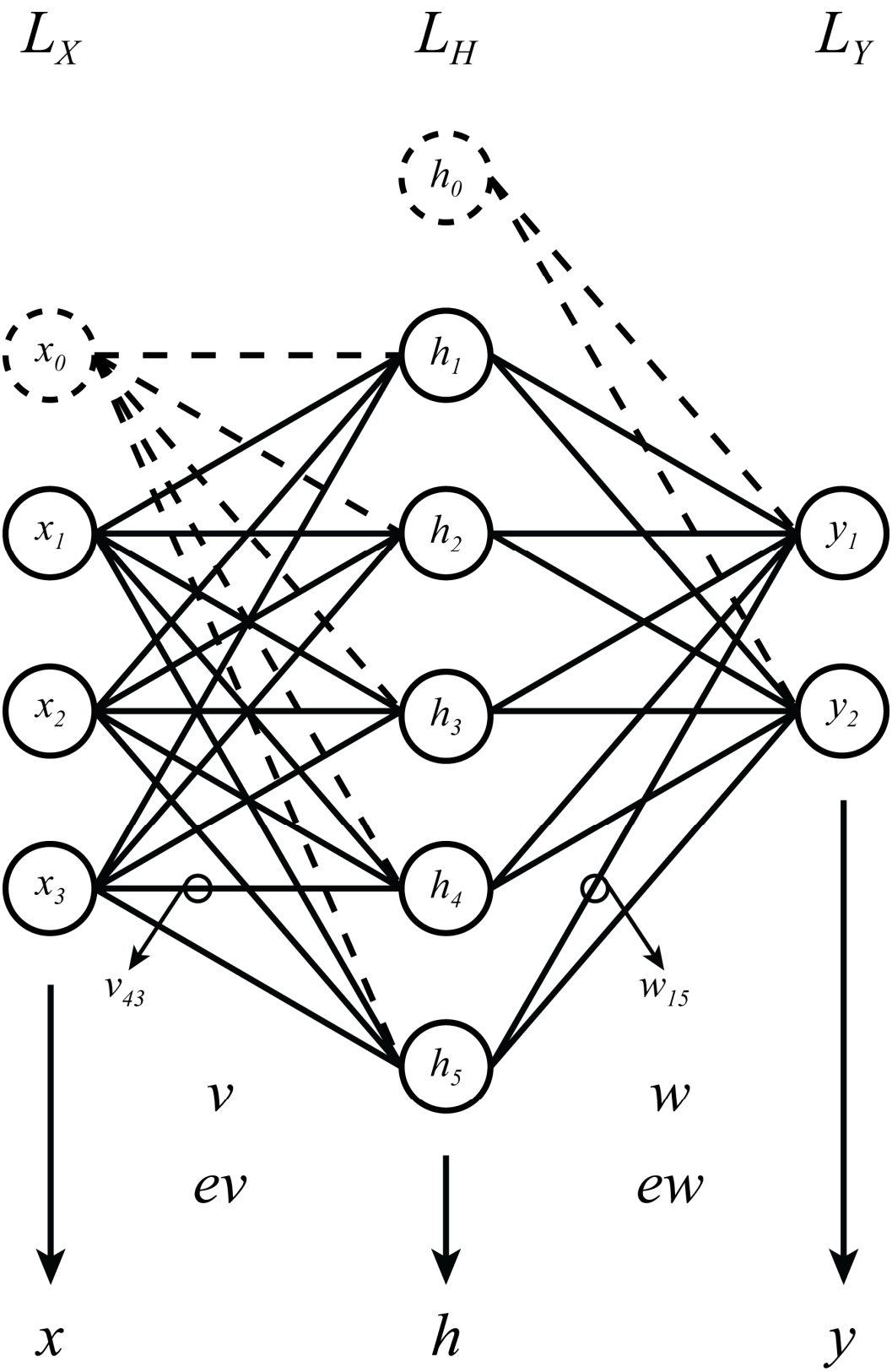


Figure 4.2. An example feedforward network with three input nodes, one hidden layer with five nodes, and two output nodes. Displayed as well are the biasing nodes for the hidden and output layers.

There is a nearly endless number of ways of arranging and interconnecting neurons in an ANN. There are, however, classical and established ways of doing so, such as the feedforward network illustrated in Figure 4.2. In a feedforward network, neurons are distributed between different, sequentially ordered layers: the input layer, a set of hidden layers, and an output layer. Each neuron in each layer connects to every neuron in the immediately succeeding layer, and there are no backward or intra-layer connections – meaning that there are no cyclical connections, hence the network's designation of "feedforward".

Feedforward ANNs are universal approximators [35]. This means that for any given continuous nonlinear function, there is at least one feedforward ANN that approximates it, in a closed and bounded input range (a compact set of $\mathbf{R}^n$), with an arbitrarily small error. This was proven for feedforward networks containing a single hidden layer of neurons with sigmoidal activation functions [37,38], though it stands to reason that more expressive networks, with more hidden layers, would perform at least as well as ANNs with a single hidden layer. Naturally, the output layer should have neurons with purely linear activation functions, otherwise the range of each of the network's output neurons would be constrained to the codomain of whatever sigmoidal activation function had been chosen.

4.1. ANNs as Analog Control Systems

Due to their massive expressive ability and structural simplicity, as well as ease of training, artificial neural networks have been used to solve board games such as backgammon [39] and Go [40], control physical systems such as inverted pendulums [41], and even predistort RF power amplifiers [8,9]. Despite their differences, all of these applications of ANNs have one thing in common: they are digital implementations. Recent technological advances have brought the possibility of reliably implementing ANNs as analog circuits. Further advances, such as commercially-available memristors, are expected to lead to even more robust and higher-performing analog ANNs.

Compared to the analog predistortion schemes presented in section 3, analog implementations of ANNs provide very substantial advantages. Not only are relatively simple ANNs much more expressive than 11th-order polynomials (the state-of-the-art

predistortion circuits until recently) in terms of function synthesis, but they also have an increased capability for generalization due to their saturating (sigmoidal) neurons, which is important when the predistorter's input range may not be clearly defined – high-order polynomials grow very quickly towards infinity outside the training sample space.

Furthermore, the bandwidth of each of an ANN's computational units (neurons) is similar to that of the predistorted signal, in contrast to the bandwidth of a polynomial's computational units (power products), which grows mostly linearly with the degree of each product (i.e., over an order of magnitude for an 11th-order polynomial predistorter).

4.2. Mathematical Formalization

Figure 4.2 represents a feedforward ANN with three layers: L_X, the input layer; L_H, the hidden layer; and L_Y, the output layer. Let there be the following symbols:

nX : the number of input nodes in L_X (excluding bias) – in this case, $nX = 3$;

nH : the number of hidden nodes in L_H (excluding bias) – in this case, $nH = 5$;

nY : the number of output nodes in L_Y – in this case, $nY = 2$;

x : a column vector, indexed as x_i, holding the node activations of L_X;

h : a column vector, indexed as h_j, holding the node activations of L_H;

y : a column vector, indexed as y_k, holding the node activations of L_Y;

v : a matrix, indexed as v_{ji}, holding the weights of the connections from L_X to L_H;

w : a matrix, indexed as w_{kj}, holding the weights of the connections from L_H to L_Y.

These symbols are defined as such, with example values based on Figure 4.2:

$$\underset{((nX+1)\times 1)}{x} = \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_{nX} \end{bmatrix} \quad \text{eg:} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\underset{((nH+1)\times 1)}{h} = \begin{bmatrix} h_0 \\ h_1 \\ \dots \\ h_{nH} \end{bmatrix} \quad \text{eg:} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \end{bmatrix}$$

$$\underset{(nY\times 1)}{y} = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_{nY} \end{bmatrix} \quad \text{eg:} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

$$\underset{(nH\times(nX+1))}{v} = \begin{bmatrix} v_{10} & v_{11} & \dots & v_{1nX} \\ v_{20} & v_{21} & \dots & v_{2nX} \\ \dots & \dots & & \dots \\ v_{nH0} & v_{nH1} & \dots & v_{nHnX} \end{bmatrix} \quad \text{eg:} \begin{bmatrix} v_{10} & v_{11} & v_{12} & v_{13} \\ v_{20} & v_{21} & v_{22} & v_{23} \\ v_{30} & v_{31} & v_{32} & v_{33} \\ v_{40} & v_{41} & v_{42} & v_{43} \\ v_{50} & v_{51} & v_{52} & v_{53} \end{bmatrix}$$

$$\underset{(nY\times(nH+1))}{w} = \begin{bmatrix} w_{10} & w_{11} & \dots & w_{1nH} \\ w_{20} & w_{21} & \dots & w_{2nH} \\ \dots & \dots & & \dots \\ w_{nY0} & w_{nY1} & \dots & w_{nYnH} \end{bmatrix} \quad \text{eg:} \begin{bmatrix} w_{10} & w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{20} & w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \end{bmatrix}$$

Thus, x_i is the activation of the i-th input node (the i-th input value, for $i > 0$), h_j is the activation of the j-th hidden node, y_k is the activation of the k-th output node, v_{ji} is the weight of the connection between the input node i and the hidden node j, and w_{kj} is the weight of the connection between the hidden node j and the output node k. One can read the matrix v, then, as a series of columns containing the weights of the connections of each input node to every hidden node (excluding the hidden bias node, which by definition has constant activation and thus does not have any input connections). Similarly, the matrix w can be read as a series of columns containing the weights of the connections of each hidden node (including the hidden bias node) to every output node.

The indexing of the v and w matrices is intentionally backwards. It would have been more aesthetic to define them as v_{ij} and w_{jk}, but this would have required the computation of their transpose matrices to perform forward propagation (explained below). The algorithmic performance gain is minimal, but it comes at essentially no cost.

To be precise, the nodes of the input layer aren't exactly neurons, but mere representations of the "input ports" of the ANN. There is no data processing or neural activation: input values just pass on through unchanged. This does not undermine the presented formalization, however, since it is trivial to devise neurons which would exhibit that exact behavior: a neuron, with no biasing and one data input with unitary weight, whose activation function is purely linear.

Furthermore, despite biasing being a property of the neurons and not the network architecture (even from the original, biological standpoint), it can be abstracted away as a node with constant activation (eg: $a = 1$) which connects to each neuron with weights proportional (or even equal) to the required biasing values. These biasing nodes and their connections are represented in Figure 4.2 with dashed lines, and they are referred to as the zeroth (0-th) node in each layer, if applicable. The output layer is the last layer, so, naturally, it doesn't contain bias nodes for its (nonexistent) succeeding layer.

4.3. Forward Propagation

Having defined a model for the architecture and the constituting parts of an ANN, it is now possible to model the network's operation, that is, to define how to determine its output vector. Forward propagation, the classical algorithm for doing precisely that, consists of sequentially computing the activations of each layer, from the input to the output layer.

Let the input (column) vector of the ANN – that is, the data being fed to it at a given instant – be $netInput$. Then, the vector of input node activations x is the concatenation of the activation of the input bias node, here defined as a constant 1 (the number one, not the lower case letter L), and the activations of the externally-stimulated data nodes – that is, $netInput$. Similarly, the vector h is the concatenation of the hidden bias node and the activations of the hidden nodes connected to the input layer; as discussed earlier, each node's activation is a function of the weighted sum of its inputs. Finally, because there are no output bias nodes, the y vector is simply obtained by computing the activations of the output nodes.

It should be noted that the Φ function is to be applied in an element-wise fashion, and it is not necessarily the same function for every neuron (even in the same layer) – the Φ symbol is used repeatedly only to simplify the notation.

$$x = \begin{bmatrix} 1 \\ netInput \end{bmatrix} \qquad h = \begin{bmatrix} 1 \\ \Phi(v \cdot x) \end{bmatrix} \qquad y = \Phi(w \cdot h)$$

(4.3) – Forward Propagation algorithm

4.3.1. Example

Let us consider the ANN illustrated in Figure 4.2. The activation function of the hidden nodes is the sigmoid function (4.2), referred to as $\mathrm{sig}(\cdot)$, and the activation function of the output nodes is the purely linear function (4.1), referred to as $\mathrm{purelin}(\cdot)$.

Let $v = 0.01 \times [10\ 11\ 12\ 13;\ \ 20\ 21\ 22\ 23;\ \ 30\ 31\ 32\ 33;\ \ 40\ 41\ 42\ 43;\ \ 50\ 51\ 52\ 53]$.
Let $w = 0.01 \times [10\ 11\ 12\ 13\ 14\ 15;\ \ 20\ 21\ 22\ 23\ 24\ 25]$.

Let $netInput = [1\ 2\ 3]^{\mathrm{T}}$.
Then, $x = [1;\ \ netInput] = [1\ 1\ 2\ 3]^{\mathrm{T}}$.
Then, $h = [1;\ \ \mathrm{sig}(v \cdot x)] = [1.0000\ \ 0.6985\ \ 0.8235\ \ 0.9038\ \ 0.9498\ \ 0.9744]^{\mathrm{T}}$.
Then, $y = \mathrm{purelin}(w \cdot h) = [0.6723\ \ 1.2073]^{\mathrm{T}}$.

4.4. Backpropagation

The Backward Propagation of Errors, or backpropagation, is the most common method of training artificial neural networks, used typically in conjunction with optimization algorithms which aim to minimize the cumulative squared error between the ANN's actual output and its target output. Such algorithms include the Nelder-Mead method [42] and the Levenberg-Marquardt algorithm [43].

Backpropagation is typically called a supervised learning algorithm, in which the target output of the ANN is explicitly specified by the modeler. This, however, is not a precise way of describing backpropagation. While it is true that it can be used (and is most often used) to perform supervised learning tasks when coupled with one of the optimization algorithms enumerated above, the true purpose of backpropagation is to solve the problem of *structural credit assignment*, that is, the problem of adjusting the weights in the network to minimize the error [44]. There is a subtle but important distinction between the two definitions – one which will be expanded upon further. Meanwhile, let us explore the formalism behind *backpropagation proper*, that is, the mechanics of weight adjustment. See [44] for this (and more) information.

Let there be an ANN whose nodes' activations have been obtained through the forward propagation of a training input vector and whose output error E has been determined according to some specific metric. For the purpose of completeness, let this metric be the sum of the square of the errors between the target output vector t and the actual output vector y of the network:

$$E = \sum (t - y)^2 \tag{4.4}$$

The global weight update rule is displayed in (4.5). This rule asserts that the change $\Delta\theta_{ij}$ in every weight θ_{ij} of the network (the elements of the v and w matrices) should be proportional (with constant α) to the negative of the derivative of the error with respect to the weight itself:

$$\Delta\theta_{ij} = -\alpha \frac{\partial E}{\partial \theta_{ij}} \tag{4.5}$$

Using the chain rule, the partial derivative of the error with respect to each weight between the hidden and output layers can be calculated, resulting in (4.6), where net_k is the net input ("net" as in "weighted", not short for "network") of the output node k, that is, $w \cdot h$:

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial a_k} \cdot \frac{\partial a_k}{\partial net_k} \cdot \frac{\partial net_k}{\partial w_{kj}} \tag{4.6}$$

Simple substitutions lead to (4.7), where $\Phi'_k(net_k)$ is the derivative of the activation function of the output node k evaluated at net_k:

$$\frac{\partial E}{\partial w_{kj}} = -2(t_k - a_k) \cdot \Phi'_k(net_k) \cdot a_j \tag{4.7}$$

We can now use δ_k to represent $(t_k - a_k) \cdot \Phi'_k(net_k)$, thus leading to (4.8):

$$-\frac{\partial E}{\partial w_{kj}} \propto \delta_k \, a_j \tag{4.8}$$

Using the chain rule, the partial derivative of the error with respect to each weight between the input and hidden layers can be calculated, resulting in (4.9), where net_j is the net input of the output node *j*, that is, $v \cdot x$:

$$\frac{\partial E}{\partial v_{ji}} = \frac{\partial E}{\partial a_k} \cdot \frac{\partial a_k}{\partial net_k} \cdot \frac{\partial net_k}{\partial a_j} \cdot \frac{\partial a_j}{\partial net_j} \cdot \frac{\partial net_j}{\partial v_{ji}} \tag{4.9}$$

Simple substitutions lead to (4.10), where $\Phi'_j(net_j)$ is the derivative of the activation function of the hidden node *j* evaluated at net_j:

$$\frac{\partial E}{\partial v_{ji}} = \delta_k \cdot w_{kj} \cdot \Phi'_j(net_j) \cdot a_i \tag{4.10}$$

Contrary to the weights between the hidden and output layers, the weights between the input and hidden layers affect all of the output nodes simultaneously. Thus, the partial derivative of the error *across all of the output nodes* is defined in (4.11)

$$\delta_j = \Phi'_j(net_j) \sum_k \delta_k \cdot w_{kj} \tag{4.11}$$

Finally, the partial derivative of the error with respect to the weights between the input and hidden layers can be defined as in (4.12):

$$-\frac{\partial E}{\partial v_{ji}} = \delta_j \, a_i \tag{4.12}$$

5. TEMPORAL DIFFERENCE LEARNING

Temporal Difference (TD) is a reinforcement learning method, that is, a way of using past experience with an incompletely known system to predict its future behavior [45]. In a more mechanistic sense, TD is an algorithm for an agent (like a predistorter) to learn which actions to take over an environment (like a power amplifier) in order to maximize some notion of cumulative reward (like a measure of an amplifier's linearity).

