PII: S0893-6080(18)30317-4
DOI: https://doi.org/10.1016/j.neunet.2018.11.002
Reference: NN 4064

To appear in: *Neural Networks*

Received date : 23 November 2017
Revised date : 30 October 2018
Accepted date : 4 November 2018

Kafnets: kernel-based non-parametric activation functions for neural networks

Simone Scardapane[a,*], Steven Van Vaerenbergh[c], Simone Totaro[b], Aurelio Uncini[a]

[a]*Department of Information Engineering, Electronics and Telecommunications (DIET), Sapienza University of Rome, Via Eudossiana 18, 00184 Rome, Italy*
[b]*Department of Statistical Sciences, Sapienza University of Rome, Piazzale Aldo Moro 5, 00185 Rome, Italy.*
[c]*Department of Communications Engineering, University of Cantabria, Av. los Castros s/n, 39005 Santander, Cantabria, Spain.*

Abstract

Neural networks are generally built by interleaving (adaptable) linear layers with (fixed) nonlinear activation functions. To increase their flexibility, several authors have proposed methods for adapting the activation functions themselves, endowing them with varying degrees of flexibility. None of these approaches, however, have gained wide acceptance in practice, and research in this topic remains open. In this paper, we introduce a novel family of flexible activation functions that are based on an inexpensive kernel expansion at every neuron. Leveraging several properties of kernel-based models, we propose multiple variations for designing and initializing these kernel activation functions (KAFs), including a multidimensional scheme allowing to nonlinearly combine information from different paths in the network. The

[*]Corresponding author. Phone: +39 06 44585495, Fax: +39 06 4873300.
Email addresses: `simone.scardapane@uniroma1.it` (Simone Scardapane), `steven.vanvaerenbergh@unican.es` (Steven Van Vaerenbergh), `aurelio.uncini@uniroma1.it` (Aurelio Uncini)

resulting KAFs can approximate any mapping defined over a subset of the real line, either convex or non-convex. Furthermore, they are smooth over their entire domain, linear in their parameters, and they can be regularized using any known scheme, including the use of ℓ_1 penalties to enforce sparseness. To the best of our knowledge, no other known model satisfies all these properties simultaneously. In addition, we provide an overview on alternative techniques for adapting the activation functions, which is currently lacking in the literature. A large set of experiments validates our proposal.

Keywords: Neural networks, Activation functions, Kernel methods

1. Introduction

Neural networks (NNs) are powerful approximators, which are built by interleaving linear layers with nonlinear mappings (generally called *activation functions*). The latter step is usually implemented using an element-wise, (sub-)differentiable, and fixed nonlinear function at every neuron. In particular, the current consensus has shifted from the use of contractive mappings (e.g., sigmoids) to the use of piecewise-linear functions (e.g., rectified linear units, ReLUs (Glorot and Bengio, 2010)), allowing a more efficient flow of the backpropagated error (Goodfellow et al., 2016). This relatively inflexible architecture might help explaining the extreme redundancy found in the trained parameters of modern NNs (Denil et al., 2013).

Designing ways to adapt the activation functions themselves, however, faces several challenges. On one hand, we can parameterize a known activation function with a small number of trainable parameters, describing for example the slope of a particular linear segment (He et al., 2015). While immediate to implement, this only results in a small increase in flexibility and a marginal improvement in performance in the general case (Agostinelli et al., 2014). On the other hand, a more interesting task is to devise a scheme allowing each activation function to model a large range of shapes, such as any smooth function defined over a subset of the real line. In this case, the inclusion of one (or more) hyper-parameters enables the user a trade-off between greater flexibility and a larger number of parameters per-neuron. We refer to these schemes in general as *non-parametric* activation functions, since the number of (adaptable) parameters can potentially grow without bound.

There are three main classes of non-parametric activation functions known in the literature:[1] adaptive piecewise linear (APL) functions (Agostinelli et al., 2014), maxout networks (Goodfellow et al., 2013), and spline activation functions (Guarnieri et al., 1999). These are described in greater depth in Section 5. The success of these functions in several datasets points to an interesting aspect: in some cases, small networks with highly flexible nonlinear functions can achieve very similar performance to much deeper networks having only fixed functions, and with a lower number of parameters globally. As an example, Agostinelli et al. (2014) builds networks with a single hidden layer of APLs that perform as well as competing networks with 4 or more hidden layers. However, in Section 5 we also argue that (despite their success) none of these non-parametric approaches is completely satisfactory, in the sense that each of them loses one or more desirable properties, such

[1]In this paper we only consider activation functions that can be used as a 'drop-in' replacement of standard functions. There exists alternative proposals which require changing the optimization process or optimizing them in separate steps. We briefly mention these approaches in Section 7.

as smoothness of the resulting functions in the APL and maxout cases (see Table 1 in Section 6 for a schematic comparison).

In this paper, we propose a fourth class of non-parametric activation functions, which are based on a kernel representation of the function. In particular, we define each activation function as a linear superposition of several kernel evaluations, where the dictionary of the expansion is fixed beforehand by sampling the real line. As we show later on, the resulting kernel activation functions (KAFs) have a number of desirable properties, including: (i) they can be computed cheaply using vector-matrix operations; (ii) they are linear with respect to the trainable parameters; (iii) they are smooth over their entire domain; (iv) using the Gaussian kernel, their parameters only have *local* effects on the resulting shapes; (v) the parameters can be regularized using any classical approach, including the possibility of enforcing sparseness through the use of ℓ_1 norms. To the best of our knowledge, none of the known methods possess all these properties simultaneously. We call a NN endowed with KAFs at every neuron a *Kafnet*.

The idea proposed in this paper is related to other proposals that tried to connect the advantages of deep learning with kernel techniques and Gaussian processes (GPs). Most notably, Damianou and Lawrence (2013) designed a deep GP by stacking multiple GPs together in a similar fashion as a standard neural network, while Wilson et al. (2016) applied a GP on top of the features extracted from a deep network. Constructing deep GPs, however, is hindered by the computational intractability of the resulting optimization task. Several authors have tried to circumvent this problem by, e.g., applying random Fourier approximations to the underlying kernel (Cutajar et al., 2017). More in general, the vast majority of literature on deep GPs tries to replace the *entire* deep NN architecture with layered GPs. In this paper we consider a different setup instead, where kernel methods replace only the activation function neuron-wise (thus avoiding any computational bottleneck).

Importantly, by learning the activation functions through kernel methods, the proposed technique can still potentially leverage their extensive literature, either in statistics, machine learning (Hofmann et al., 2008), and signal processing (Liu et al., 2011). Here, we preliminarily exploit the connection by discussing several heuristics to choose the kernel hyper-parameter, along with techniques for initializing the trainable parameters. However, much more can be applied in this context, as we discuss more in depth in the conclusive section. We also propose a bi-dimensional variant of our KAF, allowing the information from multiple linear projections to be nonlinearly combined in an adaptive fashion.

In addition, we contend that the rareness of flexible activation functions in practice can be partially attributed to the lack of a cohesive (introductory) treatment on the topic. To this end, a further aim of this paper is to provide a relatively comprehensive overview on the selection of a proper activation function. In particular, we divide the discussion on the state-of-the-art in three separate sections. Section 3 introduces the most common (fixed) activation functions used in NNs, from the classical sigmoid function up to the recently developed self-normalizing unit (Klambauer et al., 2017) and Swish function (Ramachandran et al., 2017). Subsequently, we describe in Section 4 how most of these functions can be efficiently parameterized by one or more adaptive scalar values in order to enhance their flexibility.

Finally, we introduce the three existing models for designing non-parametric activation functions in Section 5. For each of them, we briefly discuss relative strengths and drawbacks, which serve as a motivation for introducing the model subsequently.

The rest of the paper is composed of four additional sections. Section 6 describes the proposed KAFs, together with several practical implementation guidelines regarding the selection of the dictionary and the choice of a proper initialization for the weights. For completeness, Section 7 briefly describes additional strategies to improve the activation functions, going beyond the addition of trainable parameters in the model. A large set of experiments is described in Section 8, and, finally, the main conclusions and a number of future lines of research are given in Section 9.

Notation

We denote vectors using boldface lowercase letters, e.g., $\mathbf{a}$; matrices are denoted by boldface uppercase letters, e.g., $\mathbf{A}$. All vectors are assumed to be column vectors. The operator $\left\|\cdot\right\|_p$ is the standard ℓ_p norm on a Euclidean space. For $p = 2$, it coincides with the Euclidean norm, while for $p = 1$ we obtain the Manhattan (or taxicab) norm defined for a generic vector $\mathbf{v} \in \mathbb{R}^B$ as $\left\|\mathbf{v}\right\|_1 = \sum_{k=1}^{B} |v_k|$. Additional notations are introduced along the paper when required.

2. Preliminaries

We consider training a standard feedforward NN, whose l-th layer is described by the following equation:

$$\mathbf{h}_l = g_l \left(\mathbf{W}_l \mathbf{h}_{l-1} + \mathbf{b}_l \right) \,, \tag{1}$$

where $\mathbf{h}_{l-1} \in \mathbb{R}^{N_{l-1}}$ is the N_{l-1}-dimensional input to the layer, $\mathbf{W}_l \in \mathbb{R}^{N_l \times N_{l-1}}$ and $\mathbf{b}_l \in \mathbb{R}^{N_l}$ are adaptable weight matrices, and $g_l(\cdot)$ is a nonlinear function, called *activation function*, which is applied element-wise. In a NN with L layers, $\mathbf{x} = \mathbf{h}_0$ denotes the input to the network, while $\hat{\mathbf{y}} = \mathbf{h}_L$ denotes the final output.

For training the network, we are provided with a set of I input/output pairs $\mathcal{S} = \{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^{I}$, and we minimize a regularized cost function given by:

$$J(\mathbf{w}) = \sum_{i=1}^{I} l(\mathbf{y}_i, \hat{\mathbf{y}}_i) + C \cdot r(\mathbf{w}) \,, \tag{2}$$

where $\mathbf{w} \in \mathbb{R}^Q$ collects all the trainable parameters of the network, $l(\cdot, \cdot)$ is a loss function (e.g., the squared loss), $r(\cdot)$ is used to regularize the weights using, e.g., ℓ_2 or ℓ_1 penalties, and the regularization factor $C > 0$ balances the two terms.

