Kernelized Synaptic Weight Matrices

Lorenz K. Muller [1] **Julien N.P. Martel** [1] **Giacomo Indiveri** [1]

Abstract

In this paper we introduce a novel neural network architecture, in which weight matrices are reparametrized in terms of low-dimensional vectors, interacting through kernel functions. A layer of our network can be interpreted as introducing a (potentially infinitely wide) linear layer between input and output. We describe the theory underpinning this model and validate it with concrete examples, exploring how it can be used to impose structure on neural networks in diverse applications ranging from data visualization to recommender systems. We achieve state-of-the-art performance in a collaborative filtering task (MovieLens).

1. Introduction

Neural Networks have a large number of free parameters and often training algorithms need to choose between a range of near optimal value assignments for those parameters. This choice can be difficult, because optimality with respect to a training set does not guarantee good behavior on unseen, similar data (especially when there are many free parameters). This defines overfitting. To address this problem, regularization techniques are widely used in Neural Network optimization to help training procedures find generalizable solutions.

In this paper, we show that by expressing the weights of a neural network layer as the kernel interaction of low-dimensional vectors of free parameters, we can embed the weight matrix of a layer in some (potentially high-dimensional) feature-space; the embedding is controlled by the choice of the kernel function. This technique provides a structural way to regularize weight matrices.

[1]Institute of Neuroinformatics, University of Zurich and ETH Zurich, Switzerland. Correspondence to: Lorenz K. Muller <lorenz@ini.ethz.ch>.

Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018. Copyright 2018 by the author(s).

1.1. Related Work

There exist many approaches that reparametrize the weight matrix of a neural network. (Schmidhuber, 1997) and (Gomez & Schmidhuber, 2005) learned weights for a neural network by training either small programs or another neural network respectively to generate them. The approach of training a neural network to generate weights for another network can take many forms and reoccurs for example in the work of (Stanley et al., 2009; Ha et al., 2016; Fernando et al., 2016). In (Koutnik et al., 2010) the weight matrix of a neural network is decomposed by a discrete cosine transform (DCT) and learning is performed directly on the DCT parameters. Several recent papers propose different types of reparameterizations using various forms of matrix product decomposition (Denil et al., 2013; Moczulski et al., 2015; Tai et al., 2015).

Many methods have been proposed to ensure that at end of the training of a neural network the weights fulfill some desired properties: Weight-decay (or L_2 regularization) (Krogh & Hertz, 1992) makes large entries in the weight matrix costly, encouraging 'simple' models; drop-out (Srivastava et al., 2014) (the inclusion of multiplicative Bernoulli noise) performs model averaging for appropriate network architectures, prevents co-adaptation of different weights, and can be interpreted as letting networks approximate deep Gaussian processes (Gal & Ghahramani, 2016); low-rank decomposition of weight matrices (Sainath et al., 2013) (analogous to separable convolutions in ConvNets (Jaderberg et al., 2014)) is primarily used for computational speed-up and memory footprint reduction, but also has a regularizing effect. As we shall see in this work, low-rank decomposition is a special case of the method we propose in this paper.

In contrast to these approaches, our method allows the imposition of low-dimensional structure onto the network. We will show on several examples that the low dimensional embedding of the network weights does not only reduce the number of free parameters in a network, but has the advantage of increasing its interpretability as well as allowing for structural regularization.

1.2. Core Idea

Instead of assigning an individual, free weight-parameter between each input and output neuron of a neural network layer, we associate with each input and each output a vector of free parameters that we think of as a location in a low-dimensional space. The weight between two units is then set to be some fixed function of the distance between those locations (or more precisely a weighted kernel-function of the location vectors). A possible choice of kernel would be that neurons that are far away from each other are unconnected, neurons that are close to each other are strongly connected. Thus, the function of a layer is to produce a smooth kernel function, centered at the input locations, which is sampled at a few points by the output neurons (namely at their locations).

2. Theory

2.1. Definition: kernelNet

We define a d-dimensional kernelized neural network (d-kernelNet) as a hierarchical function approximator on inputs $\vec{x}^{(0)}$ and outputs $\vec{x}^{(N)}$ of the form:

$$x_j^{(l)} = f_j\left(\sum_i \alpha_i^{(l)} K(\vec{u}_i^{(l)}, \vec{v}_j^{(l)}) x_i^{(l-1)}\right). \tag{1}$$

where super-scripts are layer indices, the functions f_j are non-linearities, α_i are scalar parameters, and $K(\cdot,\cdot)$ is a kernel function that corresponds to an inner product, in some embedding space; i.e.

$$K(\vec{u},\vec{v}) = \left\langle \phi(\vec{u}), \phi(\vec{v}) \right\rangle = \langle \vec{u}^*, \vec{v}^* \rangle \tag{2}$$

with the embedding function $\phi : \mathbb{R}^d \to \mathbb{R}^{d^*}$. The d dimensional vectors $\vec{u}$, $\vec{v}$ are the free parameters of the model and $\vec{u}^*, \vec{v}^*$ denote their embedded counterparts $\vec{u}^* = \phi(\vec{u})$ and $\vec{v}^* = \phi(\vec{v})$ of dimension d^*.

Note that this is essentially a neural network layer, where the weight matrix $\mathbf{W}$ has been replaced with a sum of weighted kernels.