TD is an unsupervised learning algorithm, which means that it does *not* require the *a priori* knowledge of the desired output of the learning agent. This is an exceptionally important detail: using a supervised learning algorithm to teach an ANN how to predistort a power amplifier does not make much sense if one does not know the amplifier's inverse transfer function to begin with.

This does not mean that it is impossible to do so, as there are a variety of papers on neural predistortion of power amplifiers [8–10]. These papers, however, either don't explicitly specify the learning procedure (only mentioning backpropagation, which, as is hopefully clear by now, is not a serious answer), or describe a learning procedure consisting of iteratively training an ANN to be a post-distorter, testing its performance as a predistorter, and training it again in order to gain some measure of improvement.

While this sort of methodologies may lead to acceptable results, TD provides a learning solution that is more formal, and it has been used in applications as diverse as solving the game of Backgammon [39], controlling quadcopter motors and inverted pendulums [41], simulating the steering of a boat across a river [46], and sensor state prediction [47].

It should be noted that TD is a general learning algorithm, that is, it does not make any assumptions regarding the learning agent. TD is not, therefore, immediately applicable to the training of structurally complex constructs such as ANNs, and that means that some sort of mathematical coupling needs to be devised. Luckily, this problem has already been solved, and it is explained further.

5.1. Mathematical Formalization

5.1.1. TD Error

Let V be the value function an agent is trying to learn. TD learning consists in adjusting V so that $V(s_t)$ – where s_t is the input state at time t – approximates the return R_t at time t, defined in (5.1) as a discounted sum of future rewards. γ is the discount constant, and it controls how far the agent should look ahead when making predictions at the current time step [44]. Equation (5.2) is derived trivially from (5.1).

$$R_t \;=\; r_{t+1} \;+\; \gamma\, r_{t+2} \;+\; \gamma^2\, r_{t+3} \;+\; \cdots \;=\; \sum_{k=0}^{\infty} \gamma^k\, r_{t+k+1} \qquad (5.1)$$

$$R_t \;=\; r_{t+1} \;+\; \gamma\, R_{t+1} \qquad (5.2)$$

Thus, the TD error E_t at time t can defined as in (5.3):

$$E_t \;=\; R_t - V(s_t) \;=\; (r_{t+1} + \gamma\, R_{t+1}) - \; V(s_t) \qquad (5.3)$$

Finally, using $V(s_{t+1})$ as an approximation of R_{t+1}, we obtain the generalized TD error in (5.4):

$$E_t \;=\; r_{t+1} + \gamma\, V(s_{t+1}) - \; V(s_t) \qquad (5.4)$$

5.1.2. Weight Update

The derivation of the weight update rule (5.5) is rather involved, and can be found in [44].

$$\Delta w_t \;=\; \alpha[V(s_{t+1}) - \; V(s_t)] \sum_{k=1}^{t} \lambda^{t-k}\, \nabla_w V(s_k) \qquad (5.5)$$

This is the generalized formula for TD(λ), which is the generalized form of TD itself, introduced in [25]. α is a learning-rate parameter, $V(s_{t+1}) - V(s_t)$ is the (temporal) difference between consecutive predictions, $\nabla_w V$ is the gradient of the value function with respect to its defining weights, and λ is a gradient discount parameter such that $0 \leq \lambda \leq 1$. λ tracks to which extent the prediction values for previous observations are eligible for updating based on current errors [44]. Therefore, the sum (5.6) is called the *eligibility trace* at time t.

$$e_t = \sum_{k=1}^{t} \lambda^{t-k} \nabla_w V(s_k) \tag{5.6}$$

5.2. TD(λ) Neural Networks

As discussed earlier, backpropagation solves the problem of *structural credit assignment*. On the other hand, TD solves the problem of *temporal credit assignment*, that is, the problem of attributing credit (or "blame") for error over the complete history of predictions made by the learning agent [44], and it does so through the mechanism we've just introduced: eligibility traces.

Through TD(λ) learning, an agent can determine its error based on successive predictions, and through backpropagation an agent can modify its model of prediction in order to reduce the error. Thus, combining the two algorithms results in a very powerful coupling: a universal nonlinear function approximator which learns through acquired experience.

Contrary to other neural predistortion schemes found in the literature, the one proposed in this section – a TD(λ) Neural Network (TDNN) – is actually capable of learning how to be a predistorter. Since the learning algorithm does not require the knowledge of the target output of the ANN, the problem of predistortion may be tackled directly, and not indirectly by training the network as a post-distorter and hoping it works as a predistorter.

5.2.1. Mathematical Formalization

5.2.1.1. Weight Update

The coupling of TD learning and backpropagation is done at the weight update stage of the algorithms. Thus, and referring back to section 4, the change in the network's weights v and w is a function of the TD error E (at each output node k) and their respective eligibility traces ev and ew:

$$\Delta w_{kj} = E_k ew_{kj} \tag{5.7}$$

$$\Delta v_{ji} = \sum_k E_k ev_{ji}^{(k)} \tag{5.8}$$

From (5.7) it is very apparent that ew should be a matrix with the same size as w: $\big(nY \times (nH + 1)\big)$. From (5.8) it is apparent that ev should be, however, a three-dimensional matrix of size $(nH \times (nX + 1) \times nY)$ – or, rather, a set of nY matrices of size $\big(nH \times (nX + 1)\big)$, which is the size of w. The superscript (k) notation refers to each of the nY matrices.

5.2.1.2. Eligibility Traces

In section 4, a mathematical formalization – a model – of a generic artificial neural network was proposed. In this section, this model is expanded to include the eligibility traces introduced by the TD learning method, effectively resulting in a model of a TDNN. The basis of this work can be found in [44] and [48].

Let ew_{kj} denote the eligibility trace correspondent to the weight of the connection from the hidden node j to the output node k. Let δy_k denote $\Phi_k'(net_k)$. Then, the update rule for ew_{kj} is (5.9):

$$\mathrm{ew_{kj}} := \lambda \mathrm{ew_{kj}} + \Delta \mathrm{ew_{kj}},$$
$$\text{where } \Delta \mathrm{ew_{kj}} = \delta \mathrm{y_k} \mathrm{h_j} \tag{5.9}$$

The matrix form of (5.9) is self-evident, but the scheme in Figure 5.1 illustrates a simple way of deducing it:

$$h^{\mathrm{T}} = \begin{bmatrix} h_0 & h_1 & h_2 & h_3 & h_4 & h_5 \end{bmatrix}$$

$$\delta y = \begin{bmatrix} \delta y_1 \\ \delta y_2 \end{bmatrix} \underbrace{\begin{bmatrix} \delta y_1 h_0 & \delta y_1 h_1 & \delta y_1 h_2 & \delta y_1 h_3 & \delta y_1 h_4 & \delta y_1 h_5 \\ \delta y_2 h_0 & \delta y_2 h_1 & \delta y_2 h_2 & \delta y_2 h_3 & \delta y_2 h_4 & \delta y_2 h_5 \end{bmatrix}}_{\Delta ew}$$

Figure 5.1. Deduction of the matrix form of Δew.

Thus we get the update rule for the matrix form of ew:

$$\mathrm{ew} := \lambda \mathrm{ew} + \Delta \mathrm{ew},$$
$$\text{where } \Delta \mathrm{ew} = \delta \mathrm{y} \cdot \mathrm{h}^{\mathrm{T}} \tag{5.10}$$

The activation function of the output nodes of the TDNN is purely linear, so $\delta y_k = 1$ for all k.

Let $ev_{ji}^{(k)}$ denote the derivative of the output unit k with respect to the weight from the input unit i to the hidden unit j, that is, a partial eligibility trace correspondent to the weight of the connection from the input node i to the hidden node j.

Let $\overline{w}$ be the w matrix without its first column. Let $\overline{\delta h}$ be the δh vector without its first row. This removes the elements of these objects correspondent to h_0, the hidden bias node. This is necessary because there are no connections from the input nodes to the hidden bias node, which means that there are no corresponding weights or eligibility traces.

Then, the update rule for $ev_{ji}^{(k)}$ is (5.11):

$$ev_{ji}^{(k)} := \lambda ev_{ji}^{(k)} + \Delta ev_{ji}^{(k)},$$
$$where\ \Delta ev_{ji}^{(k)} = \delta y_k\ \overline{w}_{kj}\ \overline{h_j}\ x_i \tag{5.11}$$

Let us explore the Δ term of ev based on Figure 4.2:

$$\Delta ev_{10}^{(1)} = \delta y_1\ w_{11}\ \delta h_1\ x_0$$
$$\Delta ev_{11}^{(1)} = \delta y_1\ w_{11}\ \delta h_1\ x_1$$
$$\Delta ev_{12}^{(1)} = \delta y_1\ w_{11}\ \delta h_1\ x_2$$
$$\Delta ev_{13}^{(1)} = \delta y_1\ w_{11}\ \delta h_1\ x_3$$

$$\Delta ev_{20}^{(1)} = \delta y_1\ w_{12}\ \delta h_2\ x_0$$
$$\Delta ev_{21}^{(1)} = \delta y_1\ w_{12}\ \delta h_2\ x_1$$
$$\Delta ev_{22}^{(1)} = \delta y_1\ w_{12}\ \delta h_2\ x_2$$
$$\Delta ev_{23}^{(1)} = \delta y_1\ w_{12}\ \delta h_2\ x_3$$
$$\vdots$$
$$\Delta ev_{10}^{(2)} = \delta y_2\ w_{21}\ \delta h_1\ x_0$$

Let (5.12), where $\cdot$ is the matrix multiplication operator and $.*$ is the element-wise multiplication operator:

$$\xi = \delta y \cdot \overline{h}\ .*\ \overline{w} \tag{5.12}$$

Thus,

$$\underset{(nY \times nH)}{\xi} = \begin{bmatrix} \delta y_1 w_{11} \delta h_1 & \delta y_1 w_{12} \delta h_2 & \cdots & \delta y_1 w_{1nH} \delta h_{nH} \\ \delta y_2 w_{21} \delta h_1 & \delta y_2 w_{22} \delta h_2 & \cdots & \delta y_2 w_{2nH} \delta h_{nH} \\ \vdots & \vdots & \ddots & \vdots \\ \delta y_{nY} w_{nY1} \delta h_1 & \delta y_{nY} w_{nY2} \delta h_2 & \cdots & \delta y_{nY} w_{nYnH} \delta h_{nH} \end{bmatrix} \tag{5.13}$$

Substituting (5.12) in (5.11) we get (5.14):

$$\Delta ev_{ji}^{(k)} = \xi_{kj}\, x_i \tag{5.14}$$

Let $\xi^{(k)}$ denote the *k*-th row of the ξ matrix. Then, finally, we get the update rule for the matrix form of each $ev^{(k)}$:

$$ev^{(k)} := \lambda ev^{(k)} + \left(x \cdot \xi^{(k)}\right)^T \tag{5.15}$$

As a final note, the approximate derivatives of the activation functions used throughout the ANN are defined in (5.16) for the sigmoid function and in (5.17) for the purely linear function.

$$\Phi_j'\left(net_j\right) = 1 \tag{5.16}$$

$$\Phi_k'(net_k) = a_k(1 - a_k) \tag{5.17}$$

5.2.2. TDNN Algorithm

The model for an artificial neural network using temporal differences as a learning method has been established. Now, let us explain how it can be used. Appendix A contains a class-based Matlab implementation of the vectorized TDNN model and the learning algorithm based on Sutton's (the creator of TD(λ)) own TD/Backpropagation pseudo-code [48], also used as a reference for the expansion of the model.

In a slightly simplified way, the TDNN algorithm consists of repeatedly iterating over the following set of steps:

1. Perform the forward propagation of an input vector;
2. Calculate the TD error at the output of the network;
3. Update the network's weights;
4. Perform the forward propagation of the same input vector with the new weights;
5. Update the eligibility traces of the network.

Forward propagation is explained in section 4.3. The TD error is defined in (5.4); note that training in the first iteration must be skipped so that the error equation becomes causal. The changes applied to the weight matrices in order to update them are defined in (5.6) and (5.7). Finally, the update rules for the eligibility trace matrices are defined in (5.9) and (5.14).

5.3. Simulation Results

Despite our best efforts, TDNN ended up not producing any positive results. Interfacing with the algorithm requires two signals: the input of the ANN and a reward signal in which the performance of the ANN is encoded. There are endless ways of defining the reward signal, so it is not possible to say for sure that the TDNN algorithm does not work – we can only say that it did not work with the reward definitions that were tested. With that said, our tests were fairly exhaustive – see Appendix B.

Let PA_{out} be the actual output and PA_{out}^{T} the target output of the PA for a given input vector. Let $E = PA_{out} - PA_{out}^{T}$ be the error of each sample of the PA_{out} vector, let $SE(k) = E(k)^2$ be the squared error of each sample of the PA_{out} vector, and let $MSE = \text{mean}(SE)$ be the mean squared error of the same vector. By definition, both E and SE are vectors with the same dimension as PA_{out} and MSE is a scalar. Finally, let $reward$ be the reward vector.

The first tests of the TDNN algorithm used the definitions of reward in (5.17): a null reward for every input state except the last one, which was rewarded with the negative of the MSE calculated in the previous iteration. We chose the negative of the MSE because MSE is an error, and therefore it is a *penalization* rather than a *reward*. The idea behind this encoding is the rewarding based on the compound performance of the predistorting ANN over the complete input vector.

The result was a very quick divergence of the network weights for many combinations of the γ, λ, α, and β parameters of the TDNN – the reward discount rate, the trace decay rate, and the learning rates of the two weights matrices v and w.

$$reward \quad = \quad [0 \quad 0 \quad 0 \quad \cdots \quad 0 \quad -MSE] \tag{5.17}$$

In the second series of tests, the reward signal was defined as in (5.18), that is, similarly to what was done in the previous tests, but with a reward for every input state instead of only the last state. Unsurprisingly, this led to the divergence of the network weights.

$$reward \quad = \quad -MSE \quad \times \quad [1 \quad 1 \quad 1 \quad \cdots \quad 1] \tag{5.18}$$

The next batch of tests – (5.19) and (5.20) – departed from the previous ones in the sense that the reward values were not compound, but specific of each input state. Unfortunately, the results remained not ideal: depending on the configuration parameters, the output of the ANN either diverged like in the previous cases or oscillated wildly.

$$reward \quad = \quad -SE \tag{5.19}$$

$$reward \quad = \quad \pm E \tag{5.20}$$

Finally, the tests fully degenerated into defining the reward signal as equal to the target output of the PA (5.21). While this might seem like it does not make much sense, as it is not a measure of the network's performance and it defines the reward as a constant vector, it provided some insight into the TDNN algorithm and confirmed that it was not fully malfunctioning.

$$reward \quad = \quad PA_{out}^{T} \tag{5.21}$$

This test revealed that the TDNN algorithm mimics the Backpropagation algorithm in the sense that it adjusts the weights of the ANN so that the output of the ANN is equal to the reward signal. This only happens for $\gamma = 0$ and, to be fair, it is painfully slow – though it can be accelerated by setting λ to a relatively low value, like 0.3.

While this proves that the implementation of TDNN is not completely bug-ridden, as one might have assumed based only on the diverging tests, it is still not a viable solution for the training of a predistorting ANN.

In hindsight, it does make sense that the TDNN algorithm was not able to train an ANN as a predistorting system. Temporal Difference learning is commonly described as a method for policy evaluation, or prediction, which means that, for a given policy, TD can be used to iteratively learn the value, or utility, of a given input state.

This does not intuitively translate very well into the predistortion problem, though we could say that the policy of the PD problem is the transfer function of the ANN, parameterized by its weights. Now, the whole point of the PD exercise is to *change the weights* of the ANN in order to achieve a goal, and changing the weights of the ANN means changing the policy, which is not what TD learning is about. This might very well be the underlying reason for the TDNN strategy having failed.

In spite of the lack of success found using TD learning, this was still an important step in finding a better solution. Many meetings and discussions were held with various professors and colleagues in doctoral programs, and those resulted – among others – in the pursuit of a solution based on evolution strategies, expanded upon in the next section.

6. EVOLUTION STRATEGIES

The main idea behind the problem of optimization is the iterative improvement of a measure of the performance or value of a decision [49] – a decision which may be the selection of a set of weights for a predistorting ANN. This measure is provided by a cost function $f : \mathbf{R}^n \rightarrow \mathbf{R}$ which summarizes, in a single scalar, the fitness of an individual with n defining features.

While most common methods of optimization – such as gradient descent and Newton's method – may converge to local, non-optimal solutions due to their reliance on the gradient or higher-order statistics of the cost function, evolution strategies are guaranteed to find the globally optimal solution due to their stochastic nature, which follows the principles of natural evolution: mutation, recombination and selection in populations of candidate solutions [50].

6.1. CMA-ES

The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) is a state-of-the-art evolutionary algorithm for optimization in continuous domains [50]. Rather than calculating a metric of the cost function (such as the gradient), as most classical optimization algorithms do, and choosing the solution that improves it (or, more specifically, minimizes it) in a local search space, CMA-ES uses a (multi-variate) normal distribution to sample a set (a population) of new search points [51].

Any normal distribution, $\mathcal{N}(m, C)$, can be defined by its mean, $m \in \mathbf{R}^n$, and its covariance matrix, $C \in \mathbf{R}^{n \times n}$, for n equal to the dimension of the solutions [51]. Covariance matrices can be geometrically interpreted as hyper-ellipsoids, surfaces (in n-dimensional space) of equal density of the distribution, whose principal axes and their squared lengths correspond, respectively, to the eigenvectors and the eigenvalues of C [51].