In the following, we review common choices for the selection of g_l, before describing methods to adapt them based on the training data. For readability, we will drop the subscript l, and use the letter s to denote a single input to the function, which we call an *activation*. Note that, in most cases, the activation function g_L for the last layer cannot be chosen freely, as it depends on the task and a proper scaling of the output. In particular, it is common to select $g(s) = s$ for regression problems, and a sigmoid function for binary problems with $y_i \in \{0, 1\}$:

$$g(s) = \delta(s) = \frac{1}{1 + \exp\{-s\}} \,. \tag{3}$$

For multi-class problems with dummy encodings on the output, the softmax function generalizes the sigmoid and it ensures valid probability distributions in the output (Bishop, 2006).

3. Fixed activation functions

We briefly review some common (fixed) activation functions for neural networks, that are the basis for the parametric ones in the next section. Before the current wave of deep learning, most activation functions used in NNs were of a 'squashing' type, i.e., they were monotonically non-decreasing functions satisfying:

$$\lim_{s \to -\infty} g(s) = c, \;\; \lim_{s \to \infty} g(s) = 1 \,, \tag{4}$$

where c can be either 0 or -1, depending on convention. Apart from the sigmoid in (3), another common choice is the hyperbolic tangent, defined as:

$$g(s) = \tanh(s) = \frac{\exp\{s\} - \exp\{-s\}}{\exp\{s\} + \exp\{-s\}} \,. \tag{5}$$

Cybenko (1989) proved the universal approximation property for this class of functions, and his results were later extended to a larger class of functions in

Hornik et al. (1989). In particular, as long as $g(\cdot)$ is non-constant, bounded, continuous and monotonically-increasing, a NN with one hidden layer of appropriate size and linear activation functions in output can approximate any continuous function defined on a compact input (Hornik et al., 1989). In practice, pure squashing functions were found of limited use in deep networks (where L is large), being prone to the problem of vanishing gradients, due to their bounded derivatives (Hochreiter et al., 2001).

A breakthrough in modern deep learning came from the introduction of the rectified linear unit (ReLU) function, defined as:

$$g(s) = \max\{0, s\} \ . \tag{6}$$

Despite being unbounded and introducing a point of non-differentiability, the ReLU has proven to be extremely effective for deep networks (Glorot and Bengio, 2010; Maas et al., 2013). Most notably, the use of ReLUs has allowed to train networks with multiple hidden layers without the need for a sophisticated pretraining stage as was done previously (Glorot et al., 2011). The ReLU has two main advantages. First, its gradient is either 0 or 1,[2] making back-propagation particularly efficient. Secondly, its activations are sparse, which is beneficial from several points of views. Interestingly, a NN with ReLU activations (which is also called a *rectifier* network) can be interpreted as an ensemble of an exponential number of linear models with shared weights (Glorot et al., 2011).

A smoothed version of the ReLU, called softplus, is also introduced in

[2]For $s = 0$, the function is not differentiable, but any value in $[0, 1]$ is a valid subgradient. Most implementations of ReLU use 0 as the default choice in this case.

Glorot et al. (2011):

$$g(s) = \log\left\{1 + \exp\left\{s\right\}\right\} \,. \tag{7}$$

Despite its lack of smoothness, ReLU functions are almost always preferred to the softplus in practice. One obvious problem of ReLUs is that, for a wrong initialization or an unfortunate weight update, its activation can get stuck in 0, irrespective of the input. This is referred to as the 'dying ReLU' condition. To circumvent this problem, Maas et al. (2013) introduced the leaky ReLU function, defined as:

$$g(s) = \begin{cases} s & \text{if } s \geq 0 \\ \alpha s & \text{otherwise} \end{cases} \,, \tag{8}$$

where the user-defined parameter $\alpha > 0$ is generally set to a small value, such as 0.01. While the resulting pattern of activations is not exactly sparse anymore, the parameters cannot get stuck in a poor region. (8) can also be written more compactly as $g(s) = \max\left\{0, s\right\} + \alpha \min\left\{0, s\right\}$.

Another problem of activation functions having only non-negative output values is that their mean value is always positive by definition. Motivated by an analogy with the natural gradient, Clevert et al. (2016) introduced the exponential linear unit (ELU) to renormalize the pattern of activations:

$$g(s) = \text{ELU}(s) \begin{cases} s & \text{if } s \geq 0 \\ \alpha\left(\exp\left\{s\right\} - 1\right) & \text{otherwise} \end{cases} \,, \tag{9}$$

where in this case α is generally chosen as 1. The ELU modifies the negative part of the ReLU with a function saturating at a user-defined value α. It is computationally efficient, smooth (unlike the ReLU and the leaky ReLU), and with a gradient which is either 1 or $g(s) + \alpha$ for negative values of s.

The recently introduced scaled ELU (SELU) generalizes the ELU to have further control over the range of activations (Klambauer et al., 2017):

$$g(s) = \mathrm{SELU}(s) = \lambda \cdot \mathrm{ELU}(s)\,, \tag{10}$$

where $\lambda > 1$ is a second user-defined parameter. In particular, it is shown in Klambauer et al. (2017) that for $\lambda \approx 1.6733$ and $\alpha \approx 1.0507$, the successive application of (1) converges towards a fixed distribution with zero mean and unit variance, leading to a self-normalizing network behavior. Interestingly, the SELU was the first activation function specifically tailored for building deep feedforward (not convolutional) networks. On a set of 121 datasets taken from the UCI repository, it was found to significantly outperform competing networks with standard activation functions and different types of regularization (Klambauer et al., 2017).

Finally, Swish (Ramachandran et al., 2017) is a recently proposed activation somewhat inspired by the gating steps in a standard LSTM recurrent cell:

$$g(s) = s \cdot \delta(s)\,, \tag{11}$$

where $\delta(s)$ is the sigmoid in (3). Swish is the only function we discuss which was not hand-crafted: instead, it was found using a search in the space of possible activation functions (Ramachandran et al., 2017).

4. Parametric adaptable activation functions

An immediate way to increase the flexibility of a NN is to parameterize one of the previously introduced activation functions with a *fixed* (small) number of adaptable parameters, such that each neuron can adapt its activation function to a different shape. As long as the function remains differentiable with respect to these new parameters, it is possible to adapt them with any numerical optimization algorithm together with the linear weights and biases of the layer. Due to their fixed number of parameters and limited flexibility, we call these *parametric* activation functions.

Historically, one of the first proposals in this sense was the generalized hyperbolic tangent (Chen and Chang, 1996), a tanh function parameterized by two additional positive scalar values a and b:

$$g(s) = \frac{a\left(1 - \exp\left\{-bs\right\}\right)}{1 + \exp\left\{-bs\right\}} \,. \tag{12}$$

Note that the parameters a, b are initialized randomly and are adapted independently for every neuron. Specifically, a determines the range of the output (which is called the amplitude of the function), while b controls the slope of the curve. Trentin (2001) provides empirical evidence that learning the amplitude for each neuron is beneficial (either in terms of generalization error, or speed of convergence) with respect to having unit amplitude for all activation functions. Similar results were also obtained for recurrent networks (Goh and Mandic, 2003).

More recently, He et al. (2015) consider a parametric version of the leaky ReLU in (8), where the coefficient α is initialized at $\alpha = 0.25$ everywhere and then adapted for every neuron. The resulting activation function is called parametric ReLU (PReLU), and it has a very simple derivative with respect to the new parameter:

$$\frac{\partial g(s)}{\partial \alpha} = \begin{cases} 0 & \text{if } s \geq 0 \\ s & \text{otherwise} \end{cases} . \tag{13}$$

For a layer with N hidden neurons, this introduces only N additional parameters, compared to $2N$ parameters for the generalized tanh. Importantly, in the case of ℓ_p regularization, the user has to be careful not to regularize the α parameters, which would bias the optimization process towards classical ReLU / leaky ReLU activation functions.

Similarly, Trottier et al. (2016) propose a modification of the ELU function in (9) with an additional scalar parameter β, called parametric ELU (PELU):

$$g(s) = \begin{cases} \frac{\alpha}{\beta} s & \text{if } s \geq 0 \\ \alpha \left(\exp\left\{ \frac{s}{\beta} \right\} - 1 \right) & \text{otherwise} \end{cases} , \tag{14}$$

where both α and β are initialized randomly and adapted during the training process. Based on the analysis in Trottier et al. (2016), there always exists a setting for the linear weights and α, β which avoids the vanishing gradient problem. Unlike the PReLU, however, the two parameters should be regularized in order to avoid a degenerate behavior with respect to the linear weights in (1), where extremely small linear weights are coupled with very large values for the parameters of the activation functions. Both PReLU and PELU are able to improve over their non-parametric counterparts if the structure of the network is kept constant (He et al., 2015; Trottier et al., 2016).

A more flexible proposal is the S-shaped ReLU (SReLU) (Jin et al., 2016), which is parameterized by four scalar values $\{t^r, a^r, t^l, a^l\}$:

$$g(s) = \begin{cases} t^r + a^r\left(s - t^r\right) & \text{if } s \geq t^r \\ s & \text{if } t^r > s > t^l \\ t^l + a^l\left(s - t^l\right) & \text{otherwise} \end{cases} . \tag{15}$$

The SReLU is composed by three linear segments, the middle of which is the identity. In contrast to the PReLU the cut-off points between the three segments can also be adapted. Additionally, the function can have both convex and non-convex shapes, depending on the orientation of the left and right segments, making it more flexible than previous proposals. Similar to the PReLU, the four parameters should not be regularized (Jin et al., 2016).

Finally, a parametric version of the Swish function is the β-swish (Ramachandran et al., 2017), which includes a tunable parameter β inside the sigmoid part of the activation function:

$$g(s) = s \cdot \delta(\beta s)\,. \tag{16}$$

5. Non-parametric activation functions

Intuitively, parametric activation functions have limited flexibility, resulting in mixed performance gains on average. As opposed to parametric approaches, non-parametric activation functions allow to model a larger class of shapes (in the best case, any continuous segment), at the price of a larger number of adaptable parameters. As stated in the introduction, these methods generally introduce a further global hyper-parameter allowing to balance the flexibility of the function, by varying the effective number of free parameters, which can potentially grow without bound. Additionally, the methods can be grouped depending on whether each parameter has a local or global effect on the overall function, the former being a desirable characteristic.

In this section, we describe three state-of-the-art approaches for implementing non-parametric activation functions: APL functions in Section 5.1, spline functions in Section 5.2, and maxout networks in Section 5.3.