2.2. Relation to Fully-Connected Neural Networks

In the special case of setting ϕ to the identity (and by consequence $K(\cdot,\cdot)$ to the dot-product) and setting $\alpha_i = 1$, we obtain a fully-connected neural network, into each of whose layers a linear layer of a size d has been interposed. Consider the activation of such a layer, before the non-linearity (as we consider a single layer, we left out the superscript of α, u and v for readability):

$$\begin{aligned} x_j^{(l)} &= \sum_i \alpha_i K(\vec{u}_i, \vec{v}_j) x_i^{(l-1)} = \sum_i \alpha_i \vec{u}_i^T \vec{v}_j x_i^{(l-1)} \\ &= \sum_i \alpha_i \sum_k u_{ki} v_{jk} x_i^{(l-1)} = \sum_k v_{jk} \sum_i \alpha_i u_{ki} x_i^{(l-1)} \end{aligned} \tag{3}$$

Equation (3) describes a fully connected neural network layer with an intermediate layer, whose neurons with linear

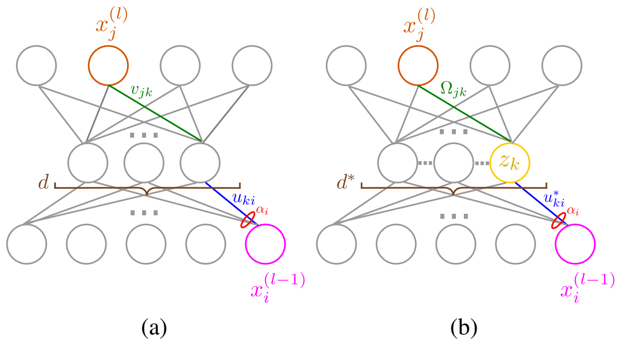


Figure 1. Illustrations to visualize the "virtual" layer introduced in (a) Equation (3) and (b) Equation (4)

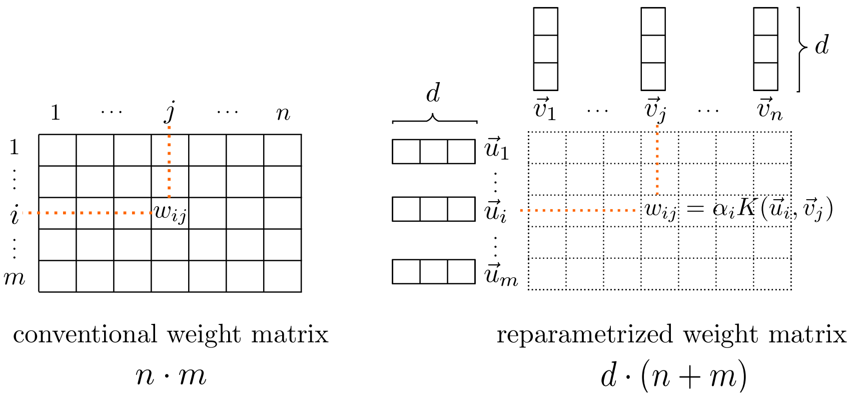
activation function are indexed by k, as illustrated in Figure 1(a). We have connection matrices u_{ki}, v_{jk} to and from the linear layer. Alternatively we could interpret this as a product decomposition of the full weight matrix between $x^{(l)}$ and $x^{(l-1)}$. The dimension d in this case is the rank of the 'effective' weight matrix from $x_i^{(l-1)}$ to $x_j^{(l)}$. Such product decompositions are a well-known approach to neural network regularization and compression (Sainath et al., 2013).

For choices of ϕ other than the identity, we obtain a virtual linear layer of dimension d^* (that is potentially much greater than d), and whose connection matrices are constrained by ϕ. We compute the activation $\vec{x}^{(l)}$ before the activation function of the layer:

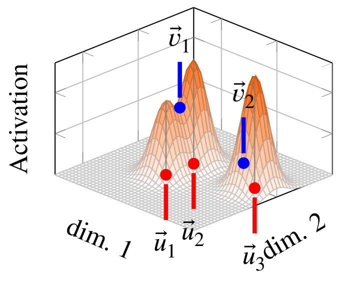
$$\begin{aligned} x_j^{(l)} &= \sum_i \alpha_i K(\vec{u}_i, \vec{v}_j) x_i^{(l-1)} = \sum_i \alpha_i \langle \phi(\vec{u}_i), \phi(\vec{v}_j) \rangle x_i^{(l-1)} \\ &= \sum_i \alpha_i \vec{u}_i^{*T} \mathbf{M}_\phi \vec{v}_j^* x_i^{(l-1)} = \sum_i \alpha_i \left[\sum_{h,k} u^*_{ki} M_{kh} v^*_{jh}\right] x_i^{(l-1)} \\ &= \sum_{h,k} v^*_{jh} M_{kh} \underbrace{\left[\sum_i \alpha_i u^*_{ki} x_i^{(l-1)}\right]}_{z_k} = \sum_k z_k \left[\sum_h v^*_{jh} M_{kh}\right] \\ &= \sum_k z_k \Omega_{jk} \end{aligned} \tag{4}$$

where we used Eq. 2 and the Hermitian form of the inner product (note that $\mathbf{M}_\phi$ is necessarily symmetric positive definite and is induced by ϕ).