The objective of CMA-ES is to fit the search distribution to the contour lines of the cost function – the lines of equal cost. Figure 6.1 illustrates three different normal search distributions in thick lines and the contour lines of an example cost function. Clearly, the distribution on the right side of the figure is the one that follows the contour of the cost function in the way that will most likely lead to an optimal solution [51].

As the name of the algorithm implies, the fitting of the search distribution is done by adapting its defining covariance matrix. Exactly how this is done, as well as the more specialized options of the algorithm, is outside of the scope of this document – to put things in perspective, the implementation used in the simulations detailed below has more than 3000 lines of code.

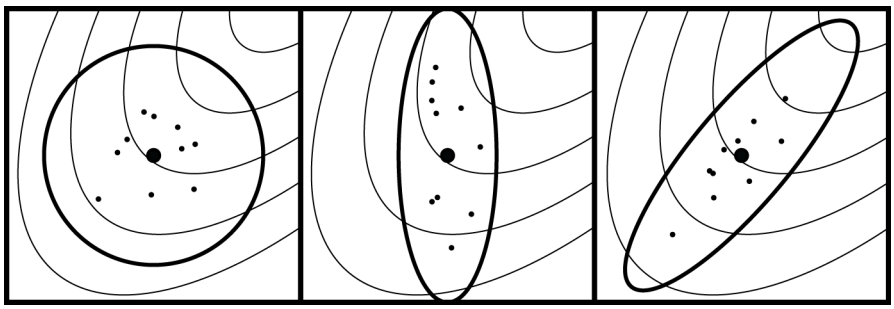


Figure 6.1. Three different normal search distributions [51].

6.2. Simulation Results

A free (GNU GPLv3) Matlab implementation [52] of the CMA-ES algorithm was used to minimize a cost function by adapting the 272 weights and biases of an ANN with two input nodes, three hidden layers of ten nodes each, and two output nodes – see Appendix C.

The cost function was the Normalized Mean Square Error (NMSE) of the VSPA output, defined in (6.1), where I^T and Q^T are the target outputs, and I and Q are the actual outputs of the VSPA for a given input (provided by the ANN that is being adapted). All squaring operations are done in an element-wise fashion.

$$SE = [I - I^T]^2 + [Q - Q^T]^2$$

$$ME = [I - \text{mean}(I^T)]^2 + [Q - \text{mean}(Q^T)]^2 \quad (6.1)$$

$$NMSE = \frac{\sum SE}{\sum ME}$$

For every iteration of the CMA-ES algorithm there are twenty evaluations of the cost function (by default), and for each one of these there is one execution of the forward propagation function of the ANN and one evaluation of the VSPA. In order to speed up the processing of the algorithm, a custom implementation of an ANN was created and the noise generator of the VSPA model was disabled by masking the random() Matlab function.

The custom ANN implementation (Appendix D) performs forward propagation about 100 times faster than the implementation available in Matlab's Neural Network Toolbox – most likely due to the processing overhead the latter requires in order to provide the whole functionality of the toolbox (though, honestly, it is quite surprising how slow it is). The lack of noise generation by the VSPA model means that the NMSE level reached may be, potentially, boundlessly negative in dB.

Figures 6.2 to 6.5 show the state of the CMA-ES algorithm at one hundred iterations, one thousand iterations, ten thousand iterations, and three hundred thousand iterations. These figures plot four different signals: the natural output of the VSPA (that is, without any sort of predistortion) as black dots, the target output of the linearized VSPA as red dots, the output of the predistorting ANN, and the response of the VSPA to that input as blue crosses. All of these signals are based on a relatively sparse grid of *I/Q* symbols – the linearization targets – that cover the complete output range of the VSPA.

The initial state of the weights of the ANN is a random vector of low values. Thus, the output of the ANN, as well as the output of the VSPA, is a cloud of dots and crosses around the center of the *I/Q* plane. Throughout the initial iterations, these expand in a random-looking way until the whole plane is filled. Then, it becomes clear that the CMA-ES algorithm is

slowly bringing the blue crosses closer and closer towards the red dots. Finally, the blue crosses become coincident with the red dots (linearization achieved) and the green dots end up warped in a way that is contrary to the warping effect shown in Figure 3.7 (the input of the VSPA has successfully been predistorted).

Figure 6.6 is a plot of the cost function (or, more correctly, the cost of the ANN selected by the algorithm among twenty alternatives in each iteration) in respect to time. While the results are excellent, it must be stated that this method is not very fast at all. Still, it can only get better: with more research time, it might have been possible to accelerate the algorithm by finely tuning its configuration parameters.

In any case, a decrease in NMSE of 20 dB per decade of iterations is very acceptable: the execution time would surely have been lower if the algorithm had been run on a quad-Nvidia Titan X machine with 64 GB of memory instead of a generic personal laptop.

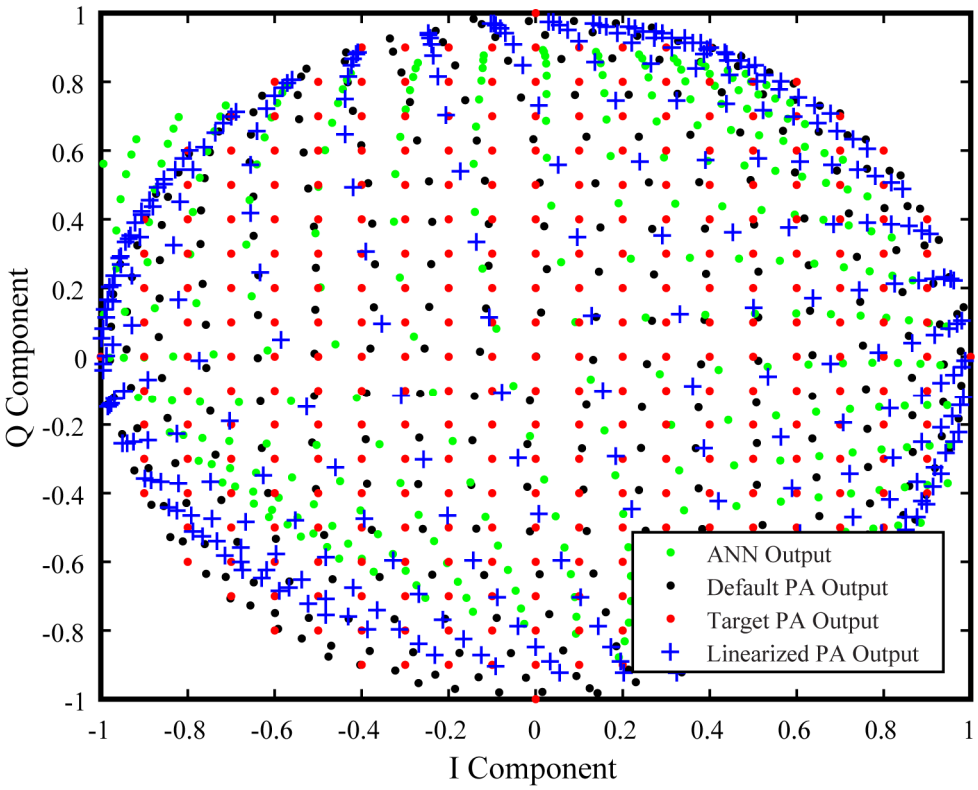


Figure 6.2. State of the CMA-ES algorithm: after 100 iterations.

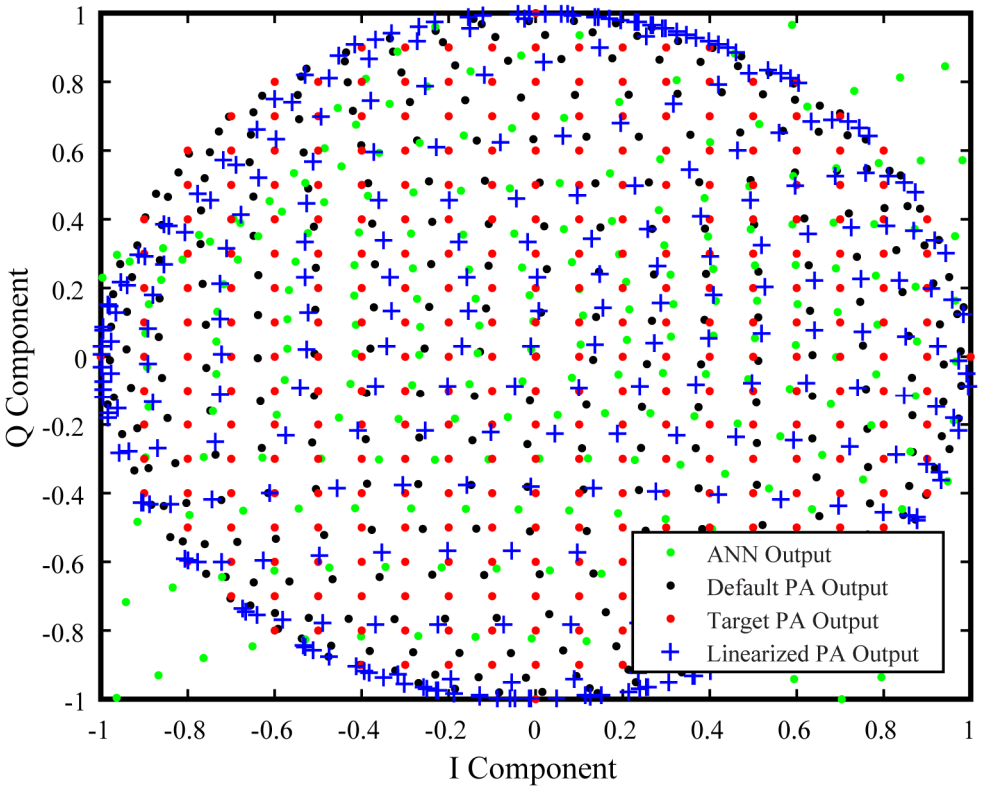


Figure 6.3. State of the CMA-ES algorithm: after 1000 iterations.

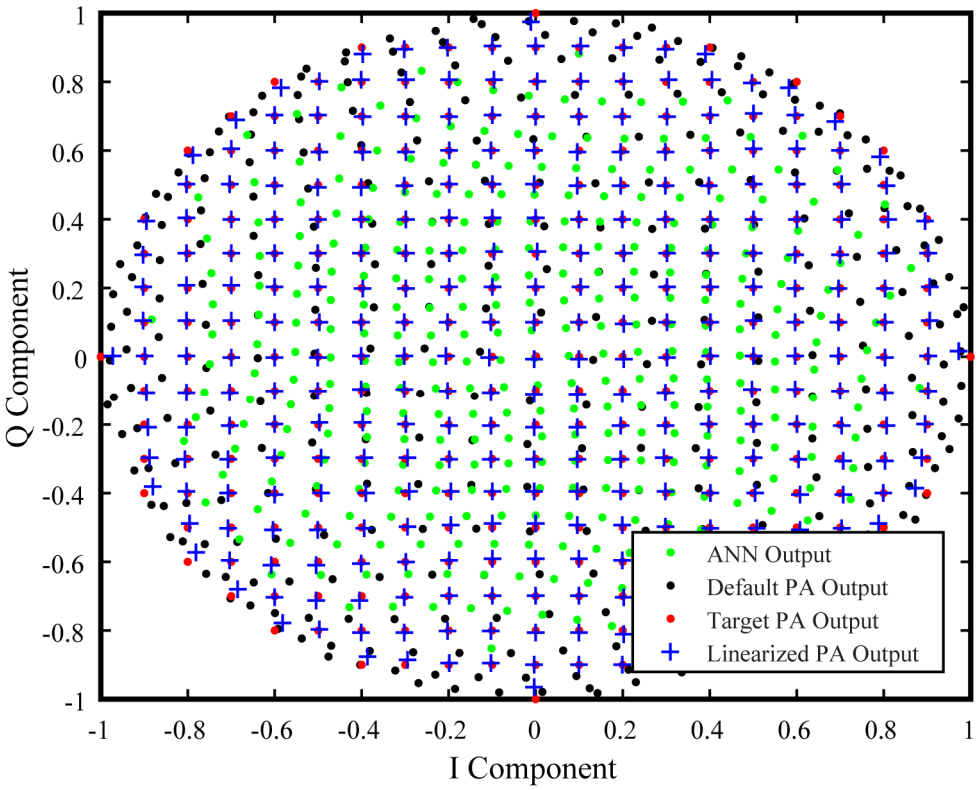


Figure 6.4. State of the CMA-ES algorithm: after 10,000 iterations.

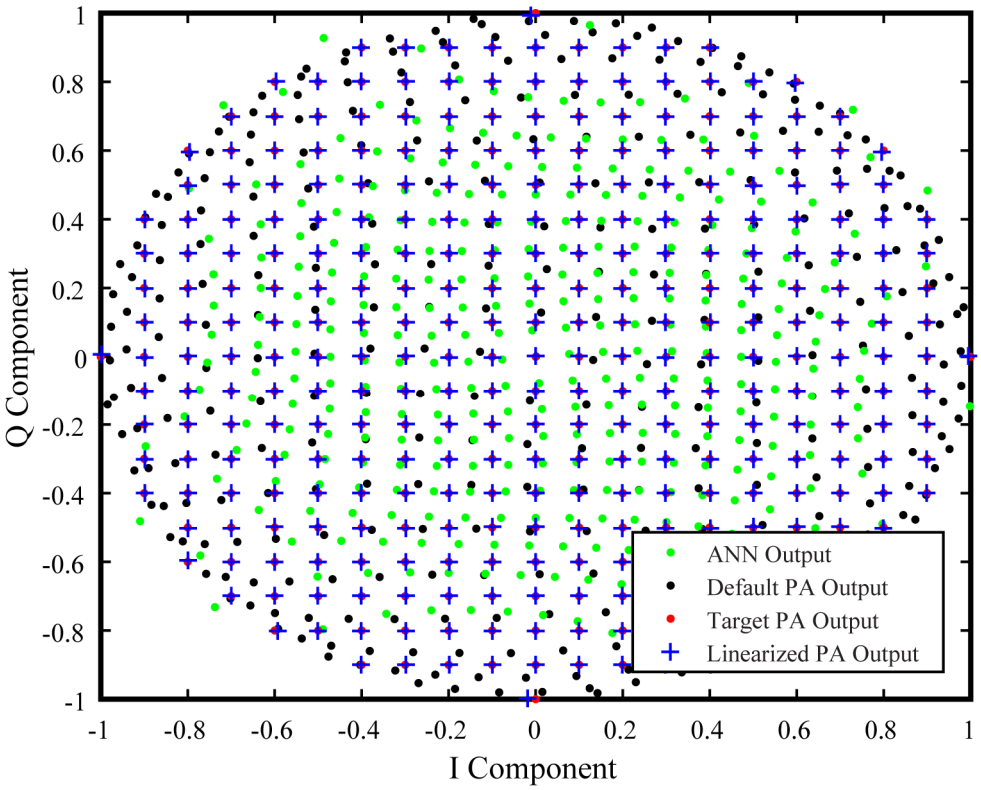


Figure 6.5. State of the CMA-ES algorithm: after 300,000 iterations.

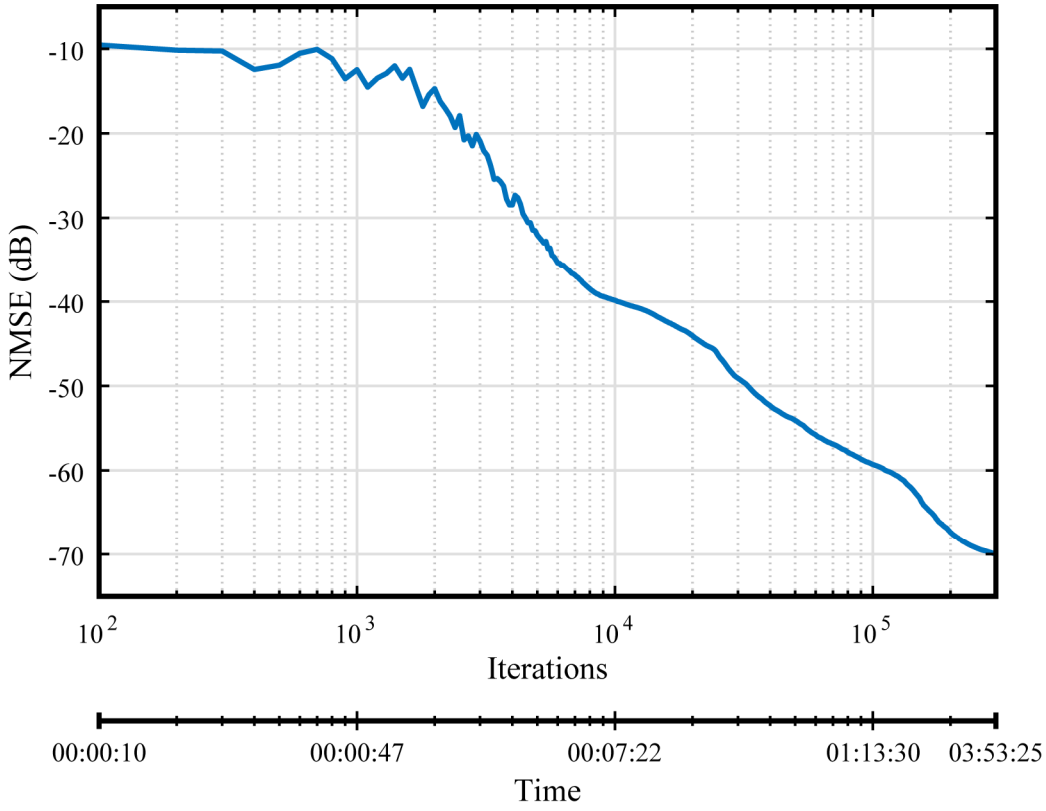


Figure 6.6. NMSE vs Time plot of the CMA-ES algorithm.