5.1. Adaptive piecewise linear methods

An APL function, introduced in Agostinelli et al. (2014), generalizes the SReLU function in (15) by summing multiple linear segments, where all slopes and cut-off points are learned under the constraint that the overall function is continuous:

$$g(s) = \max\left\{0, s\right\} + \sum_{i=1}^{S} a_i \max\left\{0, -s + b_i\right\} \,. \tag{17}$$

S is a hyper-parameter chosen by the user, while each APL is parameterized by $2S$ adaptable parameters $\{a_i, b_i\}_{i=1}^{S}$. These parameters are randomly initialized for each neuron, and can be regularized with ℓ_2 regularization, similarly to the PELU, in order to avoid the coupling of very small linear weights and very large a_i coefficients for the APL units.

The APL unit cannot approximate any possible function. Its approximation properties are described in the following theorem.

Theorem 1 (Agostinelli et al. 2014)**.** *The APL unit can approximate any continuous piecewise-linear function $h(s)$, for some choice of S and $\{a_i, b_i\}_{i=1}^S$, provided that $h(s)$ satisfies the following two conditions:*

1. *There exists $u \in \mathbb{R}$ such that $h(s) = s, \ \forall s \geq u$.*
2. *There exist two scalars, $v, t \in \mathbb{R}$ such that $\frac{dh(s)}{s} = t, \ \forall s < v$.*

The previous theorem implies that any piecewise-linear function can be approximated, provided that its behavior is linear for very large, or small, s. A possible drawback of the APL activation function is that it introduces $S+1$ points of non-differentiability for each neuron, which may damage the optimization algorithm. The next class of functions solves this problem, at the cost of a possibly larger number of parameters.

5.2. Spline activation functions

An immediate way to exploit polynomial interpolation in NNs is to build the activation function over powers of the activation s (Piazza et al., 1992):

$$g(s) = \sum_{i=0}^{P} a_i s^i \,, \tag{18}$$

where P is a hyper-parameter and we adapt the $(P+1)$ coefficients $\{a_i\}_{i=0}^P$. Since a polynomial of degree P can pass exactly through $P+1$ points, this polynomial activation function (PAF) can in theory approximate any smooth function. The drawback of this approach is that each parameter a_i has a global influence on the overall shape, and the output of the function can easily grow too large or encounter numerical problems, particularly for large absolute values of s and large P.

An improved way to use polynomial expansions is spline interpolation, giving rise to the spline activation function (SAF). The SAF was originally studied in Vecci et al. (1998); Guarnieri et al. (1999), and later re-introduced in a more modern context in Scardapane et al. (2016), following previous advances in nonlinear filtering (Scarpiniti et al., 2013). In the sequel, we adopt the newer formulation.

A SAF is described by a vector of T parameters, called *knots*, corresponding to a sampling of its y-values over an equispaced grid of T points over the x-axis, that are symmetrically chosen around the origin with sampling step Δx. For any other value of s, the output value of the SAF is computed with spline interpolation over the closest knot and its P rightmost neighbors, where P is generally chosen equal to 3, giving rise to a cubic spline interpolation scheme. Specifically, denote by k the index of the closest knot, and by $\mathbf{q}_k$ the vector comprising the corresponding knot and its P neighbors. We call this vector the *span*. We also define a new value

$$u = \frac{s}{\Delta x} - \left\lfloor \frac{s}{\Delta x} \right\rfloor , \tag{19}$$

where Δx is the user-defined sampling step. u defines a normalized abscissa value between the k-th knot and the $(k+1)$-th one. The output of the SAF is then given by (Scarpiniti et al., 2013):

$$g(s) = \mathbf{u}^T \mathbf{B} \mathbf{q}_k \,, \tag{20}$$

where the vector $\mathbf{u}$ collects powers of u up to the order P:

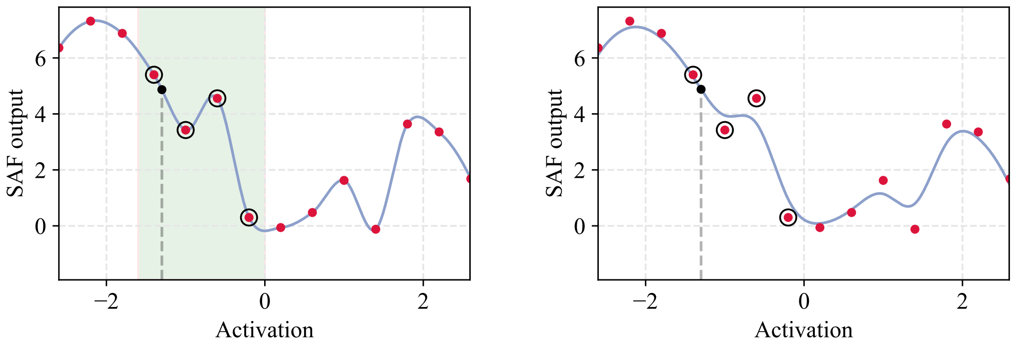
$$\mathbf{u} = \left[u^P, u^{P-1}, \ldots, u^1, 1\right]^T , \tag{21}$$

and $\mathbf{B}$ is the spline basis matrix, which defines the properties of the interpolation scheme (as shown later in Fig. 1b). For example, the popular Catmull-Rom basis for $P = 3$ is given by:

$$\mathbf{B} = \frac{1}{2}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} . \tag{22}$$

The derivatives of the SAF can be computed in a similar way, both with respect to s and with respect to $\mathbf{q}_k$, e.g., see Scardapane et al. (2016). A visual example of the SAF output is given in Fig. 1.

Each knot has only a limited local influence over the output, making their adaptation more stable. The resulting function is also smooth, and can in fact approximate any smooth function defined over a subset of the real line to a desired level of accuracy, provided that Δx is chosen small enough. The drawback is that regularizing the resulting activation functions is harder to achieve, since ℓ_p regularization cannot be applied directly to the values of the knots. In Guarnieri et al. (1999), this was solved by choosing a large Δx, in turn severely limiting the flexibility of the interpolation scheme. A different proposal was made in Scardapane et al. (2016), where the vector $\mathbf{q} \in \mathbb{R}^T$ of SAF parameters is regularized by penalizing deviations from the values at



(a) CR matrix (b) B-basis matrix

Figure 1: Example of output interpolation using a SAF neuron. Knots are shown with red markers, while the overall function is given in a light blue. (a) For a given activation, in black, only the control points in the green background are active. (b) We use the same control points as before, but we interpolate using the B-basis matrix (Scarpiniti et al., 2013) instead of the CR matrix in (22). The resulting curve is smoother, but it is not guaranteed to pass through all the control points.

initialization. Note that it is straightforward to initialize the SAF as any of the known fixed activation functions described before.

5.3. Maxout networks

Unlike the other functions described up to now, the maxout function introduced in Goodfellow et al. (2013) replaces an entire layer in (1). In particular, for each neuron, instead of computing a single dot product $\mathbf{w}^T\mathbf{h}$ to obtain the activation (where $\mathbf{h}$ is the input to the layer), we compute K different products with K separate weight vectors $\mathbf{w}_1, \ldots, \mathbf{w}_K$ and biases $b_1, \ldots, b_K$, and take their maximum:

$$g(\mathbf{h}) = \max_{i=1,\ldots,K} \left\{ \mathbf{w}_i^T\mathbf{h} + b_i \right\} \,, \tag{23}$$

where the activation function is now a function of a subset of the output of the previous layer. A NN having maxout neurons in all hidden layers is called a maxout network, and remains a universal approximator according to the following theorem.

Theorem 2 (Goodfellow et al. 2013)**.** *Any continuous function $h(\cdot) : \mathbb{R}^{N_0} \rightarrow \mathbb{R}^{N_L}$ can be approximated arbitrarily well on a compact domain by a maxout network with two maxout hidden units, provided K is chosen sufficiently large.*

The advantage of the maxout function is that it is extremely easy to implement using current linear algebra libraries. However, the resulting functions have several points of non-differentiability, similarly to the APL units. In addition, the number of resulting parameters is generally higher than with alternative formulations. In particular, by increasing K we multiply the original number of parameters by a corresponding factor, while other approaches contribute only linearly to this number. Additionally, we lose the possibility of plotting the resulting activation functions, unless the input to the maxout layer has less than 4 dimensions. An example with dimension 1 is shown in Fig. 2.

In order to solve the smoothness problem, (Zhang et al., 2014) introduced two smooth versions of the maxout neuron. The first one is the soft-maxout:

$$g(\mathbf{h}) = \log\left\{\sum_{i=1}^{K} \exp\left\{\mathbf{w}_i^T\mathbf{h} + b_i\right\}\right\} . \tag{24}$$

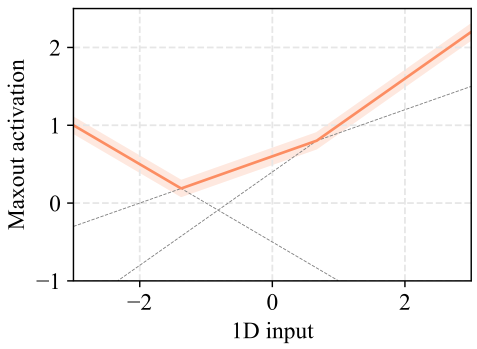


Figure 2: An example of a maxout neuron with a one-dimensional input and $K = 3$. The three linear segments are shown with a light gray, while the resulting activation is shown with a shaded red. Note how the maxout can only generate convex shapes by definition. However, plots cannot be made for inputs having more than three dimensions.