In Equation (4) we can see that the proposed substitution corresponds to the insertion of a linear layer with unit activations $\vec{z}$. The connections *to* this layer are u^*_{ki} and *from* this layer Ω_{jk} as illustrated in Figure 1(b). These matrices are dependent on the free parameters as well as ϕ, which determines their structure: Notably the index k runs over the dimensions of the *embedded* vectors $\vec{u}^*, \vec{v}^*$; ϕ prescribes how the finite entries of $\vec{v}, \vec{u}$ make up these potentially infinite



(a)



(b)

Figure 2. (a) Schematic comparison of a Kernelized and a standard synaptic weight matrix (b) Visualization of the activation of a layer with a two dimensional RBF Kernel of a 3×2 weight matrix. At the locations $\vec{u}_i$ we input a kernel (here a Gaussian) scaled by the input to unit i and α_i. At the locations $\vec{v}_i$ we read out the height of the kernel sum.

dimensional vectors. The dimensionality of this embedding depends on the choice of ϕ or in practice $K(\cdot,\cdot)$. However, we never need to explicitly evaluate ϕ nor perform the dot product in $d*$.

The idea of using a kernel-function to get high-dimensional interactions between low dimensional vectors without needing to compute in the high-dimensional space, is known as the 'Kernel-Trick' in the kernelized machines literature, e.g. kernelized SVMs (Scholkopf & Smola, 2001). In such kernelized SVM classifiers, instead of computing the dot-product between data points $\vec{x}$ and centroids $\vec{c}_i$ (that are more usually called support vectors) of standard SVMs one evaluates a kernel function $K(\vec{x}, \vec{c}_i)$. However this kernel takes data ($\vec{x}$) as one of its arguments (the other one being a centroid, which in the case of SVM is also a data-point), while in our case both arguments are parameters.

In analogy to the standard fully-connected neural network layer, we can regard the kernelNet-layer as a fully-connected layer, whose weight matrix has been reparameterized by the parameters $\vec{u}, \vec{v}$ that are "decompressed" by a kernel function $K(\cdot,\cdot)$. Notably the number of parameters of such a reparametrized weight matrix is only $\mathcal{O}\big(d \cdot (m+n)\big)$ rather than $\mathcal{O}(m \cdot n)$ (with m the dimension of the input layer and n the dimension of the output layer) as seen in Figure 2. Hence, with our reparametrization, the number of parameters is reduced, as long as d is less than half of the harmonic mean of m and n: $d < \frac{m \cdot n}{m+n} = \frac{1}{2}\mathcal{H}(m,n)$.

If the kernel function used in a kernelNet layer is differentiable, it can be trained through stochastic-gradient descent algorithms and its variants (see experiments in Section 3).

2.3. Radial Basis Function Kernels

Some kernel functions can be visualized easily (at least for a low dimensional d) and we will see that consequently also the input to neurons in a layer that uses this kernel can be visualized. A notable example of this is the Radial-Basis-Function (RBF) kernel, which has the form:

$$K(\vec{u}, \vec{v}) = \alpha \cdot \psi\big(D(\vec{u}, \vec{v})\big) \tag{5}$$

where ψ is a function $\psi : \mathbb{R}^+ \to \mathbb{R}$ and $D(\cdot,\cdot)$ is a distance and $\alpha \in \mathbb{R}$. Then we can interpret the d-KernelNet layer as follows: Input channels place a kernel scaled by their activation (and by a basis weight α) in a d dimensional space centered at $\vec{u}_i$, output neurons read the sums of these kernels at $\vec{v}_j$. For an illustration see Figure 2.

2.3.1. Gaussian RBF Kernels

The Gaussian RBF kernel

$$K(\vec{v}, \vec{u}) = \exp(-\gamma ||\vec{u} - \vec{v}||_2^2) \tag{6}$$

is of theoretical interest because its embedding function ϕ is well-known and maps into an infinite dimensional space (Scholkopf & Smola, 2001). Furthermore outputs of this kernel can be interpreted as a similarity measure (it maps to 1 for identical vectors and asymptotically approaches 0 for very distant vectors).

2.3.2. Finite Support RBF Kernels

RBF kernels whose support is finite can be used to impose a variable degree of sparsity (in a "L_0" sense) on the effective connectivity of the embedded network layer; this can be applied to a non-kernelized network by expressing an effective weight matrix as the Hadamard-product of a unconstrained matrix and a finite-support kernel matrix.

Consider an embedding as above with $\vec{v}_j$ fixed on a grid with grid constant b. We can then use a finite support kernel

$$K_{\text{fs}}(\vec{u}_i, \vec{v}_j) = \max\left(0, 1 - a \cdot D(\vec{u}_i, \vec{v}_j)\right) \tag{7}$$

where $D(\cdot, \cdot)$ is a distance. By scaling a and b we can choose the maximal number of input neurons any given output neuron can get input from. Alternatively $\vec{v}_j$ can remain free and a cost term of e.g. the form

$$R = \lambda_0 \sum_{ij} (K(\vec{u}_i, \vec{v}_j))^2, \tag{8}$$

can be introduced, measuring the overlaps between "bumps", in which λ_0 will control the degree of sparsity of the trained model (see experiments in Section 3.5).

A model of the form of Equation (7) must, by construction, find a decomposition into independent sub-parts: Each input channel can only affect the activity of a few output channels. As the model becomes deeper, however, these sparse channels get mixed, so that for random connectivity the probability of an output to connect to a given 1st layer input, increases exponentially with the depth of the model.

3. Experiments

In the following experiments we demonstrate on some examples, how kernelNets can be used in practice. We investigate the effect of using different kernel functions (Section 3.1), show how to incorporate prior knowledge into the network parameters (Section 3.2), create extensible data visualizations (Section 3.3, 3.4) and achieve state-of-the art performance on the MovieLens dataset (Section 3.5), while reducing the computational complexity of the model.