After the close-to-four hours had elapsed, the CMA-ES algorithm had produced an ANN that was capable of predistorting a grid of *I/Q* symbols with an NMSE at the output of the VSPA of –70 dB. Note, again, that this figure is only possible due to the fact that the noise generator of the VSPA model had been disabled, otherwise the NMSE would have converged to a higher value (close to –50 dB).

Figure 6.7 shows the AM-AM and AM-PM characteristics of the resulting ANN. Notice how they are opposite to those of the VSPA (Figure 3.8): at high input power levels, there is an increase in gain and a negative phase shift.

Figure 6.8 illustrates the AM-AM and AM-PM characteristics of the complete predistortion system: from the input of the ANN to the output of the VSPA. The gain is constant and there is no phase shift, so the system is linear. There is some dispersion in the plots due to numeric errors that occur at low power levels and due to the noise generator of the VSPA that was re-enabled.

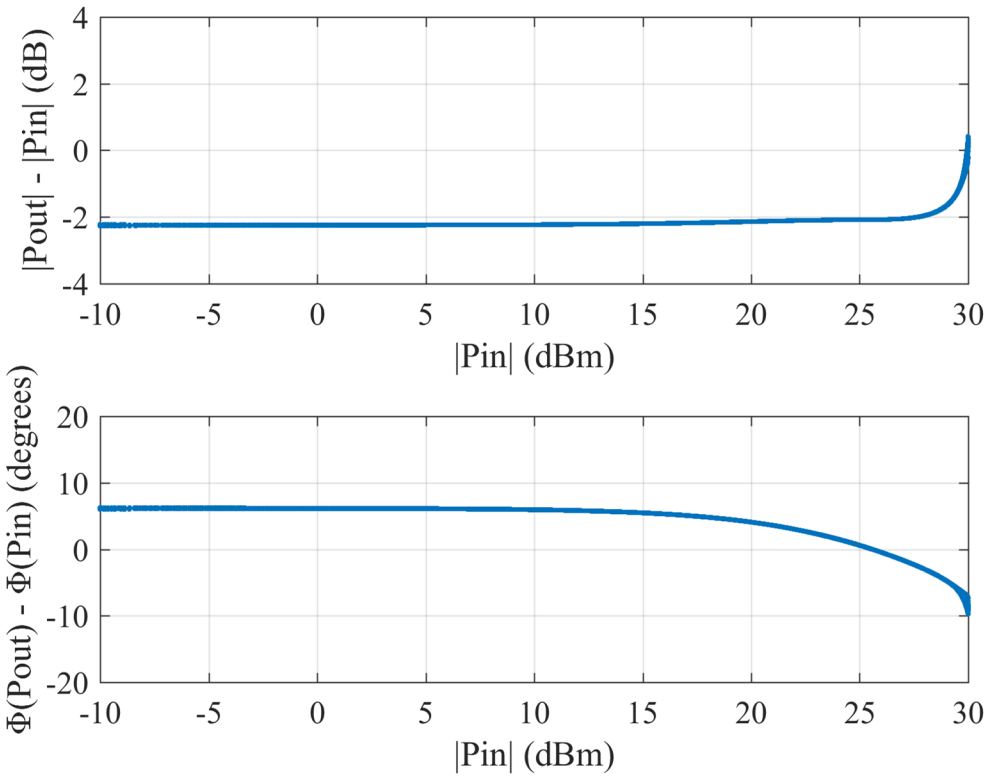


Figure 6.7. Gain and AM-PM characteristics of the ANN generated using CMA-ES.

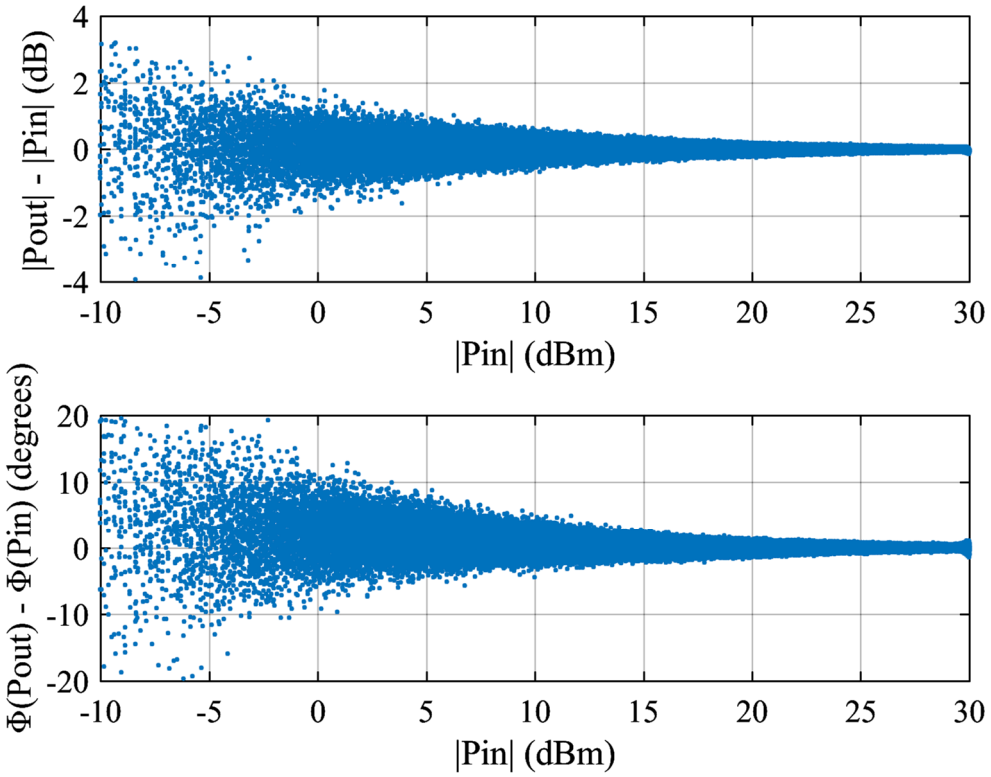


Figure 6.8. Gain and AM-PM characteristics of the VSPA linearized using CMA-ES.

The final verification of the developed predistortion system consisted in feeding it the four-carrier GSM signal described in section 3.2 and observing the frequency spectrum of the output of the VSPA. This signal is completely uncorrelated with the grid of input *I/Q* symbols used in the generation of the predistorting ANN.

This frequency spectrum is plotted in Figure 6.9, and it reveals that the linearization of the VSPA was nearly flawless: all of the intermodulation distortion tones were not just attenuated, but completely and utterly eliminated.

While it might have taken nearly four hours to generate an ANN with an NMSE of –70 dB, the linearization results show that it was worth it. With the possibility of adjusting the configuration parameters of the algorithm, and with a more capable computing platform, the CMA-ES algorithm shows great promise in generating an ANN for the linearization of a PA.

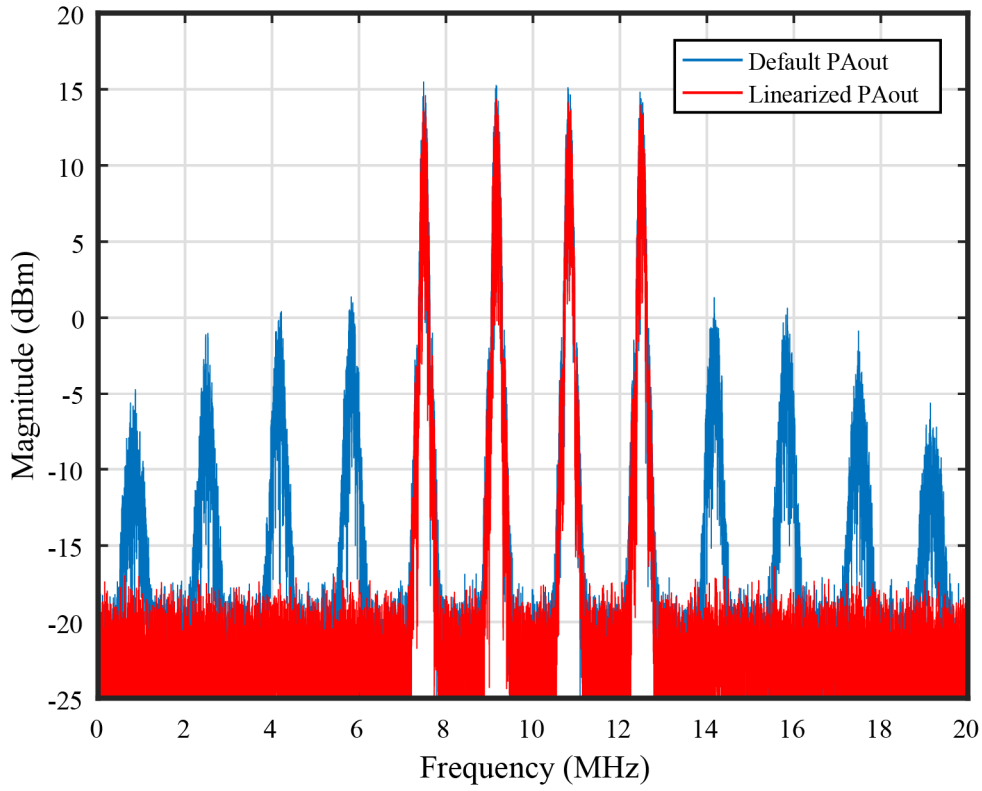


Figure 6.9. Output spectrum of the VSPA in response to the GSM signal with (red) and without (blue) predistortion by the ANN generated using CMA-ES.

7. SUCCESSIVE TARGET APPROXIMATION

Successive Target Approximation (STA) is an original, custom-made algorithm that can be used for the linearization of a PA [53]. STA computes the signal that, when fed to the PA as an input, results in a PA output signal that matches a specified target PA output signal. If the target PA output signal features the complete PA output vector space (or a sufficiently exhaustive sampling of it), then the algorithm effectively computes a mapping of the PA's output vector space to its input vector space.

In other words, STA computes the PA input signal correspondent to a given PA output signal – that is, the output of the PD. It immediately follows, then, that STA solves the problem of training a predistorting ANN using the Backpropagation algorithm: the lack of a target ANN output signal. Thus, the process of creating an ANN that predistorts a PA is simple:

1. Generate a vector of target linear PA output symbols;
2. Using the STA algorithm, compute the corresponding vector of predistorted PA input symbols – the target PD output;
3. Using the Backpropagation algorithm, train an ANN using a vector of linear input symbols as its input and the vector computed with the STA algorithm as its target.

7.1. The Algorithm

Let PA_{out}^{T} be the target output of the PA. The goal of the STA algorithm is to find the input vector of the PA, PA_{in}, that leads to the target output. Let PA_{out} be the output of the PA in response to a given input. Let $f_{PA}(\cdot)$ be the transfer function (or the model) of the PA.

While it would be possible to start STA with an initial approximation of PA_{in} as a vector of zeroes, it is intuitive that, whatever happens during the algorithm, the final outcome should not be too different from a linear input – that is, the input vector that would lead to PA_{out}^{T} if the PA were a linear device. Obviously, this linear input is equal to PA_{out}^{T} divided by the target gain of the PA, which is 0 dB, as stated in the section describing the VSPA.

Thus, let the initial approximation of PA_{in} be (7.1):

$$PA_{in} := PA_{out}^{T} \tag{7.1}$$

The algorithm proceeds as follows, with α being a learning rate parameter:

$$\begin{aligned} &\textbf{Repeat until convergence \{} \\ &\quad PA_{out} := f_{PA}(PA_{in}) \\ &\quad PA_{in} := PA_{in} + \alpha\,(PA_{out}^{T} - PA_{out}) \\ &\} \end{aligned} \tag{7.2}$$

That is it – STA is so simple, it is almost surprising it works. Once convergence is reached, PA_{in} can be used as the target for the training of an ANN using the Backpropagation algorithm, with its input being the linear input described just above (PA_{out}^{T}).

7.2. Simulation Results

The STA algorithm (Appendix E) was used to linearize the VSPA model. This is a base band model, so PA_{out}^{T}, PA_{out} and PA_{in} are vectors of *I/Q* symbolic pairs, that is, they are matrices of size $2 \times N$, with N being the number of symbols used. PA_{out}^{T} was defined as a relatively sparse grid of *I/Q* symbols that covers the whole output range of the VSPA.

Figures 7.1 to 7.4 show the state of the STA algorithm (with $\alpha = 0.5$) at zero iterations, one iteration, four iterations, and ten thousand iterations. These figures plot four different signals: the natural output of the VSPA (that is, without any sort of predistortion) as black dots, the target output of the linearized VSPA as red dots, the current state of the computed VSPA input as green dots, and the response of the VSPA to that input as blue crosses.

Throughout the various iterations, the blue crosses start coincident with the black dots (no linearization) and end up coincident with the red squares (complete linearization). Meanwhile, the green dots start coincident with the red dots (equation 7.1) and end up warped in a way that is contrary to the warping effect shown in Figure 3.7.

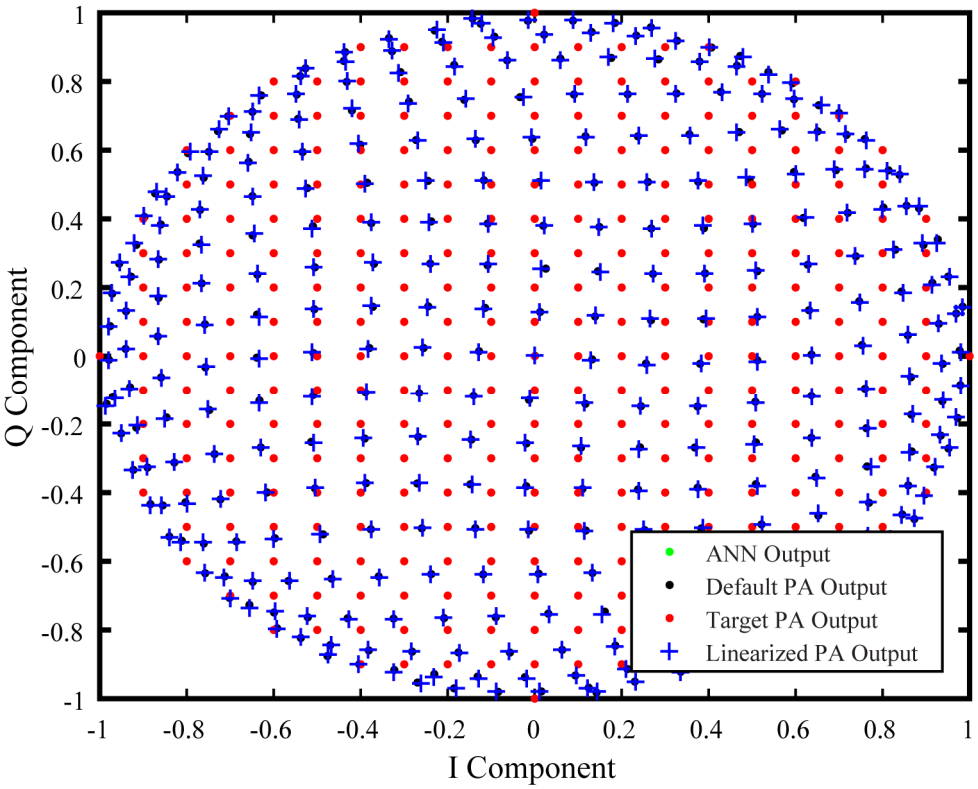


Figure 7.1. State of the STA algorithm: initial conditions.

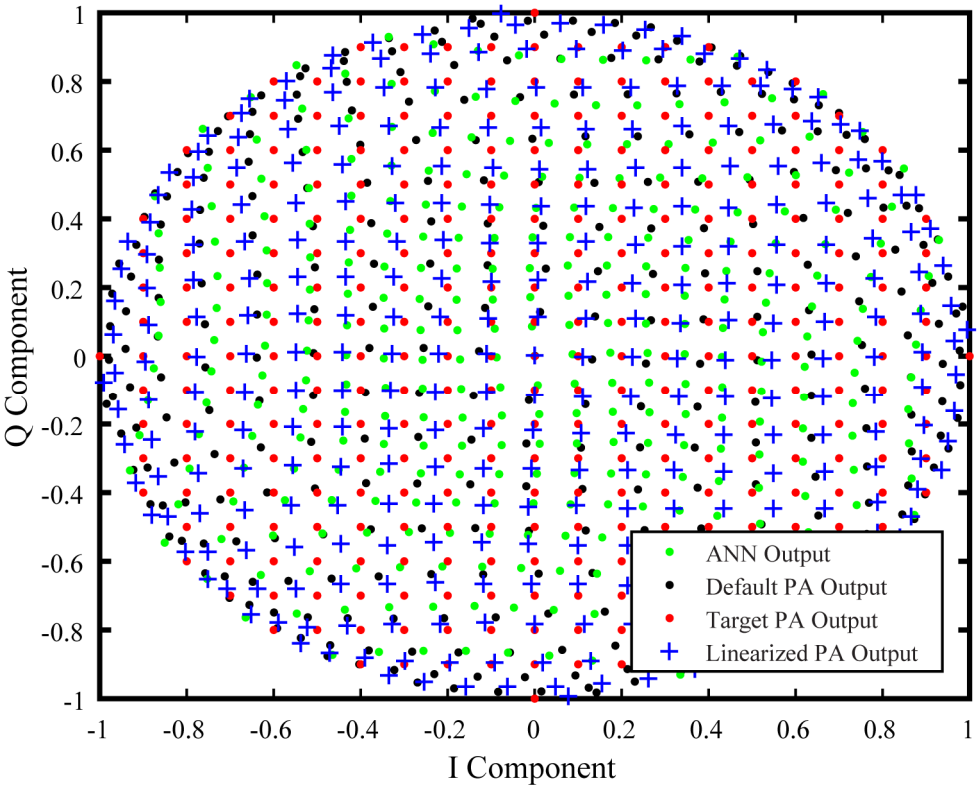


Figure 7.2. State of the STA algorithm: after the first iteration.