The second one is the ℓ_p-maxout, for a user-defined natural number p:

$$g(\mathbf{h}) = \sqrt[p]{\sum_{i=1}^{K} |\mathbf{w}_i^T \mathbf{h} + b_i|^p} \,. \tag{25}$$

Closely related to the ℓ_p-maxout neuron is the L_p unit proposed in Gulcehre et al. (2013). Denoting for simplicity $s_i = \mathbf{w}_i^T \mathbf{h} + b_i$, the L_p unit is defined as:

$$g(\mathbf{h}) = \left(\frac{1}{K} \sum_{i=1}^{K} |s_i - c_i|^p \right)^{\frac{1}{p}} \,, \tag{26}$$

where the $K+1$ parameters $\{c_1, \ldots, c_K, p\}$ are all learned via back-propagation.[3] If we fix $c_i = 0$, for p going to infinity the L_p unit degenerates to a special

[3] In practice, p is re-parameterized as $1+\log\{1+\exp\{p\}\}$ to guarantee that (26) defines a proper norm.

case of the maxout neuron:

$$\lim_{p \to \infty} g(\mathbf{h}) = \max_{i=1,\ldots,K} \{|s_i|\} \ . \tag{27}$$

6. Proposed kernel-based activation functions

In this section we describe the proposed KAF. Specifically, we model each activation function in terms of a kernel expansion over D terms as:

$$g(s) = \sum_{i=1}^{D} \alpha_i \kappa\left(s, d_i\right) \ , \tag{28}$$

where $\{\alpha_i\}_{i=1}^D$ are the mixing coefficients, $\{d_i\}_{i=1}^D$ are called the dictionary elements, and $\kappa(\cdot,\cdot) : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a 1D kernel function (Hofmann et al., 2008). In kernel methods, the dictionary elements are generally selected from the training data. In a stochastic optimization setting, this means that D would grow linearly with the number of training iterations, unless some proper strategy for the selection of the dictionary is implemented (Liu et al., 2011; Van Vaerenbergh et al., 2012). To simplify our treatment, we consider a simplified case where the dictionary elements are fixed, and we only adapt the mixing coefficients. In particular, we sample D values over the x-axis, uniformly around zero, similar to the SAF method, and we leave D as a user-defined hyper-parameter. This has the additional benefit that the resulting model is linear in its adaptable parameters, and can be efficiently implemented for a mini-batch of training data using highly-vectorized linear algebra routines. Note that there is a vast literature on kernel methods with fixed dictionary elements, particularly in the field of Gaussian processes

(Snelson and Ghahramani, 2006).

The kernel function $\kappa(\cdot,\cdot)$ only needs to respect the positive semi-definiteness property, i.e., for any possible choice of $\{\alpha_i\}_{i=1}^D$ and $\{d_i\}_{i=1}^D$ we have that:

$$\sum_{i=1}^{D}\sum_{j=1}^{D}\alpha_i\alpha_j\kappa\left(d_i,d_j\right)\geq 0\,. \tag{29}$$

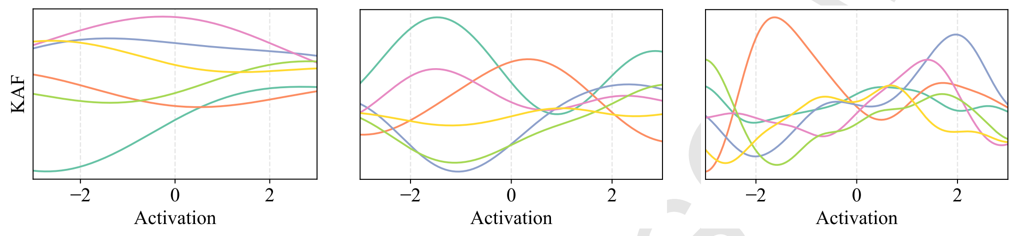
For our experiments, we use the 1D Gaussian kernel defined as:

$$\kappa(s,d_i)=\exp\left\{-\gamma\left(s-d_i\right)^2\right\}\ , \tag{30}$$

where $\gamma\in\mathbb{R}$ is called the kernel bandwidth, and its selection is discussed more at length below. Other choices, such as the polynomial kernel with $p\in\mathbb{N}$, are also possible:

$$\kappa(s,d_i)=\left(1+sd_i\right)^p\ . \tag{31}$$

By the properties of kernel methods, KAFs are equivalent to learning linear functions over a large number of nonlinear transformations of the original activation s, without having to explicitly compute such transformations. The Gaussian kernel has an additional benefit: the mixing coefficients have only a local effect over the shape of the output function (where the radius depends on γ, see below), which is advantageous during optimization. In addition, the expression in (28) with the Gaussian kernel can approximate any continuous function over a subset of the real line (Micchelli et al., 2006). The expression resembles a one-dimensional radial basis function network, whose universal approximation properties are also well studied (Park and Sandberg, 1991).



(a) $\gamma = 2.0$ (b) $\gamma = 0.5$ (c) $\gamma = 0.1$

Figure 3: Examples of KAFs. In all cases we sample uniformly 20 points on the x-axis, while the mixing coefficients are sampled from a normal distribution. The three plots show three different choices for γ.

Below we provide some additional considerations for implementing our KAF model. Note that the model has very simple derivatives for back-propagation:

$$\frac{\partial g(s)}{\partial \alpha_i} = \kappa\left(s, d_i\right) \,, \tag{32}$$

$$\frac{\partial g(s)}{\partial s} = \sum_{i=1}^{D} \alpha_i \frac{\partial \kappa\left(s, d_i\right)}{\partial s} \,. \tag{33}$$

On the selection of the kernel bandwidth

Selecting γ is crucial for the well-behavedness of the method, by acting indirectly on the effective number of adaptable parameters. In Fig. 3 we show some examples of functions obtained by fixing $D = 20$, randomly sampling the mixing coefficients, and only varying the kernel bandwidth, showing how γ acts on the smoothness of the overall functions.

In the literature, many methods have been proposed to select the bandwidth parameter for performing kernel density estimation (Jones et al., 1996). These methods include popular rules of thumb such as Scott (2015) or Silverman (1986).

In the problem of kernel density estimation, the abscissa corresponds to a given dataset with an arbitrary distribution. In the proposed KAF scheme, the abscissa are chosen according to a grid, and as such the optimal bandwidth parameter depends uniquely on the grid resolution. Instead of leaving the bandwidth parameter γ as an additional hyper-parameter, we have empirically verified that the following rule of thumb represents a good compromise between smoothness (to allow an accurate approximation of several initialization functions) and flexibility:

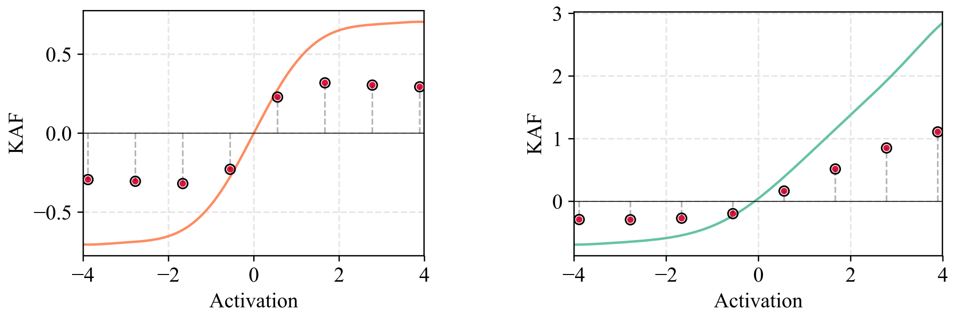
$$\gamma = \frac{1}{6\Delta^2}\,, \tag{34}$$

where Δ is the distance between the grid points. We also performed some experiments in which γ was adapted through back-propagation, though this did not provide any gain in accuracy.

On the initialization of the mixing coefficients

A random initialization of the mixing coefficients from a normal distribution, as in Fig. 3, provides good diversity for the optimization process. Nonetheless, a further advantage of our scheme is that we can initialize some (or all) of the KAFs to follow any know activation function, so as to guarantee a certain desired behavior. Specifically, denote by $\mathbf{t} = [t_1, \ldots, t_D]^T$ the vector of desired initial KAF values corresponding to the dictionary elements $\mathbf{d} = [d_1, \ldots, d_D]^T$. We can initialize the mixing coefficients $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_D]^T$ using kernel ridge regression:

$$\boldsymbol{\alpha} = \left(\mathbf{K} + \varepsilon\mathbf{I}\right)^{-1}\mathbf{t}\,, \tag{35}$$



(a) tanh (b) ELU

Figure 4: Two examples of initializing a KAF using (35), with $\varepsilon = 10^{-6}$. (a) A hyperbolic tangent. (b) The ELU in (9). The red dots indicate the corresponding initialized values for the mixing coefficients.

where $\mathbf{K} \in \mathbb{R}^{D \times D}$ is the kernel matrix computed between $\mathbf{t}$ and $\mathbf{d}$, and we add a diagonal term with $\varepsilon > 0$ to avoid degenerate solutions with very large mixing coefficients. Two examples are shown in Fig. 4.

Multi-dimensional kernel activation functions

In our experiments, we also consider a two-dimensional variant of the proposed KAF, that we denote as 2D-KAF. Roughly speaking, the 2D-KAF acts on a pair of activation values, instead of a single one, and learns a two-dimensional function to combine them. It can be seen as a generalization of a two-dimensional maxout neuron, which is instead constrained to output the maximum value among the two inputs.

Following the methodology introduced earlier, we construct a dictionary $\mathbf{d} \in \mathbb{R}^{D^2 \times 2}$ by sampling a uniform grid over the 2D plane, by considering D positions uniformly spaced around 0 in both dimensions. We group the incoming activation values in pairs (assuming that the layer has even size), and for each possible pair of activations $\mathbf{s} = [s_k, s_{k+1}]^T$ we output:

$$g(\mathbf{s}) = \sum_{i=1}^{D^2} \alpha_i \kappa(\mathbf{s}, \mathbf{d}_i) \ , \tag{36}$$

where $\mathbf{d}_i$ is the i-th element of the dictionary, and we now have D^2 adaptable coefficients $\{\alpha_i\}_{i=1}^{D^2}$. In this case, we consider the 2D Gaussian kernel:

$$\kappa(\mathbf{s}, \mathbf{d}_i) = \exp\left\{-\gamma \left\|\mathbf{s} - \mathbf{d}_i\right\|_2^2\right\} \ , \tag{37}$$

where we use the same rule of thumb in (34), multiplied by $\sqrt{2}$, to select γ. The increase in parameters is counter-balanced by two factors. Firstly, by grouping the activations we halve the size of the linear matrix in the subsequent layer. Secondly, we generally choose a smaller D with respect to the $1D$ case, i.e., we have found that values in $[5, 10]$ are enough to provide a good degree of flexibility. Table 1 provides a comparison of the two proposed KAF models to the three alternative non-parametric activation functions described before. We briefly mention here that a multidimensional variant of the SAF was explored in Solazzi and Uncini (2000).