3.1. Impact of Effective Dimensionality

Here we evaluate the performance of three kernelNets of the same structure as a function of the dimensionality d, but using different kernels, whose corresponding embedding ϕ maps to spaces of different dimensionality d^*. Namely we use a dot-product kernel (embedding dimensionality $d^* = d$), a second degree polynomial kernel (embedding dimensionality $d^* = \frac{1}{2}(d+1) \cdot (d+2)$) and a Gaussian RBF kernel (embedding dimensionality d^* is infinite independent of d).

All networks were trained using the ADAM learning rule and a range of hyperparameters (learning rate and l2 regularization); for each network the best mean performance over 5 repetitions is shown.[1]

In Figure 3 we see that kernels, whose corresponding embedding function ϕ projects into a higher dimensional space,

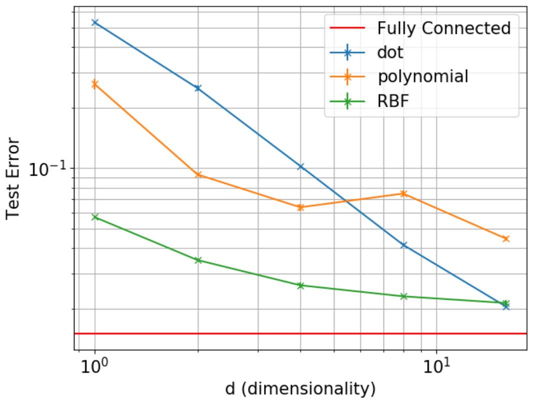


Figure 3. Comparison of identical kernelNets with different kernel functions. Kernels corresponding to higher dimensional embeddings work better for very low dimensional parametrizations. Errorbars indicate the standard error from five repetitions.

lead to better results for very low dimensional parametrization (low values of d). At high dimensionality the dot-product kernel (corresponding to the inclusion of a linear layer) catches up, and even slightly overtakes the other kernels, probably due to the fact that it is easier to optimize.

Notably it is thus more memory efficient to use a higher dimensional kernel; given some target performance, a higher-order kernel will often achieve it with fewer parameters. In a setting where memory look-ups dominate the computational cost, higher-order kernels are a preferable alternative to the commonly used dot-product kernel in model compression.

3.2. Incorporating Channel Relationships

In some situations the vectors $\vec{u}, \vec{v}$ need not be initialized randomly: If there is a known low-dimensional relationship between the different input channels, it can be beneficial to incorporate such knowledge in the choice of initial $\vec{u}, \vec{v}$.

A concrete example is image data for which there is far greater correlation between nearby pixels than between very distant pixels. This distance-dependent correlation is a useful piece of information that can be incorporated into a model. In a kernelNet, this can be achieved by initializing $\vec{u}, \vec{v}$ on a 2 dimensional grid.

A kernelNet initialized on a 2D grid with a RBF kernel, could be thought of as a convolutional neural network, in which each layer has only a single kernel, but this kernel changes slowly across the image; as a consequence there is no translation equivariance in the kernelNet. See Figure 4 for an illustration of receptive fields in the first layer of a network trained on MNIST. With this intuition it becomes apparent that it may be helpful to instantiate several such

[1]code available in supplement

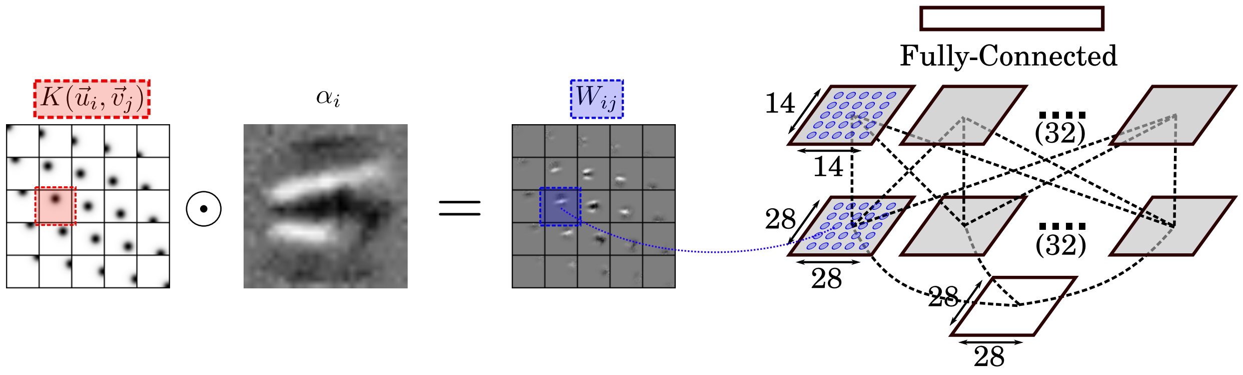


Figure 4. Architecture of a kernelNet for image classification. Dotted lines indicate 2D RBF kernelized connection matrices, an array of which is visualized on the left. At the bottom an MNIST digit is input, at the top is a fully-connected softmax layer, between are 2D RBF kernelNet-layers, either initialized with or without knowledge of the input pixel locations.

A-priori known pixel-locations	Acc.
no	98.89 %
yes	99.03 %

Table 1. Test set classification accuracy on MNIST of the 2-kernelNet in Figure 4 making use of prior knowledge of pixel locations or not. Adding the prior knowledge improves the performance. The performance is good for a non-convolutional network trained without data-augmentation.