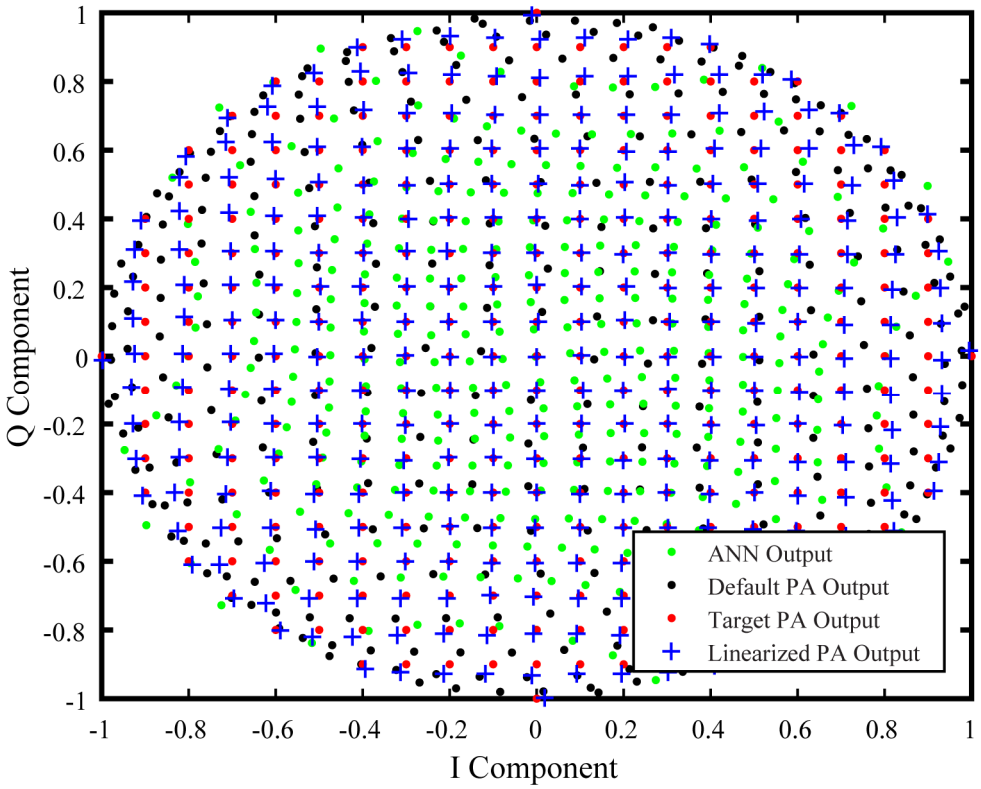


Figure 7.3. State of the STA algorithm: after the fourth iteration.

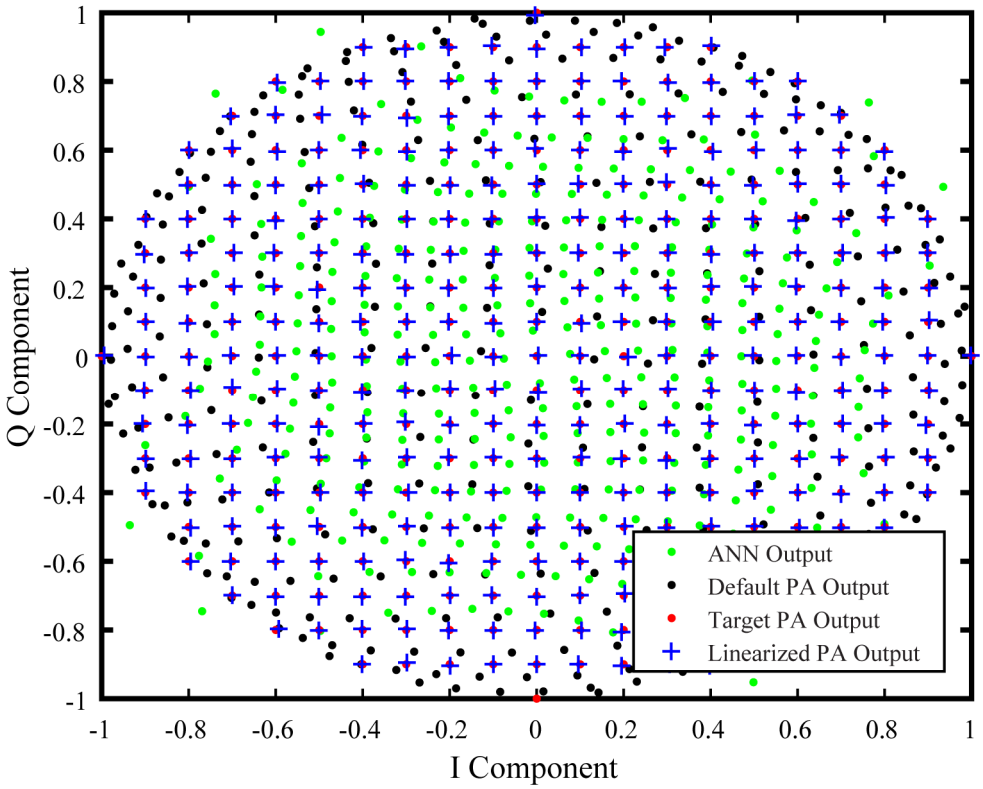


Figure 7.4. State of the STA algorithm: after the ten thousandth iteration.

It is quite the understatement to say that the results of the STA algorithm were unexpectedly good. Figure 7.5 is a plot of the NMSE at the VSPA output (that is, the error between the actual and the target outputs of the VSPA). This plot was generated with the noise generator of the VSPA model disabled – this is useful to get a more accurate measure of the actual performance of the algorithm itself, without the penalization introduced by the processing of the VSPA model.

As shown, STA achieves a staggeringly low NMSE in a matter of milliseconds.

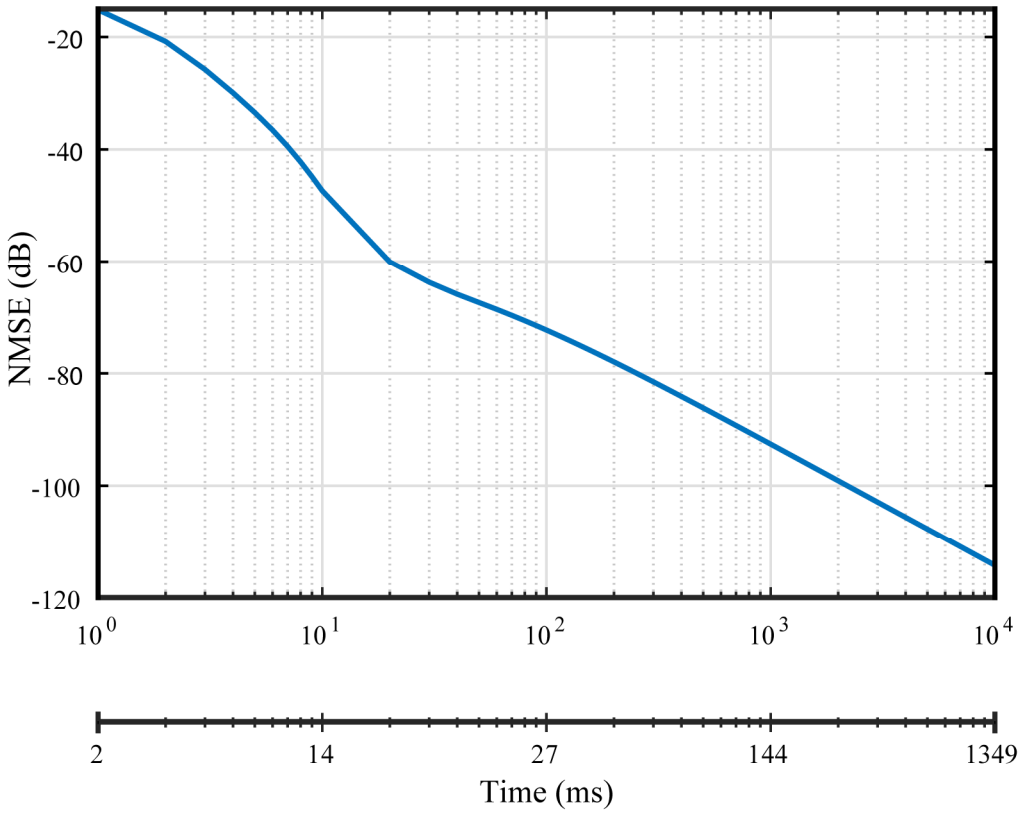


Figure 7.5. NMSE vs Time plot of the STA algorithm with the random() function disabled.

Naturally, enabling the noise generator of the VSPA model increases the processing time (by four times) and introduces a limit to how low the NMSE can be. Figure 7.6 shows exactly this.

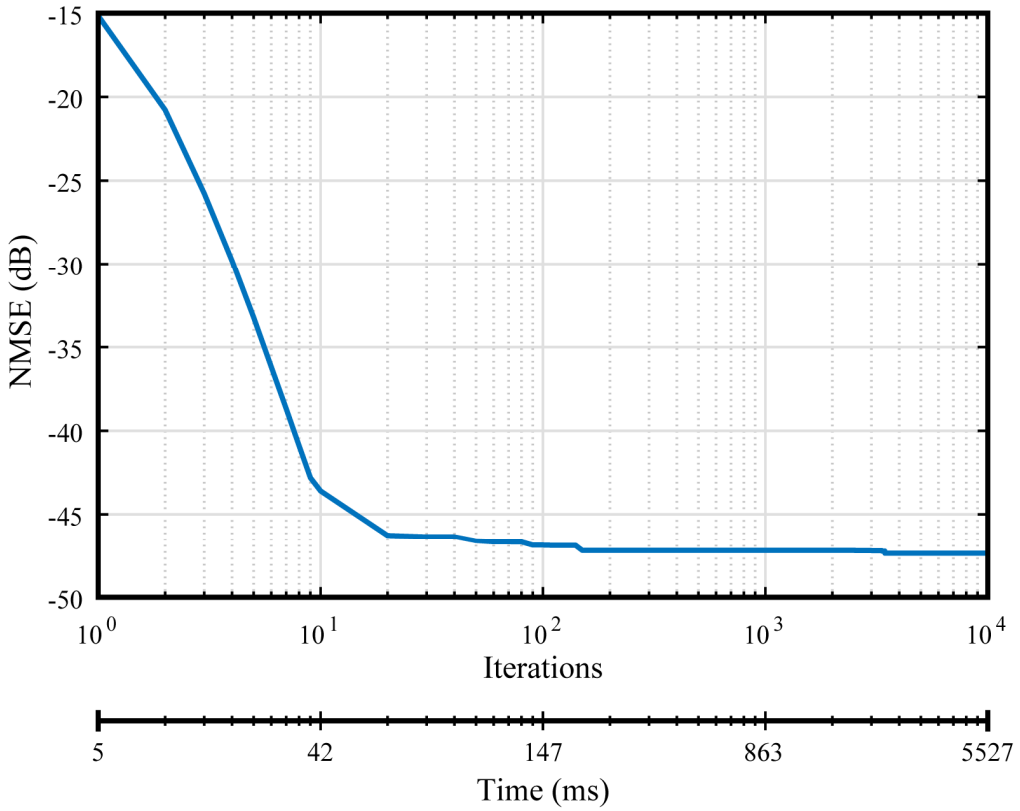


Figure 7.6. NMSE vs Time plot of the STA algorithm with the random() function enabled.

After the STA algorithm reached convergence (with the noise generator enabled), the Backpropagation algorithm was used to train the predistorting ANN. It must be noted that the Backpropagation algorithm minimizes (to a certain extent) the error at the output of the ANN, and this error is not equal to the error at the output of the VSPA. For this reason, the ANN training function (from Matlab's Neural Network Toolbox) must be run inside a loop in which the error at the output of the VSPA is monitored – otherwise there may be a significant drop in linearization performance.

Figure 7.7 illustrates the AM-AM and AM-PM characteristics of the generated ANN. Notice how they are opposite to those of the VSPA (Figure 3.8): at high input power levels, there is an increase in gain and a negative phase shift.

Figure 7.8 illustrates the AM-AM and AM-PM characteristics of the complete predistortion system: from the input of the ANN to the output of the VSPA. The gain is constant and there is no phase shift, so the system is linear. There is some dispersion in the plots due to numeric errors that occur at low power levels and due to the noise generator of the VSPA.

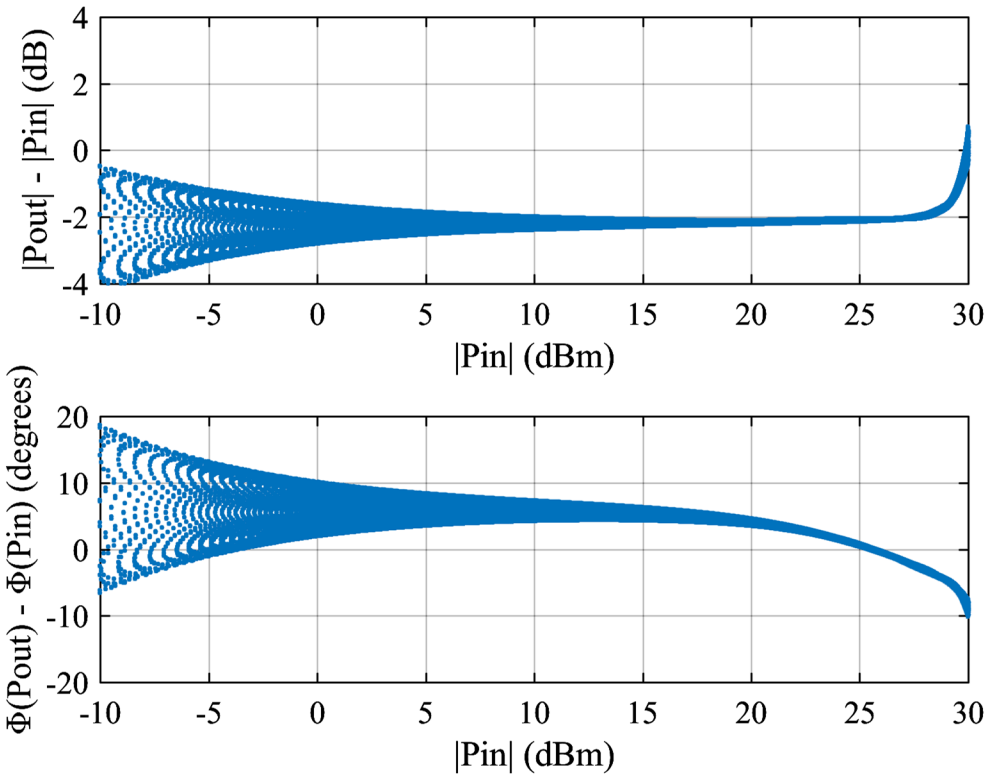


Figure 7.7. Gain and AM-PM characteristics of the ANN PD generated using STA.

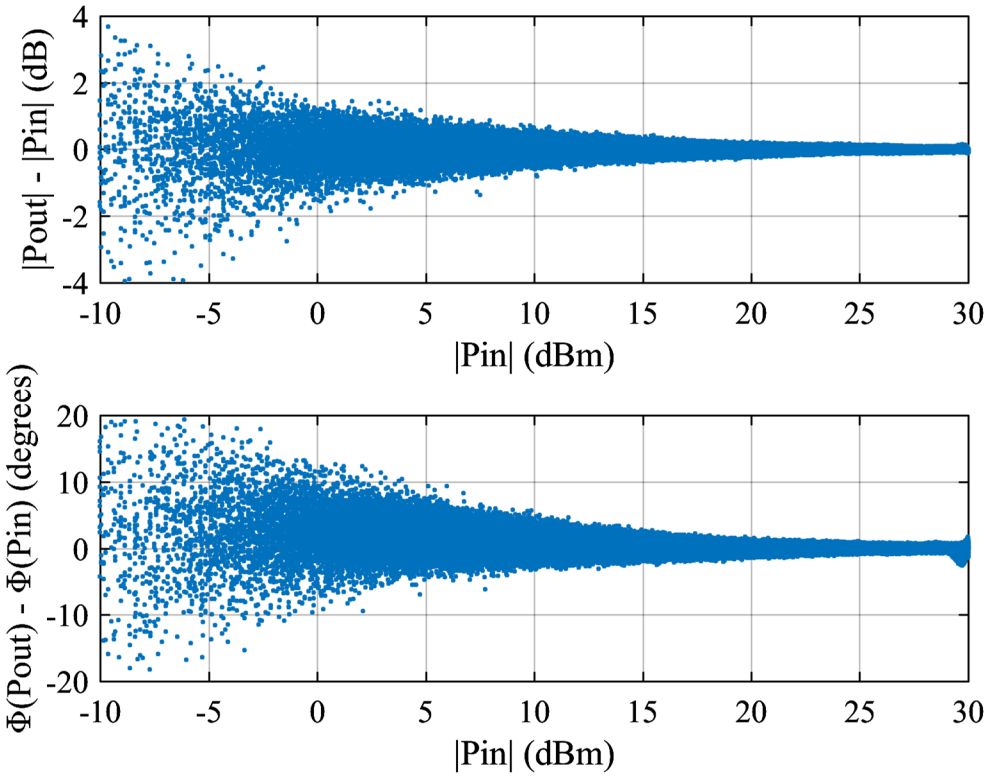


Figure 7.8. Gain and AM-PM characteristics of the VSPA linearized using STA.