7. Additional topics on the design of activation functions

Before moving to the experimental section, for completeness we review here (briefly) some works that have also considered the design of activation functions, but which cannot be trivially used as a 'drop-in' replacement of standard functions: in particular, these are methods which do not necessarily require to adapt the shape of the functions via numerical optimization,

Name	**Smooth**	**Locality**	**Can use regularization**	**Plottable**	**Hyper-parameter**	**Trainable weights**
APL	No	Partially	Only ℓ_2 regularization	Yes	Number of segments S	$N_{i-1}N_i + N_i + 2SN_i$
SAF	Yes	Yes	No	Yes	Number of control points Q	$N_{i-1}N_i + N_i + QN_i$
Maxout	No	No	Yes	No*	Number of affine maps K	$KN_{i-1}N_i + KN_i$
Proposed KAF	Yes	Yes**	Yes	Yes	Size of the dictionary D	$N_{i-1}N_i + N_i + DN_i$
Proposed 2D-KAF	Yes	Yes**	Yes	Yes	Size of the dictionary D	$N_{i-1}N_i + \frac{(N_i + D^2 N_i)}{2}$

* Maxout functions can only be plotted whenever $N_{i-1} \leq 3$.
** Only when using the Gaussian (or similar) kernel function.

Table 1: A comparison of the existing non-parametric activation functions and the proposed KAF and 2D-KAF. In our definition, an activation function is local if each adaptable weight only affects a small portion of the output values.

or that require special care when implemented (e.g., separate stages of optimization).

Randomized / noisy activation functions

To solve the dying ReLU problem, the randomized leaky ReLU (Xu et al., 2015) considers a leaky ReLU like in (8), in which during training the parameter α is randomly sampled at every step in a uniform distribution over $\mathcal{U}(l, u)$. To compensate with the stochasticity in training, in the test phase α is set equal to:

$$\alpha = \frac{l + u}{2} \,, \tag{38}$$

which is equivalent to taking the average of all possible values seen during training (this is similar to the dropout technique (Srivastava et al., 2014), which randomly deactivates some neurons during each step of training, and later rescales the weights during the test phase).

More in general, several papers have developed stochastic versions of the classical artificial neurons, whose output depends on one or more random variables sampled during their execution (Bengio et al., 2013), under the idea that the resulting noise can help guide the optimization process towards better minima. As a representative example, the noisy activation functions proposed in Gulcehre et al. (2016) achieve this by combining activation functions with 'hard saturating' regimes (i.e., their value is exactly zero outside a limited range) with random noise over the outputs, whose variance increases in the regime where the function saturates to avoid problems due to the sparse gradient terms.

Vector-valued activation functions

Another approach is to design *vector-valued* activation functions to maximize parameter sharing. In the simplest case, the concatenated ReLU (Shang et al., 2016) returns two output values by applying a ReLU function both on s and on $-s$. Similarly, the order statistic network (Rennie et al., 2014) modifies a maxout neuron by returning the input activations in sorted order, instead of picking the highest value only. Multi-bias activation functions (Li et al., 2016) compute several activations values by using different bias terms, and then apply the same activation function independently over each of the resulting values.

Additional non-parametric models

The network-in-network (Lin et al., 2014) model is a non-parametric approach specific to convolutional neural networks, wherein the nonlinear parts of the filter are replaced with a fully connected NN acting on all channels simultaneously.

For specific tasks of audio modeling, some authors have proposed the use of Hermite polynomials for adapting the activation functions (Siniscalchi et al., 2013; Siniscalchi and Salerno, 2017). Similarly to our proposed KAF, the functions are expressed as a weighted sum of several fixed nonlinear transformations of the activation values, i.e., the Hermite polynomials. However, the nonlinear transformations are computed through the use of a recurrence formula, thus highly increasing the computational load.

Another non-parametric model that we have not discussed in the main text is the use of a Fourier basis expansion, originally proposed in Eisenach et al. (2017), which provided substantial improvements on several image classification problems. However, as stated in (Eisenach et al., 2017), training of the Fourier coefficients together with the linear weights is found to be unstable, and the authors resort to a two-stage optimization process where the nonlinear coefficients are initially kept fixed.

8. Experimental results

In this section we provide a comprehensive evaluation of the proposed KAFs and 2D-KAFs when applied to several use cases. As a preliminary experiment, we begin by comparing multiple activation functions on a relatively small classification dataset (Sensorless) in Section 8.2, where we discuss several examples of the shapes that are generally obtained by the networks and initialization strategies. We then consider a large-scale dataset taken from Baldi et al. (2014) in Section 8.4, where we show that two layers of KAFs are able to significantly outperform a feedforward network with five hidden layers, even when considering parametric activation functions and state-of-the-art regularization techniques. In Section 8.5 we show that KAFs and 2D-KAF provide an increase in performance also when applied to convolutional layers on the MNIST, Fashion MNIST, CIFAR-10, and CIFAR-100 datasets. Finally, we show in Section 8.6 that they have significantly faster training and higher cumulative reward for a set of reinforcement learning scenarios using MuJoCo environments from the OpenAI Gym.[4] We provide an open-source library to replicate the experiments, with the implementa-

[4]https://gym.openai.com/

tion of KAFs and 2D-KAFs in three separate frameworks, i.e., AutoGrad,[5] TensorFlow,[6] and PyTorch,[7] which is publicly accessible on the web.[8]

8.1. Experimental setup

Unless noted otherwise, in all experiments we linearly preprocess the input features between -1 and +1, and we substitute eventual missing values with the median values computed from the corresponding feature columns. From the full dataset, we randomly keep a portion of the dataset for validation and another portion for test, unless the dataset comes with its own split. All neural networks use a softmax activation function in their output layer, and they are trained by minimizing the average cross-entropy on the training dataset, to which we add a small ℓ_2-regularization term whose weight is selected in accordance to the literature. For optimization, we use the Adam algorithm (Kingma and Ba, 2015) with mini-batches of 100 elements and default hyper-parameters. For each epoch we compute the accuracy over the validation set, and we stop training whenever the validation accuracy is not improving for 15 consecutive epochs. Experiments are performed using the PyTorch implementation on a machine with an Intel Xeon E5-2620 CPU, with 16 GB of RAM and a CUDA back-end employing an Nvidia Tesla K20c. All accuracy measures over the test set are computed by repeating the experiments for 5 different splits of the dataset (unless the splits are provided by the dataset itself) and initializations of the networks. Weights of the

[5]https://github.com/HIPS/autograd
[6]https://www.tensorflow.org/
[7]http://pytorch.org/
[8]https://github.com/ispamm/kernel-activation-functions/

linear layers are always initialized using the so-called ‘Uniform He’ strategy, while additional parameters introduced by parametric and non-parametric activation functions are initialized by following the guidelines of the original papers.

8.2. Visualizing the activation functions

We begin with an experiment on the ‘Sensorless’ dataset to investigate whether KAFs and 2D-KAFs can indeed provide improvements in accuracy with respect to other baselines, and for visualizing some of the common shapes that are obtained after training. The Sensorless dataset is a standard benchmark for supervised techniques, composed of 58509 examples with 49 input features representing electric signals, that are used to predict one among 11 different classes representing operating conditions. We partition it using a random 15% for validation and another 15% for testing, and we use a small regularization factor of 10^{-4}.

In this dataset, we found that the best performing fixed activation function is a simple hyperbolic tangent. In particular, a network with one hidden layer of 100 neurons achieves a test accuracy of 97.75%, while the best result is obtained with a network of three hidden layers (each composed of 100 neurons), which achieves a test accuracy of 99.18%. Due to the simplicity of the dataset, we have not found improvements here by adding more layers or including dropout during training as in the following sections. The best performing parametric activation function is instead the PReLU, that improves the results by obtaining a 98.48% accuracy with a single hidden layer, and 99.30% with three hidden layers.

For comparison, we train several feedforward networks with KAFs in the

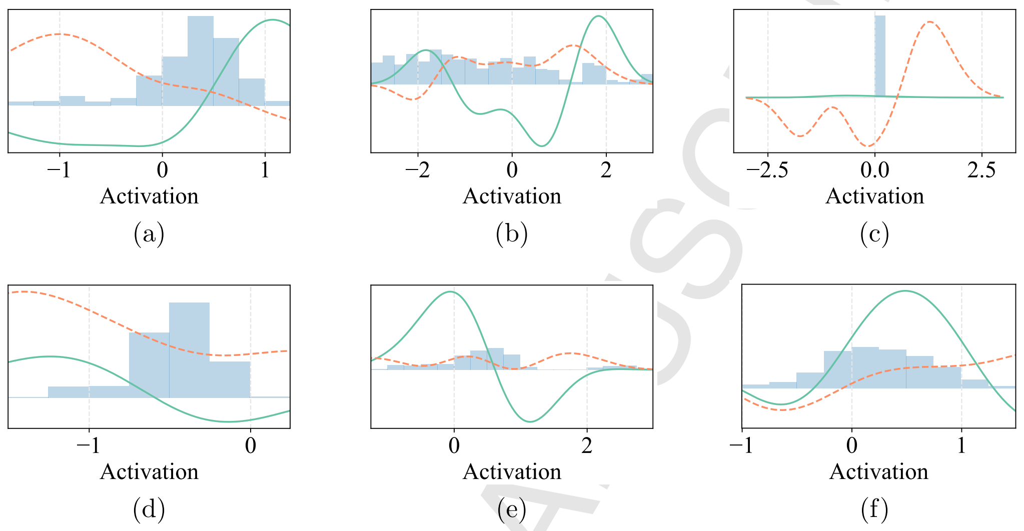


Figure 5: Examples of 6 trained KAFs (with random initialization) on the Sensorless dataset. On the y-axis we plot the output value of the KAF. The KAF after initialization is shown with a dashed red, while the final KAF is shown with a solid green. The distribution of activation values after training is shown as a reference with a light blue.

hidden layers, having a dictionary of $D = 20$ elements equispaced between -3.0 and 3.0, and initializing the linear coefficients from a normal distribution with mean 0 and variance 0.3. Using this setup, we already outperform all other baselines obtaining an accuracy of 99.04% with a single hidden layer, which improves to 99.80% when considering two hidden layers of KAFs.