2D grids in parallel (analogous to having multiple filters in a single layer of a ConvNet), see Figure 4 for the resulting network architecture.

In Table 1 we compare the performance of kernelNets using spatial information (by setting the initial $\vec{u}, \vec{v}$ appropriately and keeping them fixed) against the same network lacking this information (randomly initialized $\vec{u}, \vec{v}$, trained) in an MNIST classification task. Adding prior information in the architecture indeed improves performance of the network.

An interesting consequence of interpretable $\vec{u}_i$, is that they permit the incorporation of interpretable noise into the network: When $\vec{u}_i$ corresponds to a pixel location, adding noise to it can be viewed as a model of uncertainty about the location of the pixel. This may be beneficial for architectures akin to (Gal & Ghahramani, 2016).

3.3. Extensible Data Visualization

In this section we use a kernelNet-layer to create an extensible 2D data embedding to create t-SNE-like (Maaten & Hinton, 2008) data visualizations.

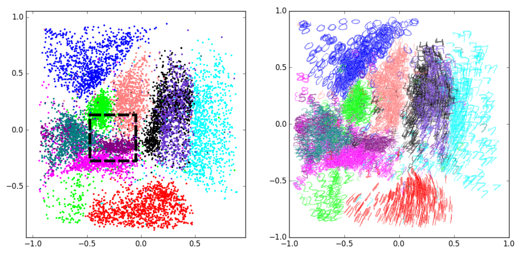
We build a deep, fully-connected network that contains a single kernelNet-layer in the middle. We choose a 2D RBF-kernel to enforce that neurons in the kernelNet-layer assume a two dimensional spatial organization, and fix the parameters $\vec{v}$ to lie on a grid (to facilitate visualizations like in Figure 6). Furthermore we equip this layer with the following non-linearity, chosen to ensure sparsity (which in conjunction with the spatial organization of the synaptic matrix leads to spatially unimodal activations)

$$f(\vec{x}) = \max\left(0, \vec{x} - (\eta \cdot \hat{x} + (1-\eta) \cdot \bar{x})\right), \tag{9}$$

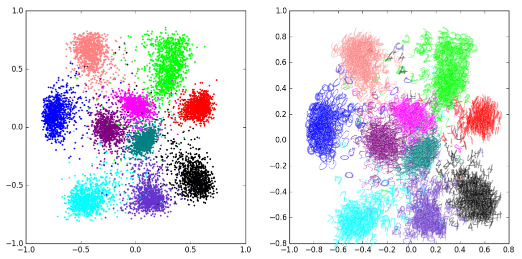
where $\hat{x}$ is the maximal value of $\vec{x}$, $\bar{x}$ is its mean value, and η is an interpolating parameter controlling sparsity (for well-behaved activations this approximates a percentile clipping function). The full network layout is $[784-2000-2000-2500-K1600-2000-2000-(10/784)]$, where all layers are fully connected, except for the one prefixed with a 'K', which is a kernelNet layer (for further details please consult the supplement). From the activations of this layer we construct a 2D embedding of the input: The location to which a particular input to the network is mapped for our data embedding, is the weighted sum of the locations of the hidden units, where the weight is the activation of the unit. We refer to this as the center of mass ($\vec{c}$) of that input

$$\vec{c} = \frac{\sum_i x_i \vec{v}_i}{\sum_i x_i} \tag{10}$$

In Figure 5 we see a 2D map of MNIST digits constructed by such a network trained with two different costs and final layers: An autoencoder (Bengio et al., 2013) (with per pixel cross-entropy), and a classifier (with categorical cross-entropy). Notably the shown digits are test data and have never been seen by the network. In contrast to e.g. t-SNE (Maaten & Hinton, 2008) this method thus is naturally extensible beyond the training set (though there exist extensible variants (van der Maaten, 2009)). Furthermore our method distinguishes itself by the fact that the embedding can be based on any cost function. The layers preceding the kernelNet layer, learn a mapping from the input space to the latent space sampled (at locations $\vec{v}$) by the kernelNet units; the layers following the embedding layer, learn a mapping



(a)



(b)

Figure 5. Embedding of unseen (test set) MNIST digits using a kernelNet-layer. (a) Trained as an autoencoder (without labels) (b) Trained as a classifier (with labels). The black, dashed window is also shown in a different representation in Figure 6. The left figures show the digits with points placed at their location according to Eq (10) while the right figures show the images of the digits placed at those locations.

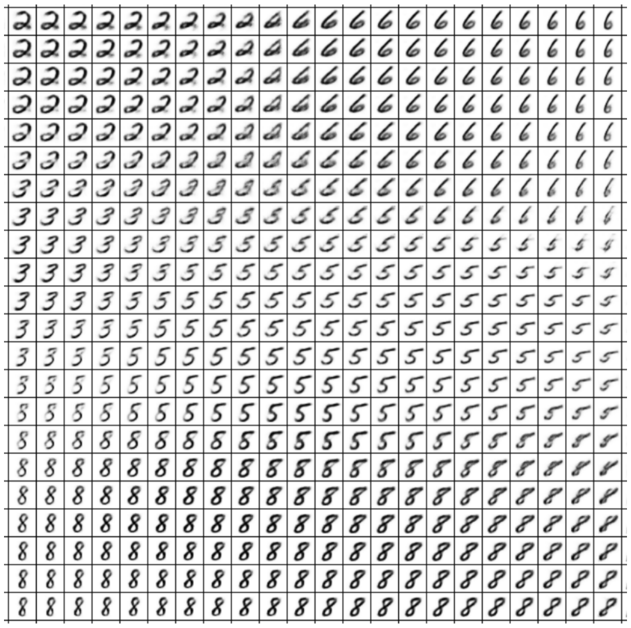


Figure 6. Visualization of part of the latent space constructed by the kernelized layer when presented with an artificial, shifted Gaussian input, traveling through the black dashed window in Figure 5. Larger Image in supplement.