In order to confirm that the predistortion system was in fact linear, the ANN generated using the STA algorithm (and the Backpropagation algorithm) was fed with the four-carrier GSM signal described in section 3.2. It must be noted that this signal is completely uncorrelated with the signals used during the STA algorithm and the training of the ANN.

Figure 7.9 is a plot of the frequency spectrum of the output of the VSPA in response to the GSM signal with (in red) and without (in blue) the predistorting ANN. It is very clear that the linearization goal was met: the spurious distortion tones were nearly completely eliminated. There appear to be some very minor distortion tones between each of the four GSM carriers, as well as a DC offset that was later modulated to 10 MHz, but these can be attenuated by making sure the Backpropagation phase of the algorithm does not degrade the linearization performance by a significant amount.

The STA algorithm has, therefore, been verified as an exceptionally fast and accurate method of generating a predistorting ANN for a static PA.

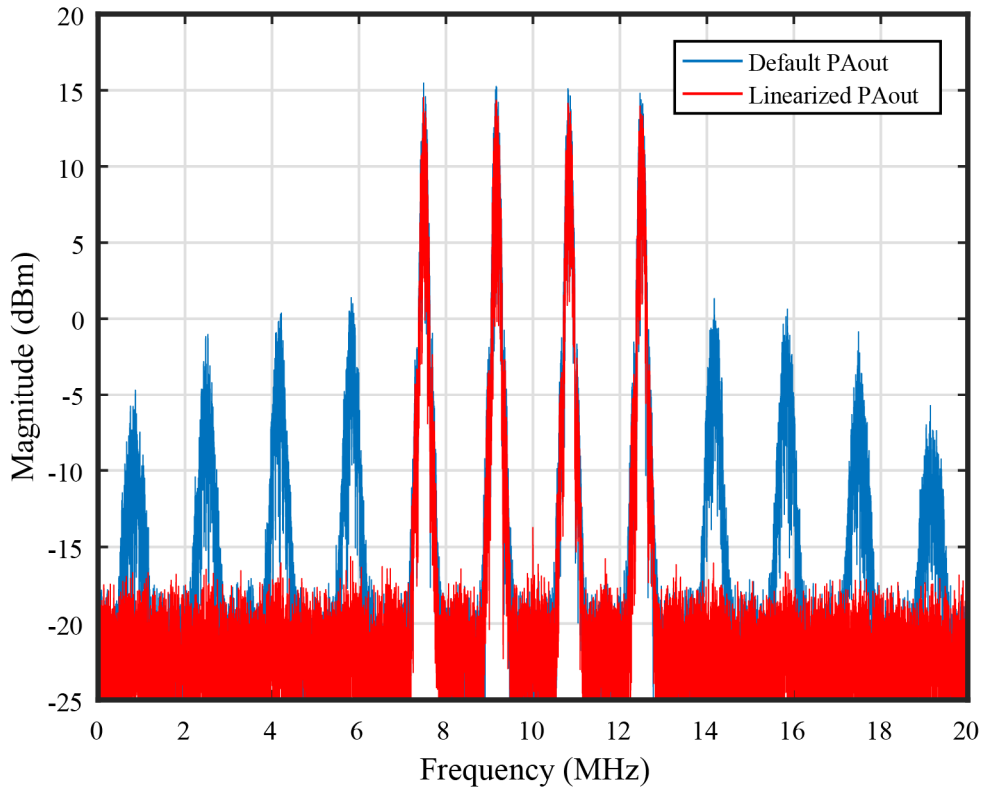


Figure 7.9. Output spectrum of the VSPA in response to the GSM signal with (red) and without (blue) predistortion by the ANN generated using STA.

8. CONCLUSION

A new generation of telecommunications networks requires a new generation of linearization systems for the power amplifiers they rely on. Thus, a base band analog predistorter implemented as an artificial neural network was proposed as a solution.

Traditionally, ANNs are trained in a supervised manner. This, however, goes against the very essence of the problem of predistortion: to *find* the optimal predistortion function. Roundabout ways of solving this paradox have been documented in the literature, such as training the ANN as a post-distorter and testing it as a predistorter.

In this dissertation, three different alternative training methods are explored: Temporal Difference learning, optimization through evolution strategies, and a custom algorithm which enables the use of the Backpropagation algorithm.

8.1. Results Summary

Despite our best efforts, the Temporal Difference learning method proved to be unsuccessful. While initially it was thought to be a good candidate for a solution, our results suggest the opposite, and some later knowledge on the true meaning of “policy evaluation” confirm that these results were, ultimately, inevitable. Alas, failure is also part of the research process.

Optimization using CMA-ES produced predistorting ANNs with exceptional performance, completely erasing any sign of intermodulation distortion introduced by the base band model of a static PA. The only downside to this method was the processing time, on the order of minutes to hours – even with a custom-made implementation of an ANN that is 100 times faster than that of Matlab’s Neural Network Toolbox. Naturally, this can be improved by resorting to a proper computation platform and by finely tuning some of its configuration parameters.

Finally, the original Successive Target Approximation algorithm proved to be astonishingly fast and produced excellent results as well, also eliminating all distortion tones. This algorithm enables the use of the Backpropagation algorithm for the training of the ANN. While this introduces a penalization in both processing time and linearization performance, these are still potentially better than those of the CMA-ES algorithm. A tighter integration of the STA and the Backpropagation algorithms would surely make for a better-performing solution.

8.2. Future Work

There is still plenty of research left to do, especially concerning the analog implementation of the predistortion system. Other topics include the determination of the optimal size of the ANN to be used as a predistorter, as well as the linearization of a dynamical (with memory) model of a PA, as opposed to a static one.

8.2.1. Dynamical Systems

While the linearization of a static model of a PA is a good start, a more complete solution would need to be able to linearize a dynamical model, which features the memory effects present in most real amplifiers.

This problem requires a completely different approach to the training of the networks: for instance, the order of the input symbols would be one of the many additional factors to take into consideration. While there are techniques that pretend to solve this issue, they are far from optimal. Some preliminary original work has been done regarding the generation of an optimal, minimum-sized input sequence that covers the complete output vector space of a dynamical PA, but it shall not be published in this document at this stage.

Some brief tests were done on a dynamical base band model of a PA with a one-sample memory depth, but there was no time to draw definite conclusions – especially because of the processing time, which increases dramatically for such systems. It is to be expected, though, that the CMA-ES algorithm should remain a good solution, but the STA algorithm should fail very quickly without any modifications.

8.2.2. Towards Analog

In order to begin the understanding of what an analog implementation of an ANN might implicate, some modifications were done to the networks generated by the CMA-ES and the STA algorithms and a brief, final test was conducted. In this test, it was assumed that the two networks were implemented as analog circuits, and that their weights were set by external voltages with a 1 mV resolution.

First, it should be noted that this is a perfectly acceptable assumption, because the weights have relatively low values: the CMA-ES ANN has weights with absolute values between 0.009 and 8.217, and the STA ANN has weights with absolute values between 0.001 and 5.479. If these were voltage, they could be produced by any commercially available DAC.

Figures 8.1 and 8.2 show the frequency spectra of the outputs of the two ANNs with their weights rounded to three decimal places. It is clear, and expected, that the limiting of the precision in the definition of the network weights introduces distortion in the system, especially as a DC (zero Hz) component (that was later modulated to 10 MHz).

This can be easily solved by a low pass filter at the output of the ANN, though it might actually be possible do it by training the ANN with limited-precision weights – as opposed to performing the training with double precision weights and later rounding them to three decimal places. Had we had more time, that would have been an interesting experiment: let the ANN solve, by itself, the problems introduced by the limited precision of its own weights.

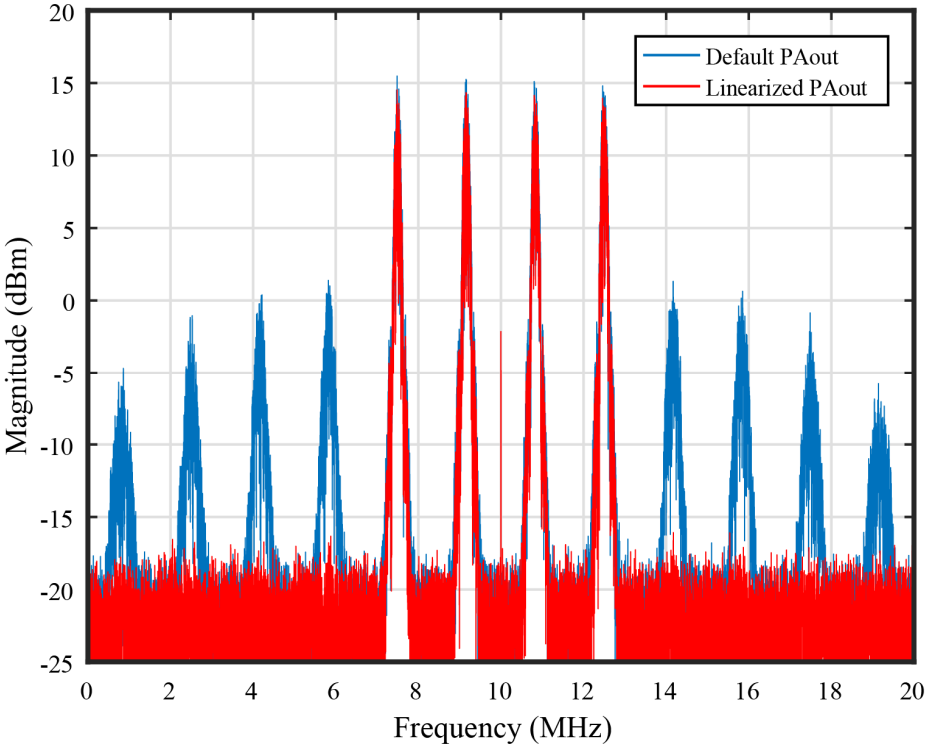


Figure 8.1. Output spectrum of the CMA-ES linearization system with the ANN weights rounded to three decimal places (1 mV resolution).

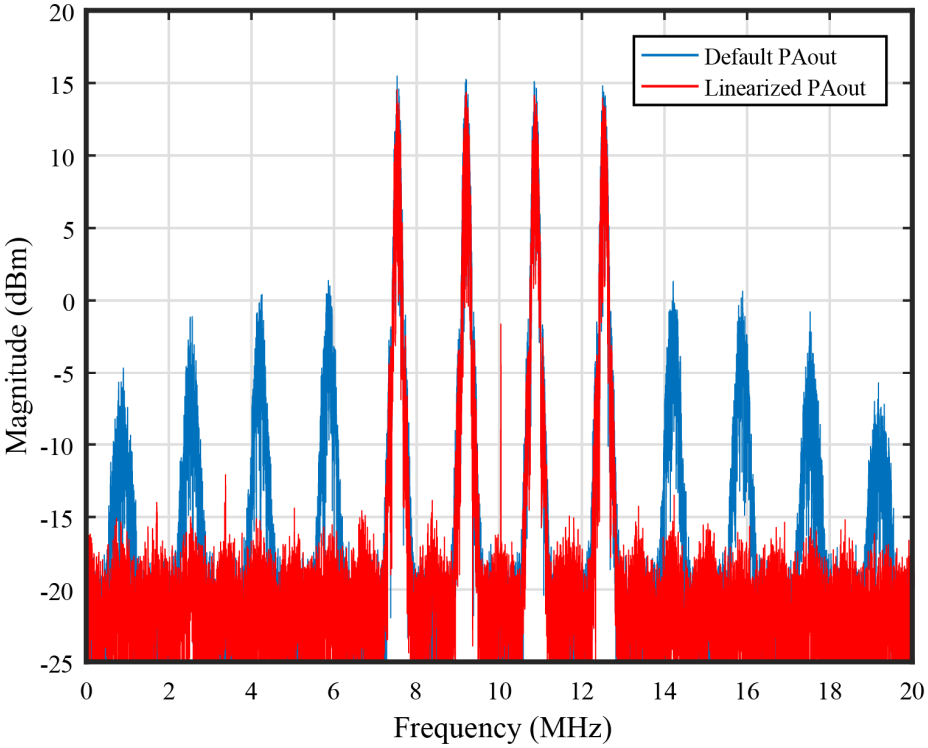


Figure 8.2. Output spectrum of the STA linearization system with the ANN weights rounded to three decimal places (1 mV resolution).

9. REFERENCES

[1] P. Tomé, Analog Neural Predistortion of Power Amplifiers, Master's Thesis, Tampere University of Technology, 2016.

[2] D. Warren, C. Dewar, Understanding 5G: Perspectives on Future Technological Advancements in Mobile, GSMA Intelligence, 2014. Available: https://www.gsmaintelligence.com/files/analysis/?file=141208-5g.pdf

[3] H. Wang, X. Zhou, M. Reed, Coverage and Throughput Analysis with a Non-Uniform Small Cell Deployment, IEEE Transactions on Wireless Communications, 2014, Vol. 13, pp. 2047–2059.

[4] Samsung, 5G Vision, White Paper, 2015. Available: http://www.samsung.com/global/business-images/insights/2015/Samsung-5G-Vision-2.pdf

[5] D. Bharadia, E. McMilin, S. Katti, Full Duplex Radios, ACM SIGCOMM Computer Communication Review, 2013, Vol. 43, pp. 375–386.

[6] M. Cho, J. Kenney, Variable Phase Shifter Design for Analog Predistortion Power Amplifier Linearization System, Proceedings of Wireless and Microwave Technology Conference (WAMICON), Orlando, FL, USA, Apr 7–9, 2013, pp. 1–5.

[7] P. Fisher, S. Al-Sarawi, Analog RF Predistorter Simulation using Well-Known Behavioral Models, Proceedings of 10th Conference on Industrial Electronics and Applications (ICIEA), Auckland, New Zealand, Jun 15–17, 2015, pp. 1667–1671.

[8] N. Benvenuto, F. Piazza, A. Uncini, A Neural Network Approach to Data Predistortion with Memory in Digital Radio Systems, IEEE International Conference on Communications, Geneva, Switzerland, May 23–26, 1993, pp. 232–236.

[9] Y. Qian, F. Liu, Neural Network Predistortion Technique for Nonlinear Power Amplifiers with Memory, First International Conference on Communications and Networking in China, Beijing, China, Oct 25–27, 2006, pp. 1–5.

[10] B. Watkins, R. North, Predistortion of Nonlinear Amplifiers Using Neural Networks, Proceedings of Military Communications Conference, McLean, VA, USA, Oct 21–24, 1996, pp. 316–320.

[11] B. Mulliez, E. Moutaye, H. Tap, L. Gatet, F. Gizard, Predistortion System Implementation Based on Analog Neural Networks for Linearizing High Power Amplifiers Transfer Characteristics, Proceedings of the 36th International Conference on Telecommunications and Signal Processing, Rome, Italy, Jul 2–4, 2013, pp. 412–416.

[12] M. Ngwar, An Analog Neural Network for Wideband Predistortion of Pico-cell Power Amplifiers, Doctoral Thesis, Carleton University, 2015. Available: https://curve.carleton.ca /2d167908-4475-4ab3-b192-eeb4c656e2de

[13] A. Katz, J. Wood, D. Chokola, The Evolution of PA Linearization, IEEE Microwave Magazine, 2016, Vol. 17, pp. 32–40.

[14] N. Carvalho, R. Madureira, Intermodulation Interference in the GSM/UMTS Bands, Proceedings of 3ª Conferência de Telecomunicações (ConfTele 2001), Figueira da Foz, Portugal, April 23–24, 2001.

[15] J. Dawson, T. Lee, Cartesian Feedback for RF Power Amplifier Linearization, Proceedings of 2004 American Control Conference, Boston, MA, USA, Jun 30 – Jul 2, 2004, Vol. 1, pp. 361–366.

[16] S. Stapleton, Presentation on Adaptive Feedforward Linearization for RF Power Amplifiers, Agilent Technologies, 2001. Available: http://literature.agilent.com/litweb /pdf/5989-9106EN.pdf

[17] R. Gupta, S. Ahmad, R. Ludwig, J. McNeill, Adaptive Digital Baseband Predistortion for RF Power Amplifier Linearization, High Frequency Electronics, 2006, Vol. 5, No. 9, pp. 16–25.

[18] A. Clarke, Extra-Terrestrial Relays, Wireless World, No. 10, 1945, pp. 305–308.

[19] J. Pierce, Orbital Radio Relays, Jet Propulsion, 1955, Vol. 25, pp. 153–157.

[20] H. Girard, K. Feher, A New Baseband Linearizer for More Efficient Utilization of Earth Station Amplifiers Used for QPSK Transmission, IEEE Journal on Selected Areas in Communications, 1983, Vol. 1, pp. 46–56.