Although the dataset is relatively simple, the shapes we obtain are representative of all the experiments we performed, and we provide a selection in Fig. 5. Specifically, the initialization of the KAF is shown with a dashed red line, while the final KAF is shown with a solid green line. For understanding the behavior of the functions, we also plot the empirical distribution of the activations on the test set using a light blue in the background. Some shapes are similar to common activation functions discussed in Section 3, although they are shifted on the x-axis to correspond to the distribution of activation values. For example Fig. 5a is similar to an ELU, while Fig. 5d is similar to a standard saturating function. Also, while in the latter case the final shape is somewhat determined by the initialization, the final shapes in general tend to be independent from initialization, as in the case of Fig. 5a. Another common shape is that of a radial-basis function, as in Fig. 5f, which is similar to a Gaussian function centered on the mean of the empirical distribution. Shapes, however, can be vastly more complex than these. For example, in Fig. 5e we show a function which acts as a standard saturating function on the main part of the activations' distribution, while its right-tail tends to remove values larger than a given threshold, effectively acting as a sort of implicit regularizer. In Fig. 5b we show a KAF without an intuitive shape, that selectively amplify (either positively or negatively) multiple regions of its activation space. Finally, in Fig. 5c we show an interesting pruning effect, where useless neurons correspond to activation functions that are practically zero everywhere. This, combined with the possibility of applying ℓ_1-regularization (Scardapane et al., 2017), allows to obtain networks with a significant smaller number of effective parameters.

Interestingly, the shapes obtained in Fig. 5 seem to be necessary for the high performance of the networks, and they are not an artifact of initialization. Specifically, we obtain similar accuracies (and similar output functions) even when initializing all KAFs as close as possible to hyperbolic tangents, following the method described in Section 6, while we obtain a vastly inferior performance (in some cases even worse than the baseline), if we initialize the KAFs randomly and we prevent their adaptation. This (informally) points to

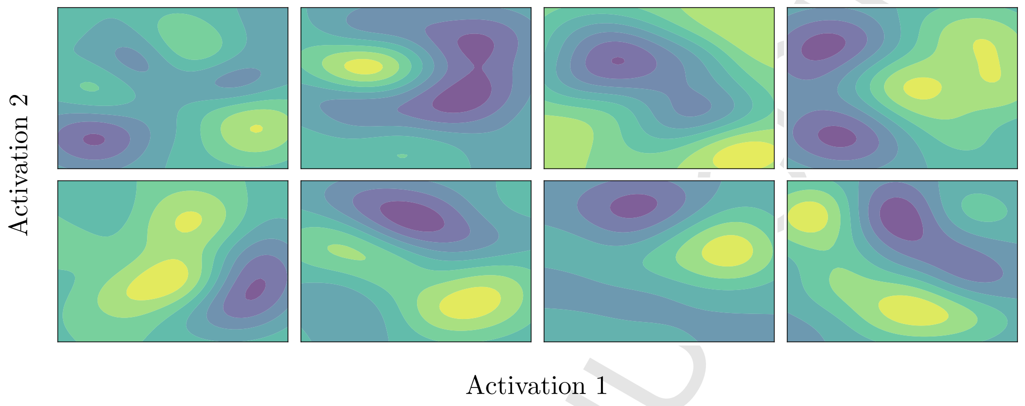


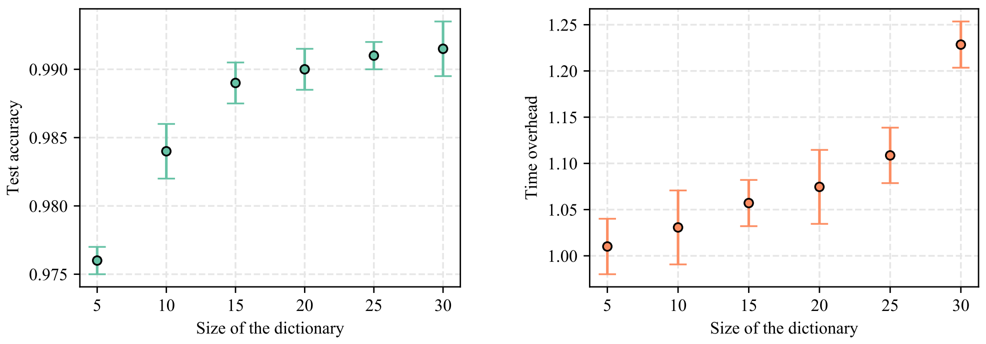
Figure 6: Examples of 8 trained 2D-KAFs on the Sensorless dataset. On the x- and y-axis we plot the two activation values (ranging from -3 to 3), while we show the output of the KAF using a heat map, where darker colors represent larger output values and lighter colors represent lower activation values, respectively.

the fact that their flexibility and adaptability seems to be an intrinsic component of their good performance in this experiment and in the following sections, an aspect that we will return to in our conclusive section.

Results are also similar when using 2D-KAFs, that we initialize with $D = 10$ elements on each axis using the same strategy as for KAFs. In this scenario, they obtain a test accuracy of 98.90% with a single hidden layer, and an accuracy of 99.84% (thus improving over the KAF) for two hidden layers. Some examples of obtained shapes are provided in Fig. 6.

8.3. On the choice of the dictionary size

In our proposed KAF implementation, the only hyper-parameter to be chosen by the user is the size of the dictionary D. While we have found $D = 20$ to be a good trade-off in general, it is interesting to consider the behavior of the method when this hyper-parameter is selected differently. To this end, we experiment again with the Sensorless dataset, using the same



(a) (b)

Figure 7: Example of varying the dictionary size D on the Sensorless dataset. (a) Final testing accuracy of the method. (b) Time required for each epoch, relative to an implementation with fixed activation functions (ReLUs).

setup as before (with a single hidden layer), but we vary the dictionary size between 5 and 30 elements. The results are plotted in Fig. 7.

We can see in Fig. 7a that, while the accuracy steadily decreases when considering small dictionary sizes (as the resulting activation functions are too ‘flat’ to represent meaningful shapes), the performance is relatively stable as soon as $D > 10$. At the same time, we plot in Fig. 7b the relative overhead (in terms of computational time) required by the method with respect to a standard NN with fixed activation functions (tanh). It can be seen how the overhead is extremely modest in most cases (and it becomes even smaller when considering deeper networks, as described in the following sections), although it quickly grows when the dictionary exceeds a certain threshold.

8.4. Comparisons on the SUSY benchmark

In this section, we evaluate the algorithms on a realistic large-scale use case, the SUSY benchmark introduced in Baldi et al. (2014). The task is to predict the presence of super symmetric particles on a simulation of a collision experiment, starting from 18 features (both low-level and high-level) describing the simulation itself. The overall dataset is composed of five million examples, of which the last 500000 are used for test, and another 500000 for validation. The task is interesting for several reasons. Due to the nature of the data, even a tiny change in accuracy (measured in terms of area under the curve, AUC) is generally statistically significant. In the original paper, Baldi et al. (2014) showed the best AUC was obtained by a deep feedforward network having five hidden layers, with significantly better results when compared to a shallow network. Surprisingly, Agostinelli et al. (2014) later showed that a shallow network is in fact sufficient, so long as it uses non-parametric activation functions (in that case, APL units).

In order to replicate these results with our proposed methods, we consider a baseline network inspired by Baldi et al. (2014), having five hidden layers with 300 neurons each and ReLU activation functions, with dropout applied to the last two hidden layer with probability 0.5. For comparison, we also consider the same architecture, but we substitute ReLUs with ELU, SELU, and PReLU functions. For SELU, we also substitute the standard dropout with the customized version proposed in Klambauer et al. (2017). We compare with simpler networks composed of one or two hidden layers of 300 neurons, employing Maxout neurons (with $K = 5$), APL units (with $S = 3$ as proposed in Agostinelli et al. (2014)), and the proposed KAFs and 2D-KAFs, following the same random initializations as the previous section. Results, in terms of AUC and amount of trainable parameters, are given in Table 2.

Activation function	**Testing AUC**	**Trainable parameters**
ReLU	0.8739(0.001)	367201
ELU	0.8739(0.001)	
SELU	0.8745(0.002)	
PReLU	0.8748(0.001)	368701
Maxout (one layer)	0.8744(0.001)	17401
Maxout (two layers)	0.8744(0.002)	288301
APL (one layer)	0.8744(0.002)	7801
APL (two layers)	0.8757(0.002)	99901
KAF (one layer)	0.8756(0.001)	12001
KAF (two layers)	<u>0.8758(0.001)</u>	108301
2D-KAF (one layer)	0.8750(0.001)	20851
2D-KAF (two layers)	**0.8764(0.002)**	81151

Table 2: Results of different activation functions on the SUSY benchmark. The last four rows are the proposed KAF and 2D-KAF. Standard deviation for the AUC is given between brackets, the best result is shown in bold, and the second best result is underlined. All networks with fixed or parametric activation functions have five hidden layers. See the text for a full description of the architectures.

There are several clear results that emerge from the analysis of Table 2. First of all, the use of sophisticated activation functions (such as the SELU), or of parametric functions (such as the PReLU) can improve performance, in some cases even significantly. However, these improvements still require several layers of depth, while they both fail to provide accurate results when experimenting with shallow networks. On the other hand, all non-parametric functions are able to achieve similar (or even superior) results, while only requiring one or two hidden layers of neurons. Among them, APL and Maxout achieve a similar AUC with one layer, but only APL is able to benefit from the addition of a second layer. Both KAF and 2D-KAF are able to significantly outperform all the competitors, and the overall best result is obtained by a 2D-KAF network with two hidden layers. This is obtained with a significant reduction in the number of trainable parameters, as also described more in depth in the following section.

8.5. Experiments with convolutional networks

Although our focus has been on feedforward networks, an interesting question is whether the superior performance exhibited by KAFs and 2D-KAFs can also be obtained on different architectures, such as convolutional neural networks (CNNs). To investigate this in this section we make several experiments with CNNs, starting from simple datasets and architectures and building towards more sophisticated tasks.