between the latent space –in which the units of the kernelNet layer live– to the output. This second part of the network thus mitigates between the cost-function and the embedding.[2]

When the whole network is trained as an autoencoder, the decoder part operates as a map from 2D Gaussian kernels (shifted according to Equation (9)) to the space of MNIST digits. Figure 6 illustrates what digit the decoder constructs, for an artificial, shifted Gaussian input, whose center is slowly translated in the 2D kernelized space. Notably, the network produces sensible outputs at a very fine granularity.

From a conceptual point of view, this network shares similarities with variational autoencoders (Kingma & Welling, 2013) or adversarial autoencoders (Makhzani et al., 2015) in that it learns the mapping (and its inverse) of a data space to a low dimensional latent space. We encourage a comparison of Figure 6 with Figure 2 of (Makhzani et al., 2015). In these other works, though the method is different, coordinates in the latent d-dimensional space are explicitly represented by a layer with d units; these units are constrained by an additional cost term, to sample from a desired distribution. In this work, we use the structural constraint of a weight matrix lying in a low dimensional kernel space to achieve a similar outcome.

3.4. Pre-trained Network Visualization

Here we visualize the action of a pre-trained network (a convolutional ResNet (He et al., 2016)) by copying it up to some prespecified depth and adding on top a fully-connected layer (with 500 units), a 2D-RBF kernelNet-layer (with 900 units) and a classification layer (a 10-way softmax). We then train the newly appended part of this truncated network with the same cost as the original network and visualize the activation in the embedding layer as described in the previous section (the lower part of the network remains unchanged)

Note that the embedding layers are trained to optimize the same cost (categorical cross-entropy) as the original network. Indeed the output of the embedding layer is sufficiently informative to reach an equally good (or slightly better) classification as the original network (see Table 2). The resulting embeddings (created as in the previous section) are shown in Figure 7.

These visualizations contain similar information as a confusion matrix would, but more intuitively presented, showing how well the various classes are separated from each other. Figure 7 shows how the classes overlap increasingly more, as we visualize the network action for lower layers.

[2]Code available in the supplement.

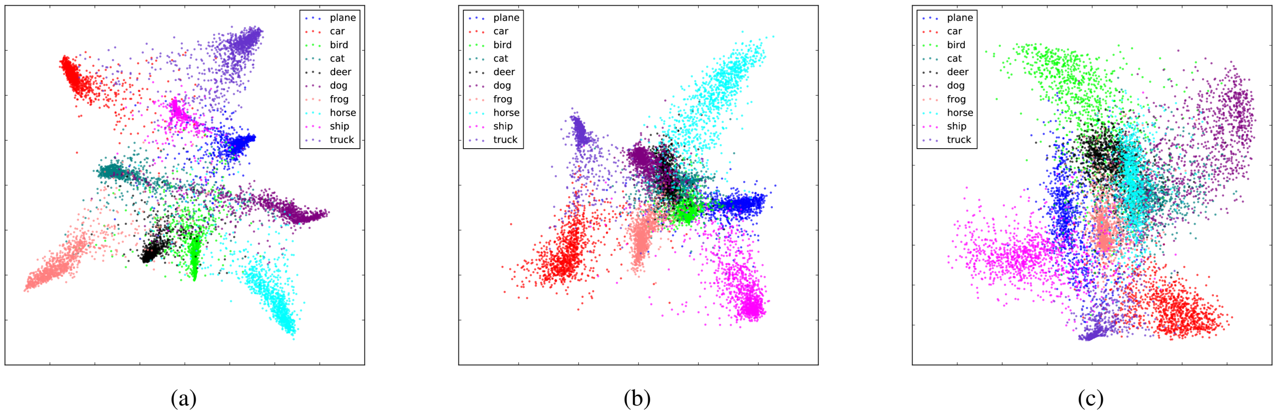


Figure 7. (a) Embedding of test-set CIFAR-10 images created by an RBF Kernelized layer stacked onto a CIFAR-10 pre-trained, 51-layer ResNet without the final classification layer (b) like previous, but the last 11 layers of the ResNet were removed (c) last 21 layers removed [Best viewed in color]

Network	kept layers	Addit. Layers	Acc.
ResNet (n=5)	32	0	92.3 %
ResNet + ker	31	3	92.6 %
ResNet + ker	21	3	90.5 %
ResNet + ker	11	3	82.3 %

Table 2. Classification performance of the pre-trained truncated ResNets used for the visualizations in Figure 7.

3.5. Recommender Systems

Recommender systems typically operate on sparse high-dimensional data. For instance, one aims at predicting movie ratings for a user based on millions of other users having each seen a small subset of thousands of movies (Lam & Herlocker, 2012). In such settings, it is a common assumption that the sparsely observed matrix entries, from which one ought to generalize, can in some way be represented in a low dimensional space. Indeed, movie-ratings supposedly correlate with a relatively small number of features: the combination of an actor playing in it and a movie genre for example. As data is expected to be best explained by such a low dimensional model it seems this is a well-suited setting for kernelNets.