[21] G. Satoh, T. Mizuno, Impact of a New TWTA Linearizer Upon QPSK/TDMA Transmission Performance, IEEE Journal on Selected Areas in Communications, 1983, Vol. 1, pp. 39–45.

[22] C. Haskins, Diode Predistortion Linearization for Power Amplifier RFICs in Digital Radios, Master's Thesis, Virginia Polytechnic Institute and State University, 2000. Available: https://theses.lib.vt.edu/theses/available/etd-04252000-16200028/unrestricted/chaskins_thesis.pdf

[23] K. Yamauchi, K. Mori, M. Nakayama, A Novel Series Diode Linearizer for Mobile Radio Power Amplifiers, IEEE International Microwave Symposium, S. Francisco, CA, USA, Jun 17–21, 1996, Vol. 2, pp. 831–834.

[24] J. Yi, M. Park, W. Kang, B. Kim, Analog Predistortion Linearizer for High-Power RF Amplifiers, IEEE Transactions on Microwave Theory and Techniques, 2000, Vol. 48, pp. 2709–2713.

[25] S. Lee, Y. Lee, S. Hong, H. Choi, Y. Jeong, Independently Controllable 3rd- and 5th-Order Analog Predistortion Linearizer for RF Power Amplifier in GSM, Proceedings of 2004 IEEE Asia-Pacific Conference on Advanced System Integrated Circuits, Fukuoka, Japan, Aug 4–5, 2004, pp. 146–149.

[26] Y. Lee, M. Lee, S. Jung, Y. Jeong, Analog Predistortion Power Amplifier Using IMD Sweet Spots for WCDMA Applications, Proceedings of 2007 Asia-Pacific Microwave Conference, Bangkok, Thailand, Dec 11–14, 2007, pp. 1–4.

[27] X. Sun, S. Cheung, T. Yuk, Design of a Fifth-order Analog Predistorter for Base Station HPA of Cellular Mobile Systems, Microwave Journal, 2011, Vol. 54, pp. 86–102.

[28] E. Westesson, L. Sundström, A Complex Polynomial Predistorter Chip in CMOS for Baseband or IF Linearization of RF Power Amplifiers, Proceedings of the 1999 IEEE International Symposium on Circuits and Systems, Orlando, FL, USA, May 30 – Jun 2, 1999, Vol. 1, pp. 206–209.

[29] E. Westesson, L. Sundström, Low-Power Complex Polynomial Predistorter Circuit in CMOS for RF Power Amplifier Linearization, Proceeding of the 27th Solid-State Circuits Conference, Villach, Austria, Sep 18–20, 2001, pp. 486–489.

[30] F. Roger, A 200mW 100MHz-to-4GHz 11th-Order Complex Analog Memory Polynomial Predistorter for Wireless Infrastructure RF Amplifiers, Proceedings of 2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers, San Francisco, CA, USA, Feb 17–21, 2013, pp. 94–95.

[31] B. Gilbert, A Precise Four-Quadrant Multiplier with Subnanosecond Response, IEEE Journal of Solid-State Circuits, 1968, Vol. 3, pp. 365–373.

[32] Y. Lee, M. Lee, S. Kam, Y. Jeong, Analog Predistortion Linearization of Doherty Power Amplifiers Using Bandwidth Reduction of Error Signal, Microwave and Optical Technology Letters, 2010, Vol. 52, pp. 1313–1316.

[33] K. Fayed, A. Ezzeddine, H. Huang, Linear Power Amplifier Uses Mirror Predistortion, High Frequency Electronics, 2011, Vol. 10, No. 6, pp. 18–25.

[34] K. Son, B. Koo, S. Hong, A CMOS Power Amplifier With a Built-In RF Predistorter for Handset Applications, IEEE Transactions on Microwave Theory and Techniques, 2012, Vol. 60, pp. 2571–2580.

[35] K. Hornik, M. Stinchcombe, H. White, Multilayer Feedforward Networks Are Universal Approximators, Neural Networks, 1989, Vol. 2, pp. 359–366.

[36] M. Bianchini, F. Scarselli, On the Complexity of Neural Network Classifiers: A Comparison Between Shallow and Deep Architectures, IEEE Transactions on Neural Networks and Learning Systems, 2014, Vol. 25, pp. 1553–1565.

[37] G. Cybenko, Approximation by Superposition of a Sigmoidal Function, Mathematics of Control, Signals and Systems, 1989, Vol. 2, pp. 303–314.

[38] K. Hornik, Approximation Capabilities of Multilayer Feedforward Networks, Neural Networks, 1991, Vol. 4, pp. 251–257.

[39] G. Tesauro, T. Seinowski, A Parallel Network that Learns to Play Backgammon, Artificial Intelligence, 1989, Vol. 39, pp. 357–390.

[40] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, et al., Mastering the game of Go with deep neural networks and tree search, Nature, 2016, Vol. 529, pp. 484–489.

[41] C. Anderson, Learning to Control an Inverted Pendulum Using Neural Networks, IEEE Control Systems Magazine, 1989, Vol. 9, pp. 31–37.

[42] J. Lagarias, J. Reeds, M. Wright, P. Wright, Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions, SIAM Journal of Optimization, 1998, Vol. 9, pp. 112–147.

[43] M. Hagan, M. Menhaj, Training Feed-Forward Networks with the Marquardt Algorithm, IEEE Transactions on Neural Networks, 1994, Vol. 5, pp. 989–993

[44] J. McClelland, Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises, Second Edition, DRAFT, 2015.

[45] R. Sutton, Learning to Predict by the Methods of Temporal Differences, Machine Learning, 1988, Vol. 3, pp. 9–44.

[46] A. Lazaric, M. Restelli, A. Bonarini, Reinforcement Learning in Continuous Action Spaces Through Sequential Monte Carlo Methods, Advances in Neural Information Processing Systems 20 (NIPS 2007), 2007, pp. 1456–1463.

[47] J. Modayil, A. White, P. Pilarski, R. Sutton, Acquiring a Broad Range of Empirical Knowledge in Real Time by Temporal-Difference Learning, 2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC), Seoul, South Korea, Oct 14–17, 2012, pp. 1903–1910.

[48] R. Sutton, A. Bonde, Nonlinear TD/Backprop Pseudo C-code, 1993. Available: https://webdocs.cs.ualberta.ca/~sutton/td-backprop-pseudo-code.text

[49] D. Fogel, An Introduction to Simulated Evolutionary Optimization, IEEE Transactions on Neural Networks, 1994, Vol. 5, pp. 3–14.

[50] N. Hansen, D. Arnold, A. Auger, Evolution Strategies, in J. Kacprzyk, W. Pedrycz, Handbook of Computational Intelligence, Springer, 2015, pp. 871–898.

[51] N. Hansen, The CMA Evolution Strategy: A Comparing Review, in J. Lozano, P. Larrañga, I. Inza, E. Bengoetxea, Towards a New Evolutionary Computation: Advances in Estimation of Distribution Algorithms, Springer, 2006, pp. 75–102.

[52] N. Hansen, CMA-ES Source Code, Matlab Production Code, Version 3.61.beta, 2012. Available: https://www.lri.fr/~hansen/cmaes_inmatlab.html#matlab

[53] During the public defense of this dissertation, the author was informed that the Successive Target Approximation algorithm had been previously published in a patent (under another name). It should be stressed, though, that this patent was unknown to the author prior to the public defense of this dissertation, and that the Successive Target Approximation algorithm was, in fact, an original effort by the author of this dissertation. J. Peroulas, A. You, Method and Apparatus for Linearizing a Non-Linear Power Amplifier, US8482349, 2013.

APPENDIX A: VECTORIZED TDNN MODEL AND LEARNING ALGORITHM (MATLAB)

```
% Temporal Difference Neural Network
% File: TDNN.m
% Author: Pedro Tomé (tome.p.m at ua.pt)
%
%
% To be used under the terms of the GNU General Public License:
% http://www.gnu.org/copyleft/gpl.html
%
%
classdef TDNN
    properties (GetAccess = 'public', SetAccess = 'private')
        RAND_INIT_EPSILON;% Random weight initialization scaling factor

        numInputs;  % Number of input nodes (excluding bias node)
        numHidden;  % Number of hidden nodes (excluding bias node)
        numOutputs; % Number of output nodes

        BIAS;       % Activation of the (constant) bias nodes
        GAMMA;      % Discount rate parameter (typically 0.9)
        LAMBDA;     % Trace decay parameter (should be <= GAMMA)
        ALPHA;      % Learning rate of v (typically 1/numInputs)
        BETA;       % Learning rate of w (typically 1/numHidden)

        x; h; y;    % Neuron activations for layers 1 to 3
        v; w;       % Weights between layers 1 and 2 and layers 2 and 3

        oldY;       % Last output
        ev; ew;     % Eligibility traces of v and w
        error;      % TD error
    end


    methods (Access = 'public')
        function self = TDNN(numInputs, numHidden, numOutputs)
            validateattributes(numInputs, ...
                {'numeric'}, {'scalar', 'positive', 'integer'}, '', ...
                'numInputs');
            validateattributes(numHidden, ...
                {'numeric'}, {'scalar', 'positive', 'integer'}, '', ...
                'numHidden');
            validateattributes(numOutputs, ...
                {'numeric'}, {'scalar', 'positive', 'integer'}, '', ...
                'numOutputs');


```

```
46              self.numInputs = numInputs;
47              self.numHidden = numHidden;
48              self.numOutputs = numOutputs;
49
50
51              self.RAND_INIT_EPSILON = 0.5;
52              self.BIAS   = 1;
53              self.GAMMA  = 0;
54              self.LAMBDA = 0;
55              self.ALPHA  = 0;
56              self.BETA   = 0;
57
58              self = self.init();
59          end
60
61
62          function self = train(self, netInput, reward, gamma, ...
63                                lambda, alpha, beta)
64              validateattributes(self, ...
65                  {'TDNN'}, {}, '', 'self');
66              validateattributes(netInput, ...
67                  {'numeric'}, {'nrows', self.numInputs},'', 'netInput');
68              validateattributes(reward, ...
69                  {'numeric'}, {'size', size(netInput)}, '', 'reward');
70              validateattributes(gamma, ...
71                  {'numeric'}, {'scalar', 'nonnegative'}, '', 'gamma');
72              validateattributes(lambda, ...
73                  {'numeric'}, {'scalar', 'nonnegative'}, '', 'lambda');
74              validateattributes(alpha, ...
75                  {'numeric'}, {'scalar', 'positive'}, '', 'alpha');
76              validateattributes(beta, ...
77                  {'numeric'}, {'scalar', 'positive'}, '', 'beta');
78
79
80              self.GAMMA = gamma;
81              self.LAMBDA = lambda;
82              self.ALPHA = alpha;
83              self.BETA = beta;
84
85
86              t = 1;
87              self = self.forwardProp(netInput(:,t));
88              self.oldY = self.y;
89              self = self.updateEligTraces();
90
91              for t = 2 : size(netInput, 2)
92                  self = self.forwardProp(netInput(t));
93                  self.error = reward(t) + self.GAMMA*self.y - self.oldY;
94                  self = self.updateWeights();
95
96                  self = self.forwardProp(netInput(t));
97                  self.oldY = self.y;
98                  self = self.updateEligTraces();
99              end
100         end
101
102
```

```
103         function netOutput = output(self, netInput)
104             validateattributes(self, ...
105                 {'TDNN'}, {}, '', 'self');
106             validateattributes(netInput, ...
107                 {'numeric'}, {'nrows', self.numInputs},'', 'netInput');
108 
109             numSamples = size(netInput, 2);
110             netOutput = zeros(self.numOutputs, numSamples);
111 
112             for t = 1:numSamples
113                 [~, tmp] = self.forwardProp(netInput(:,t));
114                 netOutput(:,t) = tmp;
115             end
116         end
117 
118 
119         % Allows the use of the following syntax
120         %     netOutput = net(netInput)
121         % equivalent to
122         %     netOutput = net.output(netInput)
123         % where 'net' is an object of class TDNN.
124         function varargout = subsref(obj, s)
125             switch s(1).type
126                 case '()'
127                     input = s.subs{:};
128                     varargout = {obj.output(input)};
129                 case '.'
130                     c = class(obj);
131                     fname = strcat(c, '>', c, '.', s(1).subs);
132                     n = nargout(fname);
133                     [varargout{1:n}] = builtin('subsref', obj, s);
134             end
135         end
136     end
137 
138 
139     methods (Access = 'private')
140         function self = init(self)
141             % Neuron Activations Initialization
142             self.x = [self.BIAS ; zeros(self.numInputs, 1)];
143             self.h = [self.BIAS ; zeros(self.numHidden, 1)];
144             self.y = zeros(self.numOutputs, 1);
145             self.oldY = zeros(self.numOutputs, 1);
146 
147             self.error = 0;
148 
149             % Random Weight Initialization
150             self.v = rand(self.numHidden, self.numInputs + 1) * 2 * ...
151                      self.RAND_INIT_EPSILON - self.RAND_INIT_EPSILON;
152             self.w = rand(self.numOutputs, self.numHidden + 1)* 2 * ...
153                      self.RAND_INIT_EPSILON - self.RAND_INIT_EPSILON;
154 
155             % Eligibility Traces Initialization
156             self.ev = zeros(self.numHidden, self.numInputs + 1, ...
157                             self.numOutputs);
158             self.ew = zeros(self.numOutputs, self.numHidden + 1);
159         end
```

```
160
161
162         function [self, output] = forwardProp(self, input)
163             self.x(2:end) = input;
164
165             self.h(2:end) = tansig(self.v * self.x);
166             %self.h(2:end) = 1 ./ (1 + exp(-(self.v * self.x)));
167
168             self.y = purelin(self.w * self.h);
169             %self.y = tansig(self.w * self.h);
170             output = self.y;
171         end
172
173
174         function self = updateWeights(self)
175             self.w = self.w + self.BETA * repmat(self.error, 1, ...
176                      self.numHidden + 1) .* self.ew;
177
178             dv = zeros(size(self.v));
179             for k = 1 : self.numOutputs
180                 dv = dv + self.error(k) * self.ev(:,:,k);
181             end
182             self.v = self.v + self.ALPHA * dv;
183         end
184
185
186         function self = updateEligTraces(self)
187             deltaY = ones(size(self.y));      % Output nodes: purelin()
188             %deltaY = self.y .* (1 - self.y); % Output nodes: tansig()
189             deltaH = self.h .* (1 - self.h);  % Hidden nodes: tansig()
190
191             self.ew = self.LAMBDA * self.ew + deltaY * self.h';
192
193             tmp = deltaY * deltaH(2:end)' .* self.w(:,2:end);
194             for k = 1 : self.numOutputs
195                 self.ev(:,:,k) = self.LAMBDA * self.ev(:,:,k) + ...
196                                  (self.x * tmp(k,:))';
197             end
198         end
199     end
200 end
```

APPENDIX B: EXAMPLE USAGE OF THE TDNN MODEL AND LEARNING ALGORITHM (MATLAB)

```
% Example Usage of the TDNN Class
% File: Test_TDNN.m
% Author: Pedro Tomé (tome.p.m at ua.pt)
%
%
% To be used under the terms of the GNU General Public License:
% http://www.gnu.org/copyleft/gpl.html
%
%
close all;

%% Define the PA's transfer function, input range and target output
PA = @(x) tanh(3 * x);
input = linspace(-1, 1, 100);
targetOutput = input;

% Try this with reward = targetOutput, gamma = 0, and lambda = 0.
% Backpropagation at a snail's pace.
% targetOutput = 0.75*sin(pi*input) + 0.25*sin(3*pi*input);


%% Prepare figures
figure();

subplot(1,5,[1 3]);
plot(input, PA(input), 'b'); hold on;
plot(input, targetOutput, 'k--');
h_netOut = plot(input, nan(size(input)), 'g');
h_PAout = plot(input, nan(size(input)), 'r');
h_reward = plot(input, nan(size(input)), 'k');
xlabel('Inputs'); ylabel('Outputs'); grid on;
legend('Default PAout', 'Target PAout', 'PDout', ...
       'Linearized PAout', 'Reward', 'Location', 'SouthEast');

subplot(1,5,4);
netOut_error_history = NaN;
h_netOutError = plot(netOut_error_history); grid on;
xlabel('Iteration'); ylabel('MSE(reward - netOut) (dB)');

subplot(1,5,5);
PAout_error_history = NaN;
h_PAoutError = plot(PAout_error_history); grid on;
xlabel('Iteration'); ylabel('MSE(targetOutput - PAout)(dB)');


```

```
%% Create and train the PD
TDnet = TDNN(1, 10, 1);
netOut = TDnet(input);
PAout = PA(netOut);

iteration = 0;
while (true)
    iteration = iteration + 1;

    E = targetOutput - PAout;
    SE = E .^ 2;
    MSE = mean(SE);
    % Pick your poison:
    %reward = [zeros(1, length(input) - 1)  ,  +1 * MSE];
    %reward = [zeros(1, length(input) - 1)  ,  -1 * MSE];
    %reward = +1 * MSE * ones(1, length(input));
    %reward = -1 * MSE * ones(1, length(input));
    %reward = +1 * SE;
    %reward = -1 * SE;
    reward = E;
    %reward = targetOutput;


    % Train the network                    gamma,lambda   alpha,beta
    TDnet = TDnet.train(input, reward,      0.0,0.3,       0.1,0.1);


    % Calculate performance measures
    netOut = TDnet(input);
    PAout = PA(netOut);

    netOut_error = mean((reward - netOut) .^ 2);
    netOut_error_history(iteration) = netOut_error;
    PAout_error = mean((targetOutput - PAout) .^ 2);
    PAout_error_history(iteration) = PAout_error;


    % Refresh figures
    if (mod(iteration,10) == 0)
        set(h_netOut, 'YData', netOut);
        set(h_PAout,  'YData', PAout);
        set(h_reward, 'YData', reward);
        set(h_netOutError,'YData', 10*log10(abs(netOut_error_history)));
        set(h_PAoutError, 'YData', 10*log10(abs(PAout_error_history)));
        drawnow();
    end
end
```