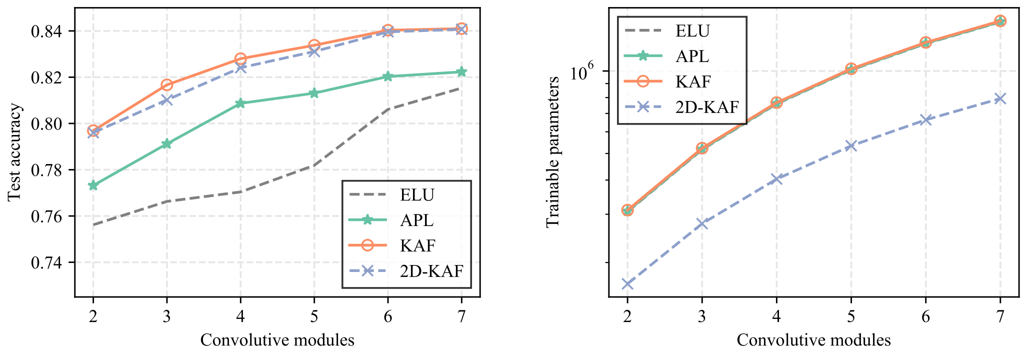
We start with an experiment on the classical MNIST dataset[9] composed of 70000 28×28 black-and-white images of handwritten digits. We consider a

[9] http://yann.lecun.com/exdb/mnist/

simple CNN architecture made of three convolutive layers, each composed of 50 filters with kernel size 5, interposed by max-pooling operations over 3×3 patches of the original image (Goodfellow et al., 2016). Without any data augmentation step, this architecture trained with the same procedure as the last section (and a regularization factor of 10^{-4}) obtains an average accuracy of 99.05% with ReLU nonlinearities, 99.31% with APL units, and 99.43% with KAFs (having $D = 10$ dictionary elements equispaced around -2.0 and 2.0 and initialized randomly). More interestingly, we apply the same setup to the more complex Fashion MNIST dataset (Xiao et al., 2017), a slightly harder version of the standard MNIST, having clothing articles instead of handwritten digits as classes. In this case, KAF networks achieve a 91.77% mean accuracy compared to 90.03% with ReLUs nonlinearities, with only a $\approx 1.5\%$ increase in the number of free parameters.

Next, we train several CNNs on the much harder CIFAR-10 dataset, composed of 60000 images of size 32×32, belonging to 10 classes. Since our aim is mostly to compare different architectures for the convolutional filters, for this experiment we train CNNs made by stacking convolutional 'modules', each of which is composed by (a) a convolutive layer with 150 filters, with a filter size of 5×5 and a stride of 1; (b) a max-pooling operation over 3×3 windows with stride of 2; (c) a dropout layer with probability of 0.25. We consider CNNs with a minimum of 2 such modules and a maximum of 7, where the output of the last dropout operation is flattened before applying a linear projection and a softmax operation. Our training setup is equivalent to the previous sections.

We consider different choices for the nonlinearity of the convolutional



(a) Test accuracy (b) Parameters

Figure 8: Results of KAF, 2D-KAF, and a baseline composed of ELU or APL functions when using only convolutive layers on the CIFAR-10 dataset (see the text for a full description of the architectures). (a) Test accuracy. (b) Number of trainable parameters for the architectures.

filters, using ELUs or APL units as baselines, and our proposed KAFs and 2D-KAFs. In order to improve the gradient flow in the initial stages of training, KAFs in this case are initialized with the KRR strategy using ELU as the target. The results are shown in Fig. 8, where we show on the left the final test accuracy, and on the right the number of trainable parameters of the three architectures.

Interestingly, both KAFs and 2D-KAFs are able to get significantly better results than the baseline, i.e., even 2 layers of convolutions are sufficient to surpass the accuracy obtained by an equivalent 5-layered network with the baseline activation functions. From Fig. 8b, we can see that this is obtained with a negligible increase in the number of trainable parameters for KAF, and with a significant decrease (roughly 50%) for 2D-KAF. The reason, as before, is that each nonlinearity for the 2D-KAF is merging information coming from two different convolutive filters, effectively halving the number of parameters

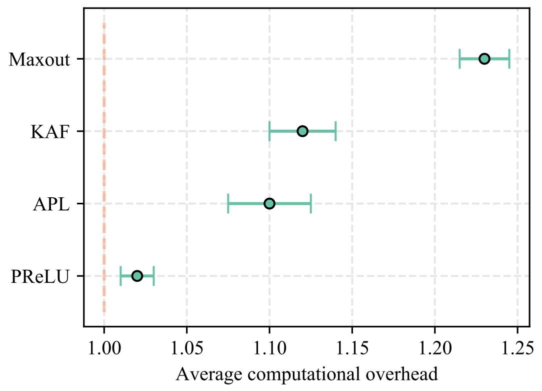


Figure 9: Average overhead time for the convolutive modules compared to a standard ReLU activation function (indicated by the dashed red line).

required for the subsequent layer.

Concerning computation time, Fig. 9 illustrates the overhead required (on average) for a single forward / backward pass for different flexible activation functions on the architectures tested for CIFAR-10. We see that the inclusion of KAFs requires on average the same time as the simpler APL, while allowing for higher expressiveness as illustrated in Fig. 8, and being considerably faster than an equivalent Maxout implementation over the different channels.

As a final experiment, we consider the CIFAR-100 dataset, a version of the CIFAR-10 with 100 different classes. In this case, we consider a known CNN architecture, the VGG-13 model from Simonyan and Zisserman (2014), which has 13 convolutional layers interspersed with max-pooling operations, with the number of filters linearly increasing from 32 to 512. We further augment each layer with a batch normalization operation (Goodfellow et al., 2016) to provide more regularization, and remove the final fully connected layers to focus on the effect of the convolutional filters. Despite the fact that

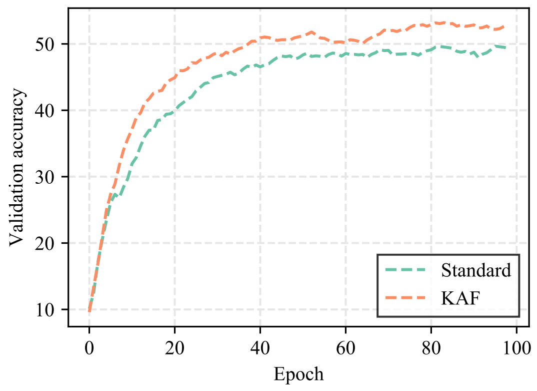


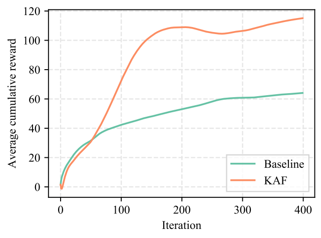
Figure 10: Validation accuracy on the CIFAR-100 dataset (see the text for details on the architectures).

the original architecture is already heavily fine-tuned, we obtain a final test accuracy which is 4.3% points higher by substituting all filters nonlinearities with KAFs. Fig. 10 shows the evolution of the validation accuracy for the two models.

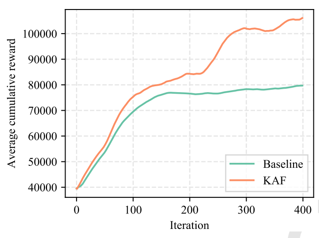
8.6. Experiments on a reinforcement learning scenario

Before concluding, we evaluate the performance of the proposed activation functions when applied to a relatively more complex reinforcement learning scenario. In particular, we consider some representative MuJoCo environments from the OpenAI Gym platform,[10] where the task is to learn a policy to control highly nonlinear physical systems, including pendulums and bipedal robots. As a baseline, we use the open-source OpenAI implementation of the proximal policy optimization algorithm (Schulman et al., 2017), that learns a policy function by alternating between gathering new

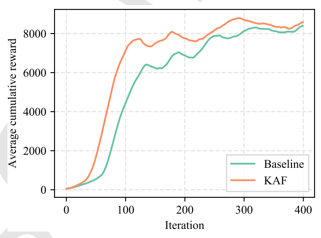
[10]`https://github.com/openai/gym/mujoco`



(a) swimmer



(b) humanoidstandup



(c) pendulum_inverted

Figure 11: Results for the reinforcement learning experiments, in terms of average cumulative rewards. We compare the baseline algorithm to an equivalent architecture with KAF nonlinearities. Details on the models and hyperparameters are provided in the main discussion.

episodes of interactions with the environment, and building the policy itself by optimizing a surrogate loss function. All hyper-parameters are taken directly from the original paper, without attempting a specific fine-tuning for our algorithm. The policy function for the baseline is implemented as a NN with two hidden layers, each of which has 64 hidden neurons with tanh nonlinearities, providing in output the mean of a Gaussian distribution that is used to select an action. For the comparison, we keep the overall setup fixed, but we replace the nonlinearities with KAF neurons, using the same initialization as the preceding sections.

We plot the average cumulative reward obtained for every iteration of the algorithms on different environments in Fig. 11. We see that the policy networks implemented with the KAF functions consistently learn faster than the baseline with, in several cases, a consistent improvement with respect to the final reward.

9. Conclusive remarks

In this paper, after extensively reviewing known methods to adapt the activation functions in a neural network, we proposed a novel family of non-parametric functions, framed in a kernel expansion of their input value. We showed that these functions combine several advantages of previous approaches, without introducing an excessive number of additional parameters. Furthermore, they are smooth over their entire domain, and their operations can be implemented easily with a high degree of vectorization. Our experiments showed that networks trained with these activations can obtain a higher accuracy than competing approaches on a number of different benchmark and scenarios, including feedforward and convolutional neural networks.

A potential drawback of our proposal is that it introduces additional design choices in the construction of the network, including how the dictionary is sampled, the selection of a kernel function, and the setting of its (eventual) hyper-parameters. In this paper we considered a specific selection for each of them, including the use of a fixed dictionary, of the Gaussian kernel, and of a hand-picked bandwidth for the kernel, that we found works well in a wide range of situations. However, many alternative choices are possible, such as the use of dictionary selection strategies, alternative kernels (e.g., periodic kernels), and several others. In particular, choosing a fixed set of points for the dictionary allows for an extremely efficient implementation, but we can envision more advanced strategies for adapting the sampled points. For example, there is a vast literature on learning the pseudo-inputs in the Gaussian processes (GPs) field (Titsias, 2009; Bauer et al., 2016) that could be leveraged by reformulating the proposed KAF method under a Bayesian perspective. In this respect, one intriguing aspect of the proposed activation functions is that they provide a further link between neural networks and kernel methods / GPs, opening the door to a large number of variations of the described framework.

Another possible concern of KAFs is that they introduce some computational complexity that might not be required in all situations (e.g., by using a predefined number of free parameters for each neuron and/or layer). To this end, we are actively working in designing some measure of complexity control (e.g., more advanced sparsification strategies) allowing to fine-tune the actual complexity of each function in a problem-dependent fashion.