3.5.1. Dataset

We train our models to predict movie ratings of the MovieLens-10M (ML-10M), MovieLens-1M (ML-1M) and MovieLens-100K (ML-100K) datasets (Harper & Konstan, 2016). These datasets comprise (10 million / 1 million / 100 thousand) ratings of (10681 / 3706 / 1700) movies by ca. (71 / 16 / 1) thousand users respectively, on a scale of $r \in \{1, 2, 3, 4, 5\}$ (ML-10M include half ratings). The datasets are highly sparse (density 0.013 / 0.045 / 0.059). We randomly designate 10% or 20% respectively of the given ratings as validation data (so chosen to match the models we compare to). The validation data is not used in training and used alone in the reported error computation. Reported performances average over five such random splits. We report the root-mean-square error (Equation (11)).

$$E_{rmse} = \sqrt{\sum_i (p_i - r_i)^2 / N}, \tag{11}$$

where p_i is the predicted rating, r_i is the true rating. N is the number of validation samples.

3.5.2. Model

Our model is an item-based autoencoder very similar to (Sedhain et al., 2015), but the weight matrices $\mathbf{W}, \mathbf{V}$ are reparameterized. Firstly we use a kernelNet-Layer with the following kernel (which is not positive semi-definite, but works well in practice)

$$K_\sigma(\vec{u}, \vec{v}) = \tanh(\vec{u} \cdot \vec{v}) \tag{12}$$

secondly we use the Hadamard-product of a dense connection matrix with a kernelized weight matrix, with finite-support kernel (as in Eq. (7)), to obtain sparse connection matrices. The full model then is

$$h(\vec{r}, \theta) = f\left(\mathbf{W} \cdot g\left(\mathbf{V}\vec{r} + \vec{\mu}\right) + \vec{b}\right) \tag{13}$$

where the weight matrices either take the form

$$\begin{aligned} V_{ij} &= \alpha_i K_\sigma(\vec{v}_i, \vec{u}_j) && (14) \\ W_{ij} &= \beta_i K_\sigma(\vec{s}_i, \vec{t}_j). && (15) \end{aligned}$$

or the form using K_{fs} (defined in Eq. (7)), that we will refer to as 'sparse fully-connected':

$$\begin{aligned} V_{ij} &= \alpha_{ij} K_{\text{fs}}(\vec{v}_i, \vec{u}_j) && (16) \\ W_{ij} &= \beta_{ij} K_{\text{fs}}(\vec{s}_i, \vec{t}_j). && (17) \end{aligned}$$

Method	ML-10M	ML-1M	ML-100K
LLORMA	0.782	0.833	0.898
GC-MC	0.777	0.832	0.910
I-CFN	0.777	0.832	-
I-AutoRec	0.782	0.831	0.895*
I-AutoRec (2)	0.770*	0.830*	0.895*
CF-NADE (2)	0.771	0.829	-
KernelNet (1)	-	0.838	0.898
KernelNet (2)	-	0.836	0.901
Sparse FC (1)	0.784	0.830	**0.890**
Sparse FC (2)	**0.769**	**0.824**	0.894

Table 3. Comparison of various methods on MovieLens tasks; the mean RMSE of the predicted ratings is given (lower is better) in our case from five CV folds (Sparse FC std. err. < 0.0005 for ML-1M, ML-10M and < 0.005 for ML-100K), training validation split 90/10 (ML-1M, ML-10M) and 80/20 (ML-100K). Numbers in brackets indicate number of hidden layers used. Architectures from this work follow the second horizontal line. *Our implementation.

As in (Sedhain et al., 2015) for optimization we use the L-BFGS-B (Zhu et al., 1997) and RPROP (Riedmiller & Braun, 1993) optimizers to minimize a regularized square-error, the regularization term added to the cost is

$$R = \lambda_2 \left(W_{ij}^2 + V_{ij}^2 \right),$$

and in the sparse fully-connected case

$$R = \lambda_2 \left(\sum_{ij} \alpha_{ij} + \sum_{ij} \beta_{ij} \right) + \lambda_0 \left(\sum_{ij} K(\vec{v}_i, \vec{u}_j) + \sum_{ij} K(\vec{s}_i, \vec{t}_j) \right),$$

One may think of λ_2 as the L_2 regularization and λ_0 as the sparsity "L_0" regularization parameters. For the kernelNet we use $d = 50$ and for the sparse fully-connected case $d = 5$. All hidden layers have size 500.[3]

3.5.3. Results

In Table 3 we report the mean validation RMSE of 5 runs in which each time a randomly drawn 10% of the ML-1M and ML-10M dataset and 20% of the ML-100K dataset were used as validation data and compare to the following methods: LLORMA (Lee et al., 2016), GC-MC (van den Berg et al., 2017), I-CFN (Strub et al., 2016), I-AutoRec (Sedhain et al., 2015) and CF-NADE (Zheng et al., 2016). We cite the performance of the models trained without information outside ratings (such as movie genres, user age, etc.) as we did not use such information either.

Table 4 further shows comparisons and highlights the gained efficiency in terms of multiply-accumulate operations (MACs) (i.e. lower number of non-zero entries in the weight matrix) thanks to the here proposed parameterization. The parameter λ_0 allows trading-off Number of MACs for performance; the table shows the best performing model. Our model (Sparse FC (2)) shows state-of-the-art performance, while decreasing the number of MACs required for its evaluation by ca 30% compared to fully-connected, unaltered I-AutoRec model. The performance gain is especially pronounced for the intermediate size dataset.

[3]Code available in the supplement.