APPENDIX C: OPTIMIZATION USING THE CMA-ES ALGORITHM (MATLAB)

```
% Example Usage of the CMA-ES Algorithm
% File: Test_CMAES.m
% Author: Pedro Tomé (tome.p.m at ua.pt)
%
%
% Uses cmaes.m, version 3.61.beta, by Nikolaus Hansen,
% with slight modifications for monitoring purposes.
%
%
% To be used under the terms of the GNU General Public License:
% http://www.gnu.org/copyleft/gpl.html
%
%
function Test_CMAES()
    %% Program setup
    netHiddenSize = [10 10 10];
    maxInputAmplitude = 1;


    close all;
    %% Define the input and target output signals
    [netInput_I, netInput_Q] = iqGrid(maxInputAmplitude, 0.1);
    netInput = [netInput_I' ; netInput_Q'];

    ampOutput_targetI = netInput_I;
    ampOutput_targetQ = netInput_Q;
    ampOutput_target = [ampOutput_targetI , ampOutput_targetQ];


    %% Create Artificial Neural Network
    net = FastANN(2, netHiddenSize, 2);
    startingWeights = getwb(net);


    %% Create IQ monitoring figure
    handles = createMonitoringFigure(net, netInput, ampOutput_target);


    %% Run Optimization Algorithm
    projectSettings.net = net;
    projectSettings.netInput = netInput;
    projectSettings.ampOutput_target = ampOutput_target;
    projectSettings.handles = handles;

```

```
% The cmaes function was modified a bit for monitoring purposes
    [xmin, fmin, counteval, stopflag, out, bestever] = cmaes( ...
        'CMAES_costFunction',                                  ...
        startingWeights,                                       ...
        0.1,                                                   ...
        [], projectSettings                                    ...
    );
end


function [I, Q] = iqGrid(maxAmplitude, delta)
    %% Create grid of (I,Q) points
    [I, Q] = meshgrid([-1 : delta : 1], [-1 : delta : 1]);
    I = I(:) * maxAmplitude;
    Q = Q(:) * maxAmplitude;


    %% Exclude points outside the maxAmplitude radius
    indices = sqrt(I.^2 + Q.^2) < maxAmplitude;
    I = I(indices);
    Q = Q(indices);
end


function handles = createMonitoringFigure(net,netInput,ampOutput_target)
    [ampOutput_I, ampOutput_Q] = VirtualStaticPA(netInput(1,:)', ...
                                                 netInput(2,:)');
    netOut = net(netInput);
    [ampOutput_Ipd, ampOutput_Qpd] = VirtualStaticPA(netOut(1,:)', ...
                                                     netOut(2,:)');

    %% IQ Mapping
    figure();
    plot(ampOutput_I, ampOutput_Q, 'k.'); hold on;
    plot(ampOutput_target(:,1), ampOutput_target(:,2), 'r.');
    handles.ampOutput = plot(ampOutput_Ipd, ampOutput_Qpd, 'b+');
    handles.netOutput = plot(netOut(1,:), netOut(2,:), 'g.');
    xlabel('I Component'); ylabel('Q Component');
    legend('Default PAout', 'Target PAout', 'Linearized PAout', ...
           'PDout', 'Location', 'SouthEast');


    %% IQ Mapping Error
    figure();
    handles.perf = plot(NaN, NaN);
    xlabel('Iteration'); ylabel('Function Value (dB)');

    drawnow();
end
```

```
% Cost Function (NMSE) for CMA-ES Optimization
% File: CMAES_costFunction.m
% Author: Pedro Tome' (tome.p.m at ua.pt)
%
%
% To be used under the terms of the GNU General Public License:
% http://www.gnu.org/copyleft/gpl.html
%
%
function netFitness = CMAES_costFunction(netWeights, options)
    %% Parse input options
    net = options.net;
    netInput = options.netInput;
    ampOutput_target = options.ampOutput_target;
    PAout_It = ampOutput_target(:,1);
    PAout_Qt = ampOutput_target(:,2);


    %% Configure ANN with input weights
    net = setwb(net, netWeights);


    %% Compute ANN output
    netOut = net(netInput);
    netOut_I = netOut(1,:)';
    netOut_Q = netOut(2,:)';


    %% Compute Linearized PA output
    [PAout_Ipd, PAout_Qpd] = VirtualStaticPA(netOut_I, netOut_Q);


    %% Compute ANN fitness
    squareError = (PAout_Ipd - PAout_It).^2 + (PAout_Qpd - PAout_Qt).^2;
    meanError = (PAout_Ipd - mean(PAout_It)).^2 + ...
                (PAout_Qpd - mean(PAout_Qt)).^2;
    NMSE = sum(squareError) / sum(meanError);

    netFitness = NMSE;
end
```

APPENDIX D: FAST IMPLEMENTATION OF AN ARTIFICIAL NEURAL NETWORK (MATLAB)

```
% Fast Implementation of an Artificial Neural Network
% File: FastANN.m
% Author: Pedro Tomé (tome.p.m at ua.pt)
%
%
% Example Usage:
%   1. Create an ANN with 2 input nodes, three hidden layers
%      of 10 nodes each, and 2 output nodes:
%          net = FastANN(2, [10 10 10], 2);
%   2. Extract the number of weights and biases of the network:
%          numWeights = length( getwb(net) );
%   3. Set the weights and biases to whatever:
%          net = setwb(net, rand(1, numWeights));
%   4. Calculate the network's output using Forward Propagation:
%          netInput = rand(2, 1000);
%          netOutput = net(netInput);
%
%
% To be used under the terms of the GNU General Public License:
% http://www.gnu.org/copyleft/gpl.html
%
%
classdef FastANN
    properties (GetAccess = 'public', SetAccess = 'public')
        RAND_INIT_EPSILON;
        BIAS;        % Activation of the (constant) bias nodes

        numInputs;   % Number of input nodes (excluding bias node)
        numHidden;   % Number of hidden nodes (excluding bias node)
        numOutputs;  % Number of output nodes

        numLayers;   % Number of layers, including input and output

        weights;     % The defining parameters of the network
    end


    methods (Access = 'public')
        function self = FastANN(numInputs, numHidden, numOutputs)
            validateattributes(numInputs, ...
                {'numeric'}, {'scalar', 'positive', 'integer'}, '', ...
                'numInputs');
            validateattributes(numHidden, ...
                {'numeric'}, {'vector', 'positive', 'integer'}, '', ...
                'numHidden');
```

```
46              validateattributes(numOutputs, ...
47                  {'numeric'}, {'scalar', 'positive', 'integer'}, '', ...
48                  'numOutputs');
49
50              self.numInputs = numInputs;
51              self.numHidden = numHidden;
52              self.numOutputs = numOutputs;
53
54              self.numLayers = 1 + length(numHidden) + 1;
55
56              self.RAND_INIT_EPSILON = 0.5;
57              self.BIAS = 1;
58
59              self = self.init();
60          end
61
62
63          function netOutput = output(self, netInput)
64              [~, netOutput] = self.forwardProp(netInput);
65          end
66
67
68          % Allows the use of the following syntax
69          %     netOutput = net(netInput)
70          % equivalent to
71          %     netOutput = net.output(netInput)
72          % where 'net' is an object of class TDNN.
73          function varargout = subsref(obj, s)
74              switch s(1).type
75                  case '()'
76                      input = s.subs{:};
77                      varargout = {obj.output(input)};
78                  case '.'
79                      c = class(obj);
80                      fname = strcat(c, '>', c, '.', s(1).subs);
81                      n = nargout(fname);
82                      [varargout{1:n}] = builtin('subsref', obj, s);
83              end
84          end
85
86
87          % Set network weights externally
88          function self = setwb(self, weights)
89              wVector = weights(1 : numel(self.weights{1}));
90              self.weights{1} = reshape(wVector, size(self.weights{1}));
91
92              pointer = numel(self.weights{1});
93              for i = 2 : length(self.weights);
94                  wVector = weights([1:numel(self.weights{i})]+pointer);
95                  self.weights{i}=reshape(wVector,size(self.weights{i}));
96                  pointer = pointer + numel(self.weights{i});
97              end
98          end
99
100
```

```
        % Get network weights
        function weightsOut = getwb(self)
            weightsOut = [];
            for i = 1 : length(self.weights)
                tmpW = self.weights{i};
                weightsOut = [weightsOut ; tmpW(:)];
            end
        end
    end


    methods (Access = 'private')
        function self = init(self)
            % Random Weight Initialization
            self.weights = cell(1, self.numLayers-1);

            self.weights{1} = rand(self.numHidden(1), ...
                self.numInputs + 1) *                  ...
                (2 * self.RAND_INIT_EPSILON) - self.RAND_INIT_EPSILON;
            for i = 2 : self.numLayers - 2
                self.weights{i} = rand(self.numHidden(i), ...
                    self.numHidden(i-1) + 1) *             ...
                    (2 * self.RAND_INIT_EPSILON)-self.RAND_INIT_EPSILON;
            end
            self.weights{end} = rand(self.numOutputs, ...
                self.numHidden(end) + 1) *             ...
                (2 * self.RAND_INIT_EPSILON) - self.RAND_INIT_EPSILON;
        end


        function [self, output] = forwardProp(self, input)
            N = size(input,2);
            b = self.BIAS * ones(1,N);

            x = [b ; input];
            h = tansig( [b ; self.weights{1} * x] );
            for i = 2 : self.numLayers - 2
                h = tansig( [b ; self.weights{i} * h] );
            end
            y = purelin( self.weights{end} * h );

            output = y;
        end
    end
end
```

APPENDIX E: SUCCESSIVE TARGET APPROXIMATION ALGORITHM (MATLAB)

```
% Successive Target Approximation (STA) Algorithm and Example Usage
% File: SuccessiveTargetApproximation.m
% Author: Pedro Tomé (tome.p.m at ua.pt)
%
%
% To be used under the terms of the GNU General Public License:
% http://www.gnu.org/copyleft/gpl.html
%
%
function SuccessiveTargetApproximation()
    %% Program setup
    maxIterations = 250;          % Target Approximation stop condition
    learningRate  = 0.5;          % Learning rate of the STA algorithm
    netHiddenSize = [10 10 10];   % Number of neurons per hidden layer
    maxNetPerformanceLoss_dB = 1; % Maximum performance loss allowed
                                  % when synthesizing the ANN

    maxInputAmplitude = 1;
    targetAmplifierGain = 1;


    close all;
    %% Define the input and target output signals
    [netInput_I, netInput_Q] = iqGrid(maxInputAmplitude, 0.1);
    netInput = [netInput_I' ; netInput_Q'];

    ampOutput_targetI = netInput_I * targetAmplifierGain;
    ampOutput_targetQ = netInput_Q * targetAmplifierGain;
    ampOutput_target = [ampOutput_targetI , ampOutput_targetQ];


    %% Create IQ monitoring figure
    handles = createMonitoringFigure(netInput, ampOutput_target, ...
                                     maxIterations);


    %% STA Algorithm
    netTarget = netInput;

    bestError = Inf;
    errorHistory = nan(1, maxIterations);
    iteration = 0;
```

```
43      while (iteration < maxIterations)
44          iteration = iteration + 1;
45
46          % Calculate target performance
47          [trained_ampOutput_I, trained_ampOutput_Q] = ...
48              PA(netTarget(1,:)', netTarget(2,:)');
49          ampOutput = [trained_ampOutput_I , trained_ampOutput_Q];
50
51          error = costFunction(ampOutput_target, ampOutput);
52          if (iteration == 1)
53              errorHistory(iteration) = error;
54          else
55              errorHistory(iteration) = bestError;
56          end
57
58
59          % Accept new target if there was a performance increase
60          if (error < bestError)
61              bestError = error;
62
63              % This is the learning trick!
64              netTarget = netTarget  +  ...
65                          learningRate * (ampOutput_target - ampOutput)';
66          end
67
68
69          % Update figures
70          if (mod(iteration,10) == 0)
71              set(handles.netOutput, 'XData', netTarget(1,:), ...
72                                     'YData', netTarget(2,:));
73              set(handles.ampOutput, 'XData', ampOutput(:,1), ...
74                                     'YData', ampOutput(:,2));
75              set(handles.error, 'XData', 1:maxIterations, ...
76                                 'YData', 10*log10(errorHistory));
77              drawnow();
78          end
79      end
80      fprintf('Target NMSE: %g dB\n', 10*log10(bestError));
81
82
83
84      %% Create the predistorting Artificial Neural Network
85      net = feedforwardnet(netHiddenSize);
86      %net.trainParam.showWindow = 0;
87      net = configure(net, 'inputs', netInput);
88      net = configure(net, 'outputs', netInput);
89
90      % Allow for a loss of 'maxNetPerformanceLoss_dB' dB in policy
91      % performance when synthesizing it as an artificial neural network
92      trained_error = Inf;
93      training_iterations = 0;
94      while (10*log10(trained_error) > 10*log10(bestError) + ...
95                                       maxNetPerformanceLoss_dB)
96          training_iterations = training_iterations + 1;
97          if (mod(training_iterations, 50) == 0)
98              net = init(net);  % Hack in case it hangs
99          end
```

```
100         net = train(net, netInput, netTarget);
101
102         netOutput = net(netInput);
103         [trained_ampOutput_I, trained_ampOutput_Q] = ...
104             PA(netOutput(1,:)', netOutput(2,:)');
105         trained_ampOutput = [trained_ampOutput_I trained_ampOutput_Q];
106
107         trained_error=costFunction(ampOutput_target,trained_ampOutput);
108     end
109     fprintf('Neural Network NMSE: %g dB\n', 10*log10(trained_error));
110     fprintf('Absolute values of ANN weights range from %g to %g\n', ...
111         min(abs(getwb(net))), max(abs(getwb(net))));
112
113
114
115     %% Round network weights
116     decimalPlaces = 3;  % Akin to an implementation with 1 mV precision
117     round_net = setwb(net, round(getwb(net), decimalPlaces));
118
119     round_netOutput = round_net(netInput);
120     [round_ampOutput_I, round_ampOutput_Q] = ...
121         PA(round_netOutput(1,:)', round_netOutput(2,:)');
122     round_ampOutput = [round_ampOutput_I , round_ampOutput_Q];
123
124     round_error = costFunction(ampOutput_target, round_ampOutput);
125     fprintf('Rounded Neural Network NMSE: %g dB',10*log10(round_error));
126     fprintf(' (weights rounded to %d decimal places)\n', decimalPlaces);
127 end
128
129
130 function [yI, yQ] = PA(xI, xQ)
131     [yI, yQ] = VirtualStaticPA(xI(:), xQ(:));
132
133     gain = 1;
134     yI = gain * yI;
135     yQ = gain * yQ;
136 end
137 function cost = costFunction(ampOutput_target, ampOutput)
138     I = ampOutput(:,1);
139     Q = ampOutput(:,2);
140     It = ampOutput_target(:,1);
141     Qt = ampOutput_target(:,2);
142
143     squareError = (I - It).^2 + (Q - Qt).^2;
144     meanError = (I - mean(It)).^2 + (Q - mean(Qt)).^2;
145     NMSE = sum(squareError) / sum(meanError);
146
147     cost = NMSE;
148 end
149
150 function [I, Q] = iqGrid(maxAmplitude, delta)
151     %% Create grid of (I,Q) points
152     [I, Q] = meshgrid([-1 : delta : 1], [-1 : delta : 1]);
153     I = I(:) * maxAmplitude;
154     Q = Q(:) * maxAmplitude;
155
156
```

```
157     %% Exclude points outside the maxAmplitude radius
158     indices = sqrt(I.^2 + Q.^2) < maxAmplitude;
159     I = I(indices);
160     Q = Q(indices);
161 end
162 
163 function handles = createMonitoringFigure(netInput, ...
164                                           ampOutput_target, maxIterations)
165     figure();
166     %% IQ Mapping
167     [defaultAmpOutput_I, defaultAmpOutput_Q] = ...
168         PA(netInput(1,:)', netInput(2,:)');
169     subplot(1, 4, [1:3]);
170 
171     handles.netOutput = plot(nan(size(netInput, 2),1), ...
172                              nan(size(netInput, 2),1), 'g.');
173     hold on;
174     plot(defaultAmpOutput_I, defaultAmpOutput_Q, 'k.');
175     plot(ampOutput_target(:,1), ampOutput_target(:,2), 'r.');
176     handles.ampOutput = plot(nan(size(netInput, 2),1), ...
177                              nan(size(netInput, 2),1), 'b+');
178     axis([-1 1 -1 1]);
179     xlabel('I Component'); ylabel('Q Component');
180     legend('ANN Output', 'Default PA Output', 'Target PA Output', ...
181            'Linearized PA Output', 'Location', 'SouthEast');
182 
183 
184     %% IQ Mapping Error
185     subplot(1, 4, 4);
186     handles.error = plot(nan, nan);
187     xlim([1 maxIterations]);
188     xlabel('Iteration'); ylabel('NMSE (dB)');
189 end
```