Finally, we are interested in exploring the use of KAFs in state-of-the-art architectures (having possibly hundreds of hidden layers), to explore the gain in structural complexity that could be obtained. We also plan on investigating some theoretical properties of the proposed functions: for example, generalization bounds akin to those obtained in Eisenach et al. (2017) could be derived in a similar fashion because of the boundedness of the kernel functions we use. This would allow us to explicitly define the gain in expressive power obtained by selecting non-parametric activation functions instead of fixed ones.

Acknowledgments

The work of Steven Van Vaerenbergh is supported by the Ministerio de Economía, Industria y Competitividad (MINECO) of Spain under grants TEC2014-57402-JIN (PRISMA) and TEC2016-81900-REDT (KERMES). The authors would like to thank the anonymous reviewers for their suggestions on how to improve the paper.

References

Agostinelli, F., Hoffman, M., Sadowski, P., Baldi, P., 2014. Learning activation functions to improve deep neural networks. arXiv preprint arXiv:1412.6830.

Baldi, P., Sadowski, P., Whiteson, D., 2014. Searching for exotic particles in high-energy physics with deep learning. Nature communications 5, 4308.

Bauer, M., van der Wilk, M., Rasmussen, C. E., 2016. Understanding probabilistic sparse gaussian process approximations. In: Advances in Neural Information Processing Systems. pp. 1533–1541.

Bengio, Y., Léonard, N., Courville, A., 2013. Estimating or propagating gradients through stochastic neurons for conditional computation. arXiv preprint arXiv:1308.3432.

Bishop, C. M., 2006. Pattern recognition and machine learning. Springer International.

Chen, C.-T., Chang, W.-D., 1996. A feedforward neural network with function shape autotuning. Neural Networks 9 (4), 627–641.

Clevert, D.-A., Unterthiner, T., Hochreiter, S., 2016. Fast and accurate deep network learning by exponential linear units (ELUs). In: Proc. 2016 International Conference on Learning Representations (ICLR).

Cutajar, K., Bonilla, E. V., Michiardi, P., Filippone, M., 2017. Random feature expansions for deep gaussian processes. In: Proc. 34th International Conference on Machine Learning (ICML). pp. 884–893.

Cybenko, G., 1989. Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals, and Systems 2 (4), 303–314.

Damianou, A., Lawrence, N., 2013. Deep gaussian processes. In: Proc. 16th International Conference on Artificial Intelligence and Statistics (AISTATS). pp. 207–215.

Denil, M., Shakibi, B., Dinh, L., de Freitas, N., et al., 2013. Predicting parameters in deep learning. In: Advances in Neural Information Processing Systems. pp. 2148–2156.

Eisenach, C., Wang, Z., Liu, H., 2017. Nonparametrically learning activation functions in deep neural nets. In: 5th International Conference for Learning Representations (Workshop Track).

Glorot, X., Bengio, Y., 2010. Understanding the difficulty of training deep feedforward neural networks. In: Proc. 11th International Conference on Artificial Intelligence and Statistics (AISTATS). pp. 249–256.

Glorot, X., Bordes, A., Bengio, Y., 2011. Deep sparse rectifier neural networks. In: Proc. 14th International Conference on Artificial Intelligence and Statistics (AISTATS). p. 275.

Goh, S. L., Mandic, D. P., 2003. Recurrent neural networks with trainable amplitude of activation functions. Neural Networks 16 (8), 1095–1100.

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep learning. MIT Press.

Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., Bengio, Y., 2013. Maxout networks. In: Proc. 30th International Conference on Machine Learning (ICML).

Guarnieri, S., Piazza, F., Uncini, A., 1999. Multilayer feedforward networks with adaptive spline activation function. IEEE Transactions on Neural Networks 10 (3), 672–683.

Gulcehre, C., Cho, K., Pascanu, R., Bengio, Y., 2013. Learned-norm pooling for deep feedforward and recurrent neural networks. arXiv preprint arXiv:1311.1780.

Gulcehre, C., Moczulski, M., Denil, M., Bengio, Y., 2016. Noisy activation functions. In: Proc. 33rd International Conference on Machine Learning (ICML). pp. 3059–3068.

He, K., Zhang, X., Ren, S., Sun, J., 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In: Proc. IEEE International Conference on Computer Vision (ICCV). pp. 1026–1034.

Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., 2001. Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies. pp. 464–480.

Hofmann, T., Schölkopf, B., Smola, A. J., 2008. Kernel methods in machine learning. The Annals of Statistics, 1171–1220.

Hornik, K., Stinchcombe, M., White, H., 1989. Multilayer feedforward networks are universal approximators. Neural Networks 2 (5), 359–366.

Jin, X., Xu, C., Feng, J., Wei, Y., Xiong, J., Yan, S., 2016. Deep learning with S-shaped rectified linear activation units. In: Proc. Thirtieth AAAI Conference on Artificial Intelligence.

Jones, M. C., Marron, J. S., Sheather, S. J., 1996. A brief survey of bandwidth selection for density estimation. Journal of the American Statistical Association 91 (433), 401–407.

Kingma, D., Ba, J., 2015. Adam: A method for stochastic optimization. In: Proc. 3rd International Conference for Learning Representations (ICLR).

Klambauer, G., Unterthiner, T., Mayr, A., Hochreiter, S., 2017. Self-normalizing neural networks. In: Advances in Neural Information Processing Systems.

Li, H., Ouyang, W., Wang, X., 2016. Multi-bias non-linear activation in deep neural networks. In: Proc. 33rd International Conference on Machine Learning (ICML). pp. 221–229.

Lin, M., Chen, Q., Yan, S., 2014. Network in network. In: Proc. 2nd International Conference for Learning Representations (ICLR).

Liu, W., Principe, J. C., Haykin, S., 2011. Kernel adaptive filtering: a comprehensive introduction. John Wiley & Sons.

Maas, A. L., Hannun, A. Y., Ng, A. Y., 2013. Rectifier nonlinearities improve neural network acoustic models. In: Proc. 30th International Conference on Machine Learning (ICML).

Micchelli, C. A., Xu, Y., Zhang, H., 2006. Universal kernels. Journal of Machine Learning Research 7 (Dec), 2651–2667.

Park, J., Sandberg, I. W., 1991. Universal approximation using radial-basis-function networks. Neural Computation 3 (2), 246–257.

Piazza, F., Uncini, A., Zenobi, M., 1992. Artificial neural networks with adaptive polynomial activation function. In: Proc. 1992 IEEE International Joint Conference on Neural Networks (IJCNN).

Ramachandran, P., Zoph, B., Le, Q. V., 2017. Searching for activation functions. arXiv preprint arXiv:1710.05941.

Rennie, S. J., Goel, V., Thomas, S., 2014. Deep order statistic networks. In: Proc. 2014 IEEE Spoken Language Technology Workshop (SLT). IEEE, pp. 124–128.

Scardapane, S., Comminiello, D., Hussain, A., Uncini, A., 2017. Group sparse regularization for deep neural networks. Neurocomputing 241, 81–89.

Scardapane, S., Scarpiniti, M., Comminiello, D., Uncini, A., 2016. Learning activation functions from data using cubic spline interpolation. arXiv preprint arXiv:1605.05509.

Scarpiniti, M., Comminiello, D., Parisi, R., Uncini, A., 2013. Nonlinear spline adaptive filtering. Signal Processing 93 (4), 772–783.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O., 2017. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

Scott, D. W., 2015. Multivariate density estimation: theory, practice, and visualization. John Wiley & Sons.

Shang, W., Sohn, K., Almeida, D., Lee, H., 2016. Understanding and improving convolutional neural networks via concatenated rectified linear units. In: Proc. 33rd International Conference on Machine Learning (ICML). pp. 2217–2225.

Silverman, B. W., 1986. Density estimation for statistics and data analysis. Vol. 26. CRC press.

Simonyan, K., Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

Siniscalchi, S. M., Li, J., Lee, C.-H., 2013. Hermitian polynomial for speaker adaptation of connectionist speech recognition systems. IEEE Transactions on Audio, Speech, and Language Processing 21 (10), 2152–2161.

Siniscalchi, S. M., Salerno, V. M., 2017. Adaptation to new microphones using artificial neural networks with trainable activation functions. IEEE Transactions on Neural Networks and Learning Systems 28 (8), 1959–1965.

Snelson, E., Ghahramani, Z., 2006. Sparse gaussian processes using pseudo-inputs. In: Advances in Neural Information Processing Systems. pp. 1257–1264.

Solazzi, M., Uncini, A., 2000. Artificial neural networks with adaptive multidimensional spline activation functions. In: Proc. 2000 International Joint Conference on Neural Networks (IJCNN). Vol. 3. IEEE, pp. 471–476.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2014. Dropout: a simple way to prevent neural networks from overfitting. Journal of Machine Learning Research 15 (1), 1929–1958.

Titsias, M., 2009. Variational learning of inducing variables in sparse gaussian processes. In: Proc. 12th International Conference on Artificial Intelligence and Statistics (AISTATS). pp. 567–574.

Trentin, E., 2001. Networks with trainable amplitude of activation functions. Neural Networks 14 (4), 471–493.

Trottier, L., Giguère, P., Chaib-draa, B., 2016. Parametric exponential linear unit for deep convolutional neural networks. arXiv preprint arXiv:1605.09332.

Van Vaerenbergh, S., Lázaro-Gredilla, M., Santamaría, I., 2012. Kernel recursive least-squares tracker for time-varying regression. IEEE Transactions on Neural Networks and Learning Systems 23 (8), 1313–1326.

Vecci, L., Piazza, F., Uncini, A., 1998. Learning and approximation capabilities of adaptive spline activation function neural networks. Neural Networks 11 (2), 259–270.

Wilson, A. G., Hu, Z., Salakhutdinov, R., Xing, E. P., 2016. Deep kernel learning. In: Proc. 19th International Conference on Artificial Intelligence and Statistics (AISTATS). pp. 370–378.

Xiao, H., Rasul, K., Vollgraf, R., 2017. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. arXiv preprint arXiv:1708.07747.

Xu, B., Wang, N., Chen, T., Li, M., 2015. Empirical evaluation of rectified activations in convolutional network. arXiv preprint arXiv:1505.00853.

Zhang, X., Trmal, J., Povey, D., Khudanpur, S., 2014. Improving deep neural network acoustic models using generalized maxout networks. In: Proc. 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, pp. 215–219.