Method	Parameters	MACs	ML-1M
I-AutoRec (1)	6.05 M	3.03 M	0.831
I-AutoRec (2)	6.30 M	3.28 M	0.830
I-KernelNet (1)	0.67 M	3.03 M	0.838
I-KernelNet (2)	0.72 M	3.28 M	0.836
Sparse FC (1)	6.70 M	2.77 M	0.830
Sparse FC (2)	7.00 M	2.23 M	0.824

Table 4. Here we highlight how the two proposed reparameterizations reduce the number of free parameters (KernelNet) or the expected number of MACs required (Sparse FC) for a prediction (assuming 10 random rated Movies and dense prediction on ML-1M).

4. Conclusion

We presented a novel neural network layer structure, based on kernel-approximations of the synaptic weight matrix. We detailed the mathematical relationship of the proposed kernelNet to standard fully-connected layers. Furthermore we demonstrated state-of-the-art performance on various MovieLens datasets (in terms of generalization MSE), using an Autoencoder whose weight matrices were sparsified using finite-support kernels, which additionally decreased the computational cost at inference in terms of multiply-accumulate operations. Finally we gave some illustrative examples with further possible applications, including a natively extensible data visualization technique that can be trained to reflect any cost function. KernelNets give a new approach to imposing structure on neural networks, regularizing and sparsifying them and for making the inner workings – also of pretrained networks – more easily interpretable.

Acknowledgements

We acknowledge funding by the Swiss National Science Foundation under grant number CRSII2_160756. We thank the NCS and CO^2 groups at the Institute of Neuroinformatics for helpful discussions, and we thank the reviewers for their work and their insightful comments.

References

Bengio, Y., Courville, A., and Vincent, P. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8): 1798–1828, 2013.

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

Fernando, C., Banarse, D., Reynolds, M., Besse, F., Pfau, D., Jaderberg, M., Lanctot, M., and Wierstra, D. Convolution by evolution: Differentiable pattern producing networks. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, pp. 109–116. ACM, 2016.

Gal, Y. and Ghahramani, Z. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pp. 1050–1059, 2016.

Gomez, F. and Schmidhuber, J. Evolving modular fast-weight networks for control. *Artificial Neural Networks: Formal Models and Their Applications–ICANN 2005*, pp. 750–750, 2005.

Ha, D., Dai, A., and Le, Q. V. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

Harper, F. M. and Konstan, J. A. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 5(4):19, 2016.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pp. 448–456, 2015.

Jaderberg, M., Vedaldi, A., and Zisserman, A. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

Kingma, D. P. and Welling, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Klambauer, G., Unterthiner, T., Mayr, A., and Hochreiter, S. Self-normalizing neural networks. In *Advances in Neural Information Processing Systems*, pp. 972–981, 2017.

Koutnik, J., Gomez, F., and Schmidhuber, J. Evolving neural networks in compressed weight space. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pp. 619–626. ACM, 2010.

Krogh, A. and Hertz, J. A. A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pp. 950–957, 1992.

Lam, S. and Herlocker, J. Movielens 1m dataset, 2012.

Lee, J., Kim, S., Lebanon, G., Singer, Y., and Bengio, S. Llorma: Local low-rank matrix approximation. *The Journal of Machine Learning Research*, 17(1):442–465, 2016.

Maaten, L. v. d. and Hinton, G. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov): 2579–2605, 2008.

Makhzani, A., Shlens, J., Jaitly, N., Goodfellow, I., and Frey, B. Adversarial autoencoders. *arXiv preprint arXiv:1511.05644*, 2015.

Moczulski, M., Denil, M., Appleyard, J., and de Freitas, N. Acdc: A structured efficient linear layer. *arXiv preprint arXiv:1511.05946*, 2015.

Riedmiller, M. and Braun, H. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pp. 586–591. IEEE, 1993.

Sainath, T. N., Kingsbury, B., Sindhwani, V., Arisoy, E., and Ramabhadran, B. Low-rank matrix factorization for deepneuralnetworktrainingwithhigh-dimensionalout-put targets. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 6655–6659. IEEE, 2013.

Schmidhuber, J. Discovering neural nets with low kolmogorov complexity and high generalization capability. *Neural Networks*, 10(5):857–873, 1997.

Scholkopf, B. and Smola, A. J. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2001.

Sedhain, S., Menon, A. K., Sanner, S., and Xie, L. Autorec: Autoencoders meet collaborative filtering. In *Proceedings of the 24th International Conference on World Wide Web*, pp. 111–112. ACM, 2015.

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

Stanley, K. O., D'Ambrosio, D. B., and Gauci, J. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009.

Strub, F., Gaudel, R., and Mary, J. Hybrid recommender system based on autoencoders. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, pp. 11–16. ACM, 2016.

Tai, C., Xiao, T., Zhang, Y., Wang, X., et al. Convolutional neural networks with low-rank regularization. *arXiv preprint arXiv:1511.06067*, 2015.

van den Berg, R., Kipf, T. N., and Welling, M. Graph convolutional matrix completion. *stat*, 1050:7, 2017.

van der Maaten, L. Learning a parametric embedding by preserving local structure. *RBM*, 500(500):26, 2009.

Zheng, Y., Tang, B., Ding, W., and Zhou, H. A neural autoregressive approach to collaborative filtering. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning-Volume 48*, pp. 764–773. JMLR. org, 2016.

Zhu, C., Byrd, R. H., Lu, P., and Nocedal, J. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software (TOMS)*, 23(4):550–560, 1997.